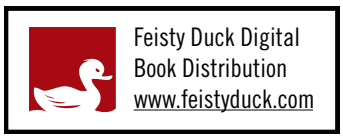
Programming in Lua, Fourth Edition

Roberto Ierusalimschy



Contents

List of Figures

About the Book

When Waldemar, Luiz, and I started the development of Lua, back in 1993, we could hardly imagine that it would spread as it did. Started as an in-house language for two specific projects, currently Lua is widely used in all areas that can benefit from a simple, extensible, portable, and efficient scripting language, such as embedded systems, mobile devices, the Internet of Things, and, of course, games.

We designed Lua, from the beginning, to be integrated with software written in C/C++ and other conventional languages. This integration brings many benefits. Lua is a small and simple language, partly because it does not try to do what C is already good for, such as sheer performance and interface with third-party software. Lua relies on C for these tasks. What Lua does offer is what C is not good for: a good distance from the hardware, dynamic structures, no redundancies, and ease of testing and debugging. For these goals, Lua has a safe environment, automatic memory management, and good facilities for handling strings and other kinds of data with dynamic size.

Part of the power of Lua comes from its libraries. This is not by chance. After all, one of the main strengths of Lua is its extensibility. Many features contribute to this strength. Dynamic typing allows a great degree of polymorphism. Automatic memory management simplifies interfaces, because there is no need to decide who is responsible for allocating and deallocating memory or how to handle overflows. First-class functions allow a high degree of parameterization, making functions more versatile.

More than an extensible language, Lua is also a *glue language*. Lua supports a component-based approach to software development, where we create an application by gluing together existing high-level components. These components are written in a compiled, statically-typed language, such as C or C++; Lua is the glue that we use to compose and connect these components. Usually, the components (or objects) represent more concrete, low-level concepts (such as widgets and data structures) that are not subject to many changes during program development, and that take the bulk of the CPU time of the final program. Lua gives the final shape of the application, which will probably change a lot during the life cycle of the product. We can use Lua not only to glue components, but also to adapt and reshape them, and to create completely new components.

Of course, Lua is not the only scripting language around. There are other languages that you can use for more or less the same purposes. Nevertheless, Lua offers a set of features that makes it your best choice for many tasks and gives it a unique profile:

Extensibility: Lua's extensibility is so remarkable that many people regard Lua not as a language, but as a kit for building domain-specific languages. We designed Lua from scratch to be extended, both through Lua code and through external C code. As a proof of concept, Lua implements most of its own basic functionality through external libraries. It is really easy to interface Lua with external languages like C/C++, Java, C#, and Python.

Simplicity: Lua is a simple and small language. It has few (but powerful) concepts. This simplicity makes Lua easy to learn and contributes to its small size. (Its Linux 64-bit executable, including all standard libraries, has 220 KB.)

Efficiency: Lua has a quite efficient implementation. Independent benchmarks show Lua as one of the fastest languages in the realm of scripting languages.

Portability: When we talk about portability, we are talking about running Lua on all platforms we have ever heard about: all flavors of UNIX (Linux, FreeBSD, etc.) Windows, Android, iOS, OS X, IBM mainframes, game consoles (PlayStation, Xbox, Wii, etc.), microcontrollers (Arduino, etc.), and many more. The source code for each of these platforms is virtually the same. Lua does not use conditional compilation

to adapt its code to different machines; instead, it sticks to the standard ISO (ANSI) C. This way, you do not usually need to adapt it to a new environment: if you have an ISO C compiler, you just have to compile Lua, out of the box.

Audience

This book does not assume any prior knowledge of Lua or any specific programming language —except for its last part, which discusses the Lua API with C. However, it assumes the knowledge of some basic programming concepts, in particular variables and assignment, control structures, functions and parameters, recursion, streams and files, and basic data structures.

Lua users typically fall into three broad groups: those that use Lua already embedded in an application program, those that use Lua stand alone, and those that use Lua and C together. This book has much to offer to all these groups.

Many people use Lua embedded in an application program, such as Adobe Lightroom, Nmap, or World of Warcraft. These applications use Lua's C API to register new functions, to create new types, and to change the behavior of some language operations, configuring Lua for their specific domains. Often, the users of such applications do not even know that Lua is an independent language adapted for a particular domain. For instance, many developers of plug-ins for Lightroom do not know about other uses of the language; Nmap users tend to think of Lua as the language of the Nmap Scripting Engine; many players of World of Warcraft regard Lua as a language exclusive to that game. Despite these different worlds, the core language is still the same, and the programming techniques you will learn here apply everywhere.

Lua is useful also as a stand-alone language, not only for text processing and one-shot little programs, but for medium-to-large projects, too. For such uses, the main functionality of Lua comes from libraries. The standard libraries, for instance, offer pattern matching and other functions for string handling. As Lua has improved its support for libraries, there has been a proliferation of external packages. LuaRocks, a deployment and management system for Lua modules, passed one thousand modules in 2015, covering all sorts of domains.

Finally, there are those programmers that work on the other side of the bench, writing applications that use Lua as a C library. Those people will program more in C than in Lua, although they need a good understanding of Lua to create interfaces that are simple, easy to use, and well integrated with the language.

Book Structure

This edition adds new material and examples in many areas, including sandboxing, coroutines, date and time manipulation, in addition to the new material related to version 5.3: integers, bitwise operations, unsigned integers, etc.

More importantly, this edition marks a major restructuring of the text. Instead of organizing the material around the language (e.g., with separate chapters for each library), I tried to organize the material around common themes in programming. That organization allows the book to better follow an order of increasing complexity, with simple themes coming first. That order came from experience teaching courses about the language; in particular, I think this new organization fits the book better as a didactic resource for courses involving Lua.

As the previous editions, this one is organized in four parts, each with around nine chapters. However, the parts have a quite new character.

The first part covers the basics of the language (and it is fittingly named *The Basics*). It is organized around the main types of values in Lua: numbers, strings, tables, and functions. It also covers basic I/O and gives an overview of the syntax of the language.

The second part, called *Real Programming*, covers more advanced topics that you can expect to find in other similar languages, such as closures, pattern matching, date and time manipulation, data structures, modules, and error handling.

The third part is called *Lua-isms*. As the name implies, it covers aspects of Lua that are particularly different from other languages, such as metatables and its uses, environments, weak tables, coroutines, and reflection. These are also the more advanced aspects of the language.

Finally, as in previous editions, the last part of the book covers the API between Lua and C, for those that use C to get the full power of Lua. The flavor of that part is necessarily quite different from the rest of the book. There, we will be programming in C, not in Lua; therefore, we will be wearing a different hat. For some readers, the discussion of the C API may be of marginal interest; for others, it may be the most relevant part of this book.

Along all parts, we focus on different language constructs and use numerous examples and exercises to show how to use them for practical tasks. We also have a few interludes among the chapters. Each interlude presents a short but complete program in Lua, which gives a more holistic view of the language.

Other Resources

The reference manual is a must for anyone who wants to really learn a language. This book does not replace the Lua reference manual; quite the opposite, it complements the manual. The manual only describes Lua. It shows neither examples nor a rationale for the constructs of the language. On the other hand, it describes the whole language; this book skips over seldom-used dark corners of Lua. Moreover, the manual is the authoritative document about Lua. Wherever this book disagrees with the manual, trust the manual. To get the manual and more information about Lua, visit the Lua site at `http://www.lua.org`.

You can also find useful information at the Lua users' site, kept by the community of users at `http://lua-users.org`. Among other resources, it offers a tutorial, a list of third-party packages and documentation, and an archive of the official Lua mailing list.

This book describes Lua 5.3, although most of its contents also apply to previous versions and probably to future versions as well. All differences between Lua 5.3 and older Lua 5 versions are clearly marked in the text. If you are using a more recent version (released after the book), check the corresponding manual for differences between versions.

A Few Typographical Conventions

The book encloses `"literal strings"` between double quotes and single characters, such as `a`, between single quotes. Strings that are used as patterns are also enclosed between single quotes, like `'[%w_]*'`. The book uses a `typewriter font` both for chunks of code and for identifiers. For reserved words, it uses a **boldface font**. Larger chunks of code are shown in display style:

```
-- program "Hello World"
print("Hello World")         --> Hello World
```

The notation `-->` shows the output of a statement or the result of an expression:

```
print(10)      --> 10
13 + 3         --> 16
```

Because a double hyphen (`--`) starts a comment in Lua, there is no problem if you include these annotations in your code.

Several code fragments in the book, mainly in the initial chapters, should be entered in interactive mode. In that case, I use a notation showing the Lua prompt (`"> "`) in each line:

```
> 3 + 5           --> 8
> math.sin(2.3)   --> 0.74570521217672
```

In Lua 5.2 and older versions, to print the result of an expression in interactive mode, you must precede the expression with an equals sign:

```
> = 3 + 5           --> 8
> a = 25
> = a               --> 25
```

For compatibility, Lua 5.3 still accepts this equal sign.

Finally, the book uses the notation `<-->` to indicate that something is equivalent to something else:

```
this     <-->     that
```

Running the Examples

You will need a Lua interpreter to run the examples in this book. Ideally, you should use Lua 5.3, but most of the examples run on older versions without modifications.

The Lua site (`http://www.lua.org`) keeps the source code for the interpreter. If you have a C compiler and a working knowledge of how to compile C code in your machine, you should try to install Lua from its source code; it is really easy. The *Lua Binaries* site (search for `luabinaries`) offers precompiled Lua interpreters for most major platforms. If you use Linux or another UNIX-like system, you may check the repository of your distribution; several distributions already offer a package with Lua.

There are several Integrated Development Environments (IDEs) for Lua. Again, you can easily find them with a basic search. (Nevertheless, I am an old timer. I still prefer a command-line interface in a window and a text editor in another, specially for the initial learning steps.)

Acknowledgments

It is more than ten years since I published the first edition of this book. Several friends and institutions have helped me along this journey.

As always, Luiz Henrique de Figueiredo and Waldemar Celes, Lua coauthors, offered all kinds of help. André Carregal, Asko Kauppi, Brett Kapilik, Diego Nehab, Edwin Moragas, Fernando Jefferson, Gavin Wraith, John D. Ramsdell, Norman Ramsey, Reuben Thomas, and Robert Day provided invaluable suggestions and useful insights for diverse editions of this book. Luiza Novaes provided key support for the cover design.

Lightning Source, Inc. proved a reliable and efficient option for printing and distributing the book. Without them, the option of self-publishing the book would not be an option.

Tecgraf, headed by Marcelo Gattass, housed the Lua project from its birth in 1993 until 2005, and continues to help the project in several ways.

I also would like to thank the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and the Brazilian National Research Council (CNPq) for their continuous support to my work. In particular, the Lua project would be impossible without the environment that I have at PUC-Rio.

Finally, I must express my deep gratitude to Noemi Rodriguez, for all kinds of help (technical and non-technical) and for illumining my life.

Part I. The Basics

Table of Contents

Chapter 1. Getting Started

To keep with tradition, our first program in Lua just prints `"Hello World"`:

```
print("Hello World")
```

If you are using the stand-alone Lua interpreter, all you have to do to run your first program is to call the interpreter —usually named `lua` or `lua5.3`— with the name of the text file that contains your program. If you save the above program in a file `hello.lua`, the following command should run it:

```
% lua hello.lua
```

As a more complex example, the next program defines a function to compute the factorial of a given number, asks the user for 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))
```

Chunks

We call each piece of code that Lua executes, such as a file or a single line in interactive mode, a *chunk*. A chunk is simply a sequence of commands (or statements).

A chunk can be as simple as a single statement, such as in the “Hello World” example, or it can be composed of a mix of statements and function definitions (which are actually assignments, as we will see later), such as the factorial example. A chunk can be as large as we wish. Because Lua is used also as a data-description language, chunks with several megabytes are not uncommon. The Lua interpreter has no problems at all with large chunks.

Instead of writing your program to a file, you can run the stand-alone interpreter in interactive mode. If you call `lua` without any arguments, you will get its prompt:

```
% lua
Lua 5.3  Copyright (C) 1994-2016 Lua.org, PUC-Rio
>
```

Thereafter, each command that you type (such as `print "Hello World"`) executes immediately after you enter it. To exit the interactive mode and the interpreter, just type the end-of-file control character (`ctrl-D` in POSIX, `ctrl-Z` in Windows), or call the function `os.exit`, from the Operating System library —you have to type `os.exit()`.

Starting in version 5.3, we can enter expressions directly in the interactive mode, and Lua will print their values:

```
% lua
Lua 5.3  Copyright (C) 1994-2016 Lua.org, PUC-Rio
> math.pi / 4       --> 0.78539816339745
> a = 15
> a^2               --> 225
> a + 2             --> 17
```

In older versions, we need to precede these expressions with an equals sign:

```
% lua5.2
Lua 5.2.3  Copyright (C) 1994-2013 Lua.org, PUC-Rio
> a = 15
> = a^2               --> 225
```

For compatibility, Lua 5.3 still accepts these equals signs.

To run that code as a chunk (not in interactive mode), we must enclose the expressions inside calls to `print`:

```
print(math.pi / 4)
a = 15
print(a^2)
print(a + 2)
```

Lua usually interprets each line that we type in interactive mode as a complete chunk or expression. However, if it detects that the line is not complete, it waits for more input, until it has a complete chunk. This way, we can enter a multi-line definition, such as the factorial function, directly in interactive mode. However, it is usually more convenient to put such definitions in a file and then call Lua to run the file.

We can use the `-i` option to instruct Lua to start an interactive session after running a given chunk:

```
% lua -i prog
```

A command line like this one will run the chunk in the file `prog` and then prompt for interaction. This is especially useful for debugging and manual testing. At the end of this chapter, we will see other options for the stand-alone interpreter.

Another way to run chunks is with the function `dofile`, which immediately executes a file. For instance, suppose we have a file `lib1.lua` with the following code:

```
function norm (x, y)
  return math.sqrt(x^2 + y^2)
end

function twice (x)
  return 2.0 * x
end
```

Then, in interactive mode, we can type this code:

```
> dofile("lib1.lua")     -- load our library
> n = norm(3.4, 1.0)
> twice(n)               --> 7.0880180586677
```

The function `dofile` is useful also when we are testing a piece of code. We can work with two windows: one is a text editor with our program (in a file `prog.lua`, say) and the other is a console running Lua

in interactive mode. After saving a modification in our program, we execute `dofile("prog.lua")` in the Lua console to load the new code; then we can exercise the new code, calling its functions and printing the results.

Some Lexical Conventions

Identifiers (or names) in Lua can be any string of letters, digits, and underscores, not beginning with a digit; for instance

```
i       j       i10       _ij
aSomewhatLongName     _INPUT
```

You should avoid identifiers starting with an underscore followed by one or more upper-case letters (e.g., `_VERSION`); they are reserved for special uses in Lua. Usually, I reserve the identifier _ (a single underscore) for dummy variables.

The following words are reserved; we cannot use them as identifiers:

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

Lua is case-sensitive: **and** is a reserved word, but `And` and `AND` are two different identifiers.

A comment starts anywhere with two consecutive hyphens (`--`) and runs until the end of the line. Lua also offers long comments, which start with two hyphens followed by two opening square brackets and run until the first occurrence of two consecutive closing square brackets, like here:[1]

```
--[[A multi-line
    long comment
]]
```

A common trick that we use to comment out a piece of code is to enclose the code between `--[[` and `--]]`, like here:

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

To reactivate the code, we add a single hyphen to the first line:

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

In the first example, the `--[[` in the first line starts a long comment, and the two hyphens in the last line are still inside that comment. In the second example, the sequence `---[[` starts an ordinary, single-line comment, so that the first and the last lines become independent comments. In this case, the `print` is outside comments.

Lua needs no separator between consecutive statements, but we can use a semicolon if we wish. Line breaks play no role in Lua's syntax; for instance, the following four chunks are all valid and equivalent:

[1] Long comments can be more complex than that, as we will see in the section called “Long strings”.

```
a = 1
b = a * 2

a = 1;
b = a * 2;

a = 1; b = a * 2

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

My personal convention is to use semicolons only when I write two or more statements in the same line (which I hardly do).

Global Variables

Global variables do not need declarations; we simply use them. It is not an error to access a non-initialized variable; we just get the value nil as the result:

```
> b          --> nil
> b = 10
> b          --> 10
```

If we assign nil to a global variable, Lua behaves as if we have never used the variable:

```
> b = nil
> b          --> nil
```

Lua does not differentiate a non-initialized variable from one that we assigned nil. After the assignment, Lua can eventually reclaim the memory used by the variable.

Types and Values

Lua is a dynamically-typed language. There are no type definitions in the language; each value carries its own type.

There are eight basic types in Lua: *nil*, *Boolean*, *number*, *string*, *userdata*, *function*, *thread*, and *table*. The function `type` gives the type name of any given value:

```
> type(nil)             --> nil
> type(true)            --> boolean
> type(10.4 * 3)        --> number
> type("Hello world")   --> string
> type(io.stdin)        --> userdata
> type(print)           --> function
> type(type)            --> function
> type({})              --> table
> type(type(X))         --> string
```

The last line will result in `"string"` no matter the value of `X`, because the result of `type` is always a string.

The userdata type allows arbitrary C data to be stored in Lua variables. It has no predefined operations in Lua, except assignment and equality test. Userdata are used to represent new types created by an application program or a library written in C; for instance, the standard I/O library uses them to represent open files. We will discuss more about userdata later, when we get to the C API.

Variables have no predefined types; any variable can contain values of any type:

```
> type(a)           --> nil   ('a' is not initialized)
> a = 10
> type(a)           --> number
> a = "a string!!"
> type(a)           --> string
> a = nil
> type(a)           --> nil
```

Usually, when we use a single variable for different types, the result is messy code. However, sometimes the judicious use of this facility is helpful, for instance in the use of nil to differentiate a normal return value from an abnormal condition.

We will discuss now the simple types nil and Boolean. In the following chapters, we will discuss in detail the types number (Chapter 3, *Numbers*), string (Chapter 4, *Strings*), table (Chapter 5, *Tables*), and function (Chapter 6, *Functions*). We will explain the thread type in Chapter 24, *Coroutines*, where we discuss coroutines.

Nil

Nil is a type with a single value, nil, whose main property is to be different from any other value. Lua uses nil as a kind of non-value, to represent the absence of a useful value. As we have seen, a global variable has a nil value by default, before its first assignment, and we can assign nil to a global variable to delete it.

Booleans

The Boolean type has two values, *@false{}* and *@true{}*, which represent the traditional Boolean values. However, Booleans do not hold a monopoly of condition values: in Lua, any value can represent a condition. Conditional tests (e.g., conditions in control structures) consider both the Boolean **false** and nil as false and anything else as true. In particular, Lua considers both zero and the empty string as true in conditional tests.

Throughout this book, I will write “false” to mean any false value, that is, the Boolean **false** or nil. When I mean specifically the Boolean value, I will write “**false**”. The same holds for “true” and “**true**”.

Lua supports a conventional set of logical operators: **and**, **or**, and **not**. Like control structures, all logical operators consider both the Boolean **false** and nil as false, and anything else as true. The result of the **and** operator is its first operand if that operand is false; otherwise, the result is its second operand. The result of the **or** operator is its first operand if it is not false; otherwise, the result is its second operand:

```
> 4 and 5         --> 5
> nil and 13      --> nil
> false and 13    --> false
> 0 or 5          --> 0
> false or "hi"   --> "hi"
> nil or false    --> false
```

Both **and** and **or** use short-circuit evaluation, that is, they evaluate their second operand only when necessary. Short-circuit evaluation ensures that expressions like `(i ~= 0 and a/i > b)` do not cause run-time errors: Lua will not try to evaluate `a / i` when `i` is zero.

A useful Lua idiom is `x = x or v`, which is equivalent to

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

That is, it sets `x` to a default value `v` when `x` is not set (provided that `x` is not set to **false**).

Another useful idiom is `((a and b) or c)` or simply `(a and b or c)` (given that **and** has a higher precedence than **or**). It is equivalent to the C expression `a ? b : c`, provided that `b` is not false. For instance, we can select the maximum of two numbers `x` and `y` with the expression `(x > y) and x or y`. When `x > y`, the first expression of the **and** is true, so the **and** results in its second operand (`x`), which is always true (because it is a number), and then the **or** expression results in the value of its first operand, `x`. When `x > y` is false, the **and** expression is false and so the **or** results in its second operand, `y`.

The **not** operator always gives a Boolean value:

```
> not nil      --> true
> not false    --> true
> not 0        --> false
> not not 1    --> true
> not not nil  --> false
```

The Stand-Alone Interpreter

The stand-alone interpreter (also called `lua.c` due to its source file or simply `lua` due to its executable) is a small program that allows the direct use of Lua. This section presents its main options.

When the interpreter loads a file, it ignores its first line if this line starts with a hash (#). This feature allows the use of Lua as a script interpreter in POSIX systems. If we start our script with something like

```
#!/usr/local/bin/lua
```

(assuming that the stand-alone interpreter is located at `/usr/local/bin`), or

```
#!/usr/bin/env lua
```

then we can call the script directly, without explicitly calling the Lua interpreter.

The usage of `lua` is

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

Everything is optional. As we have seen already, when we call `lua` without arguments the interpreter enters the interactive mode.

The `-e` option allows us to enter code directly into the command line, like here:

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

(POSIX systems need the double quotes to stop the shell from interpreting the parentheses.)

The `-l` option loads a library. As we saw previously, `-i` enters interactive mode after running the other arguments. Therefore, the next call will load the `lib` library, then execute the assignment `x = 10`, and finally present a prompt for interaction.

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

If we write an expression in interactive mode, Lua prints its value:

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

(Remember, this feature came with Lua 5.3. In older versions, we must precede the expressions with equals signs.) To avoid this print, we can finish the line with a semicolon:

```
> io.flush()                --> true
> io.flush();
```

The semicolon makes the line syntactically invalid as an expression, but still valid as a command.

Before running its arguments, the interpreter looks for an environment variable named `LUA_INIT_5_3` or else, if there is no such variable, `LUA_INIT`. If there is one of these variables and its content is `@`*`file-name`*, then the interpreter runs the given file. If `LUA_INIT_5_3` (or `LUA_INIT`) is defined but it does not start with an at-sign, then the interpreter assumes that it contains Lua code and runs it. `LUA_INIT` gives us great power when configuring the stand-alone interpreter, because we have the full power of Lua in the configuration. We can preload packages, change the path, define our own functions, rename or delete functions, and so on.

A script can retrieve its arguments through the predefined global variable `arg`. In a call like `% lua script a b c`, the interpreter creates the table `arg` with all the command-line arguments, before running any code. The script name goes into index 0; its first argument (`"a"` in the example) goes to index 1, and so on. Preceding options go to negative indices, as they appear before the script. For instance, consider this call:

```
% lua -e "sin=math.sin" script a b
```

The interpreter collects the arguments as follows:

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

More often than not, a script uses only the positive indices (`arg[1]` and `arg[2]`, in the example).

A script can also retrieve its arguments through a vararg expression. In the main body of a script, the expression `...` (three dots) results in the arguments to the script. (We will discuss vararg expressions in the section called “Variadic Functions”.)

Exercises

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

Exercise 1.2: Run the `twice` example, both by loading the file with the `-l` option and with `dofile`. Which way do you prefer?

Exercise 1.3: Can you name other languages that use `"--"` for comments?

Exercise 1.4: Which of the following strings are valid identifiers?

```
___   _end   End   end   until?   nil   NULL   one-step
```

Exercise 1.5: What is the value of the expression `type(nil) == nil`? (You can use Lua to check your answer.) Can you explain this result?

Exercise 1.6: How can you check whether a value is a Boolean without using the function `type`?

Exercise 1.7: Consider the following expression:

```
(x and y and (not z)) or ((not y) and x)
```

Are the parentheses necessary? Would you recommend their use in that expression?

Exercise 1.8: Write a simple script that prints its own name without knowing it in advance.

Chapter 2. Interlude: The Eight-Queen Puzzle

In this chapter we make a short interlude to present a simple but complete program in Lua that solves the *eight-queen puzzle*: its goal is to position eight queens in a chessboard in such a way that no queen can attack another one.

The code here does not use anything specific to Lua; we should be able to translate the code to several other languages with only cosmetic changes. The idea is to present the general flavor of Lua, in particular how the Lua syntax looks like, without going into details. We will cover all missing details in subsequent chapters.

A first step to solving the eight-queen puzzle is to note that any valid solution must have exactly one queen in each row. Therefore, we can represent potential solutions with a simple array of eight numbers, one for each row; each number tells at which column is the queen at that row. For instance, the array `{3, 7, 2, 1, 8, 6, 5, 4}` means that the queens are in the squares *(1,3)*, *(2,7)*, *(3,2)*, *(4,1)*, *(5,8)*, *(6,6)*, *(7,5)*, and *(8,4)*. (By the way, this is not a valid solution; for instance, the queen in square *(3,2)* can attack the one in square *(4,1)*.) Note that any valid solution must be a permutation of the integers 1 to 8, as a valid solution also must have exactly one queen in each column.

The complete program is in Figure 2.1, “The eight-queen program”.

Figure 2.1. The eight-queen program

```
N = 8    -- board size

-- check whether position (n,c) is free from attacks
function isplaceok (a, n, c)
  for i = 1, n - 1 do   -- for each queen already placed
    if (a[i] == c) or               -- same column?
       (a[i] - i == c - n) or       -- same diagonal?
       (a[i] + i == c + n) then     -- same diagonal?
      return false            -- place can be attacked
    end
  end
  return true    -- no attacks; place is OK
end

-- print a board
function printsolution (a)
  for i = 1, N do      -- for each row
    for j = 1, N do    -- and for each column
      -- write "X" or "-" plus a space
      io.write(a[i] == j and "X" or "-", " ")
    end
    io.write("\n")
  end
  io.write("\n")
end

-- add to board 'a' all queens from 'n' to 'N'
function addqueen (a, n)
  if n > N then    -- all queens have been placed?
    printsolution(a)
  else  -- try to place n-th queen
    for c = 1, N do
      if isplaceok(a, n, c) then
        a[n] = c    -- place n-th queen at column 'c'
        addqueen(a, n + 1)
      end
    end
  end
end

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

The first function is `isplaceok`, which checks whether a given position on a board is free from attacks from previously placed queens. More specifically, it checks whether putting the `n`-th queen in column `c` will conflict with any of the previous `n-1` queens already set in the array `a`. Remember that, by representation, two queens cannot be in the same row, so `isplaceok` checks whether there are no queens in the same column or in the same diagonals of the new position.

Next we have the function `printsolution`, which prints a board. It simply traverses the entire board, printing an `X` at positions with a queen and a `-` at other positions, without any fancy graphics. (Note its use of the **and**–**or** idiom to select the character to print at each position.) Each result will look like this:

```
X - - - - - - -
- - - - X - - -
- - - - - - - X
- - - - - X - -
- - X - - - - -
- - - - - - X -
- X - - - - - -
- - - X - - - -
```

The last function, `addqueen`, is the core of the program. It tries to place all queens larger than or equal to `n` in the board. It uses backtracking to search for valid solutions. First, it checks whether the solution is complete and, if so, prints that solution. Otherwise, it loops through all columns for the `n`-th queen; for each column that is free from attacks, the program places the queen there and recursively tries to place the following queens.

Finally, the main body simply calls `addqueen` on an empty solution.

Exercises

Exercise 2.1: Modify the eight-queen program so that it stops after printing the first solution.

Exercise 2.2: An alternative implementation for the eight-queen problem would be to generate all possible permutations of 1 to 8 and, for each permutation, to check whether it is valid. Change the program to use this approach. How does the performance of the new program compare with the old one? (Hint: compare the total number of permutations with the number of times that the original program calls the function `isplaceok`.)

Chapter 3. Numbers

Until version 5.2, Lua represented all numbers using double-precision floating-point format. Starting with version 5.3, Lua uses two alternative representations for numbers: 64-bit integer numbers, called simply *integers*, and double-precision floating-point numbers, called simply *floats*. (Note that, in this book, the term "float" does not imply single precision.) For restricted platforms, we can compile Lua 5.3 as *Small Lua*, which uses 32-bit integers and single-precision floats.[1]

The introduction of integers is the hallmark of Lua 5.3, its main difference against previous versions of Lua. Nevertheless, this change created few incompatibilities, because double-precision floating-point numbers can represent integers exactly up to 2^{53}. Most of the material we will present here is valid for Lua 5.2 and older versions, too. In the end of this chapter I will discuss in more detail the incompatibilities.

Numerals

We can write numeric constants with an optional decimal part plus an optional decimal exponent, like these examples:

```
> 4                  --> 4
> 0.4                --> 0.4
> 4.57e-3            --> 0.00457
> 0.3e12             --> 300000000000.0
> 5E+20              --> 5e+20
```

Numerals with a decimal point or an exponent are considered floats; otherwise, they are treated as integers.

Both integer and float values have type `"number"`:

```
> type(3)    --> number
> type(3.5)  --> number
> type(3.0)  --> number
```

They have the same type because, more often than not, they are interchangeable. Moreover, integers and floats with the same value compare as equal in Lua:

```
> 1 == 1.0      --> true
> -3 == -3.0    --> true
> 0.2e3 == 200  --> true
```

In the rare occasions when we need to distinguish between floats and integers, we can use `math.type`:

```
> math.type(3)     --> integer
> math.type(3.0)   --> float
```

Moreover, Lua 5.3 shows them differently:

```
> 3          --> 3
> 3.0        --> 3.0
> 1000       --> 1000
> 1e3        --> 1000.0
```

Like many other programming languages, Lua supports hexadecimal constants, by prefixing them with `0x`. Unlike many other programming languages, Lua supports also floating-point hexadecimal constants,

[1] We create Small Lua from the same source files of Standard Lua, compiling them with the macro `LUA_32BITS` defined. Except for the sizes for number representations, Small Lua is identical to Standard Lua.

which can have a fractional part and a binary exponent, prefixed by `p` or `P`.[2] The following examples illustrate this format:

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

Lua can write numbers in this format using `string.format` with the `%a` option:

```
> string.format("%a", 419)       --> 0x1.a3p+8
> string.format("%a", 0.1)       --> 0x1.999999999999ap-4
```

Although not very friendly to humans, this format preserves the full precision of any float value, and the conversion is faster than with decimals.

Arithmetic Operators

Lua presents the usual set of arithmetic operators: addition, subtraction, multiplication, division, and negation (unary minus). It also supports floor division, modulo, and exponentiation.

One of the main guidelines for the introduction of integers in Lua 5.3 was that *"the programmer may choose to mostly ignore the difference between integers and floats or to assume complete control over the representation of each number."*[3] Therefore, any arithmetic operator should give the same result when working on integers and when working on reals.

The addition of two integers is always an integer. The same is true for subtraction, multiplication, and negation. For those operations, it does not matter whether the operands are integers or floats with integral values (except in case of overflows, which we will discuss in the section called "Representation Limits"); the result is the same in both cases:

```
> 13 + 15             --> 28
> 13.0 + 15.0         --> 28.0
```

If both operands are integers, the operation gives an integer result; otherwise, the operation results in a float. In case of mixed operands, Lua converts the integer one to a float before the operation:

```
> 13.0 + 25       --> 38.0
> -(3 * 6.0)      --> -18.0
```

Division does not follow that rule, because the division of two integers does not need to be an integer. (In mathematical terms, we say that the integers are not closed under division.) To avoid different results between division of integers and divisions of floats, division always operates on floats and gives float results:

```
> 3.0 / 2.0      --> 1.5
> 3 / 2          --> 1.5
```

For integer division, Lua 5.3 introduced a new operator, called *floor division* and denoted by `//`. As its name implies, floor division always rounds the quotient towards minus infinity, ensuring an integral result for all operands. With this definition, this operation can follow the same rule of the other arithmetic operators: if both operands are integers, the result is an integer; otherwise, the result is a float (with an integral value):

[2] This feature was introduced in Lua 5.2.
[3] From the Lua 5.3 Reference Manual.

```
> 3 // 2         --> 1
> 3.0 // 2       --> 1.0
> 6 // 2         --> 3
> 6.0 // 2.0     --> 3.0
> -9 // 2        --> -5
> 1.5 // 0.5     --> 3.0
```

The following equation defines the modulo operator:

```
a % b == a - ((a // b) * b)
```

Integral operands ensure integral results, so this operator also follows the rule of other arithmetic operations: if both operands are integers, the result is an integer; otherwise, the result is a float.

For integer operands, modulo has the usual meaning, with the result always having the same sign as the second argument. In particular, for any given positive constant `K`, the result of the expression `x % K` is always in the range *[0,K-1]*, even when `x` is negative. For instance, `i % 2` always results in 0 or 1, for any integer `i`.

For real operands, modulo has some unexpected uses. For instance, `x - x % 0.01` is `x` with exactly two decimal digits, and `x - x % 0.001` is `x` with exactly three decimal digits:

```
> x = math.pi
> x - x%0.01        --> 3.14
> x - x%0.001       --> 3.141
```

As another example of the use of the modulo operator, suppose we want to check whether a vehicle turning a given angle will start to backtrack. If the angle is in degrees, we 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 we want to work with radians instead of degrees, we simply change the constants in our function:

```
local tolerance = 0.17
function isturnback (angle)
  angle = angle % (2*math.pi)
  return (math.abs(angle - math.pi) < tolerance)
end
```

The operation `angle % (2 * math.pi)` is all we need to normalize any angle to a value in the interval *[0, 2#)*.

Lua also offers an exponentiation operator, denoted by a caret (^). Like division, it always operates on floats. (Integers are not closed under exponentiation; for instance, 2^{-2} is not an integer.) We can write `x^0.5` to compute the square root of `x` and `x^(1/3)` to compute its cubic root.

Relational Operators

Lua provides the following relational operators:

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

All these operators always produce a Boolean value.

The `==` operator tests for equality; the `~=` operator is the negation of equality. We can apply these operators to any two values. If the values have different types, Lua considers them not equal. Otherwise, Lua compares them according to their types.

Comparison of numbers always disregards their subtypes; it makes no difference whether the number is represented as an integer or as a float. What matters is its mathematical value. (Nevertheless, it is slightly more efficient to compare numbers with the same subtypes.)

The Mathematical Library

Lua provides a standard `math` library with a set of mathematical functions, including trigonometric functions (`sin`, `cos`, `tan`, `asin`, etc.), logarithms, rounding functions, `max` and `min`, a function for generating pseudo-random numbers (`random`), plus the constants `pi` and `huge` (the largest representable number, which is the special value *inf* on most platforms.)

```
> math.sin(math.pi / 2)         --> 1.0
> math.max(10.4, 7, -3, 20)     --> 20
> math.huge                     --> inf
```

All trigonometric functions work in radians. We can use the functions `deg` and `rad` to convert between degrees and radians.

Random-number generator

The `math.random` function generates pseudo-random numbers. We can call it in three ways. When we call it without arguments, it returns a pseudo-random real number with uniform distribution in the interval *[0,1)*. When we call it with only one argument, an integer *n*, it returns a pseudo-random integer in the interval *[1,n]*. For instance, we can simulate the result of tossing a die with the call `random(6)`. Finally, we can call `random` with two integer arguments, *l* and *u*, to get a pseudo-random integer in the interval *[l,u]*.

We can set a seed for the pseudo-random generator with the function `randomseed`; its numeric sole argument is the seed. When a program starts, the system initializes the generator with the fixed seed 1. Without another seed, every run of a program will generate the same sequence of pseudo-random numbers. For debugging, this is a nice property; but in a game, we will have the same scenario over and over. A common trick to solve this problem is to use the current time as a seed, with the call `math.randomseed(os.time())`. (We will see `os.time` in the section called “The Function `os.time`”.)

Rounding functions

The math library offers three rounding functions: `floor`, `ceil`, and `modf`. Floor rounds towards minus infinite, ceil rounds towards plus infinite, and `modf` rounds towards zero. They return an integer result if it fits in an integer; otherwise, they return a float (with an integral value, of course). The function `modf`, besides the rounded value, also returns the fractional part of the number as a second result.[4]

```
> math.floor(3.3)           --> 3
> math.floor(-3.3)          --> -4
```

[4] As we will discuss in the section called “Multiple Results”, a function in Lua can return multiple values.

```
> math.ceil(3.3)             --> 4
> math.ceil(-3.3)            --> -3
> math.modf(3.3)             --> 3     0.3
> math.modf(-3.3)            --> -3    -0.3
> math.floor(2^70)           --> 1.1805916207174e+21
```

If the argument is already an integer, it is returned unaltered.

If we want to round a number `x` to the nearest integer, we could compute the floor of `x + 0.5`. However, this simple addition can introduce errors when the argument is a large integral value. For instance, consider the next fragment:

```
x = 2^52 + 1
print(string.format("%d %d", x, math.floor(x + 0.5)))
    --> 4503599627370497 4503599627370498
```

What happens is that $2^{52} + 1.5$ does not have an exact representation as a float, so it is internally rounded in a way that we cannot control. To avoid this problem, we can treat integral values separately:

```
function round (x)
  local f = math.floor(x)
  if x == f then return f
  else return math.floor(x + 0.5)
  end
end
```

The previous function will always round half-integers up (e.g., *2.5* will be rounded to *3*). If we want unbiased rounding (that rounds half-integers to the nearest even integer), our formula fails when `x + 0.5` is an odd integer:

```
> math.floor(3.5 + 0.5)    --> 4   (ok)
> math.floor(2.5 + 0.5)    --> 3   (wrong)
```

Again, the modulo operator for floats shows its usefulness: the test `(x % 2.0 == 0.5)` is true exactly when `x + 0.5` is an odd integer, that is, when our formula would give a wrong result. Based on this fact, it is easy to define a function that does unbiased rounding:

```
function round (x)
  local f = math.floor(x)
  if (x == f) or (x % 2.0 == 0.5) then
    return f
  else
    return math.floor(x + 0.5)
  end
end

print(round(2.5))         --> 2
print(round(3.5))         --> 4
print(round(-2.5))        --> -2
print(round(-1.5))        --> -2
```

Representation Limits

Most programming languages represent numbers with some fixed number of bits. Therefore, those representations have limits, both in range and in precision.

Standard Lua uses 64-bit integers. Integers with 64 bits can represent values up to $2^{63} - 1$, roughly 10^{19}. (Small Lua uses 32-bit integers, which can count up to two billions, approximately.) The math library defines constants with the maximum (`math.maxinteger`) and the minimum (`math.mininteger`) values for an integer.

This maximum value for a 64-bit integer is a large number: it is thousands times the total wealth on earth counted in cents of dollars and one billion times the world population. Despite this large value, overflows occur. When we compute an integer operation that would result in a value smaller than `mininteger` or larger than `maxinteger`, the result *wraps around*.

In mathematical terms, to wrap around means that the computed result is the only number between `mininteger` and `maxinteger` that is equal modulo 2^{64} to the mathematical result. In computational terms, it means that we throw away the last carry bit. (This last carry bit would increment a hypothetical 65th bit, which represents 2^{64}. Thus, to ignore this bit does not change the modulo 2^{64} of the value.) This behavior is consistent and predictable in all arithmetic operations with integers in Lua:

```
> math.maxinteger + 1 == math.mininteger      --> true
> math.mininteger - 1 == math.maxinteger      --> true
> -math.mininteger == math.mininteger         --> true
> math.mininteger // -1 == math.mininteger    --> true
```

The maximum representable integer is `0x7ff...fff`, that is, a number with all bits set to one except the highest bit, which is the signal bit (zero means a non-negative number). When we add one to that number, it becomes `0x800...000`, which is the minimum representable integer. The minimum integer has a magnitude one larger than the magnitude of the maximum integer, as we can see here:

```
> math.maxinteger        -->  9223372036854775807
> 0x7fffffffffffffff     -->  9223372036854775807
> math.mininteger        --> -9223372036854775808
> 0x8000000000000000     --> -9223372036854775808
```

For floating-point numbers, Standard Lua uses double precision. It represents each number with 64 bits, 11 of which are used for the exponent. Double-precision floating-point numbers can represent numbers with roughly 16 significant decimal digits, in a range from -10^{308} to 10^{308}. (Small Lua uses single-precision floats, with 32 bits. In this case, the range is from -10^{38} to 10^{38}, with roughly seven significant decimal digits.)

The range of double-precision floats is large enough for most practical applications, but we must always acknowledge the limited precision. The situation here is not different from what happens with pen and paper. If we use ten digits to represent a number, `1/7` becomes rounded to *0.142857142*. If we compute `1/7 * 7` using ten digits, the result will be *0.999999994*, which is different from *1*. Moreover, numbers that have a finite representation in decimal can have an infinite representation in binary. For instance, `12.7 - 20 + 7.3` is not exactly zero even when computed with double precision, because both `12.7` and `7.3` do not have an exact finite representation in binary (see Exercise 3.5).

Because integers and floats have different limits, we can expect that arithmetic operations will give different results for integers and floats when the results reach these limits:

```
> math.maxinteger + 2            --> -9223372036854775807
> math.maxinteger + 2.0          --> 9.2233720368548e+18
```

In this example, both results are mathematically incorrect, but in quite different ways. The first line makes an integer addition, so the result wraps around. The second line makes a float addition, so the result is rounded to an approximate value, as we can see in the following equality:

```
> math.maxinteger + 2.0 == math.maxinteger + 1.0   --> true
```

Each representation has its own strengths. Of course, only floats can represent fractional numbers. Floats have a much larger range, but the range where they can represent integers exactly is restricted to $[-2^{53},2^{53}]$. (Those are quite large numbers nevertheless.) Up to these limits, we can mostly ignore the differences between integers and floats. Outside these limits, we should think more carefully about the representations we are using.

Conversions

To force a number to be a float, we can simply add `0.0` to it. An integer always can be converted to a float:

```
> -3 + 0.0                      --> -3.0
> 0x7fffffffffffff + 0.0        --> 9.2233720368548e+18
```

Any integer up to 2^{53} (which is 9007199254740992) has an exact representation as a double-precision floating-point number. Integers with larger absolute values may lose precision when converted to a float:

```
> 9007199254740991 + 0.0 == 9007199254740991     --> true
> 9007199254740992 + 0.0 == 9007199254740992     --> true
> 9007199254740993 + 0.0 == 9007199254740993     --> false
```

In the last line, the conversion rounds the integer $2^{53}+1$ to the float 2^{53}, breaking the equality.

To force a number to be an integer, we can OR it with zero:[5]

```
> 2^53               --> 9.007199254741e+15      (float)
> 2^53 | 0           --> 9007199254740992        (integer)
```

Lua does this kind of conversion only when the number has an exact representation as an integer, that is, it has no fractional part and it is inside the range of integers. Otherwise, Lua raises an error:

```
> 3.2 | 0       -- fractional part
stdin:1: number has no integer representation
> 2^64 | 0      -- out of range
stdin:1: number has no integer representation
> math.random(1, 3.5)
stdin:1: bad argument #2 to 'random'
                      (number has no integer representation)
```

To round a fractional number, we must explicitly call a rounding function.

Another way to force a number into an integer is to use `math.tointeger`, which returns nil when the number cannot be converted:

```
> math.tointeger(-258.0)   --> -258
> math.tointeger(2^30)     --> 1073741824
> math.tointeger(5.01)     --> nil      (not an integral value)
> math.tointeger(2^64)     --> nil      (out of range)
```

This function is particularly useful when we need to check whether the number can be converted. As an example, the following function converts a number to integer when possible, leaving it unchanged otherwise:

```
function cond2int (x)
  return math.tointeger(x) or x
```

[5] Bitwise operations are new in Lua 5.3. We will discuss them in the section called “Bitwise Operators”.

```
end
```

Precedence

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

```
^
unary operators  (-   #   ~   not)
*    /    //   %
+    -
..                (concatentation)
<<   >>           (bitwise shifts)
&                 (bitwise AND)
~                 (bitwise exclusive OR)
|                 (bitwise OR)
<    >    <=   >=   ~=   ==
and
or
```

All binary operators are left associative, except for exponentiation and concatenation, which are right associative. Therefore, the following expressions on the left are equivalent to those 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 explicit parentheses. It is easier than looking it up in the manual and others will probably have the same doubt when reading your code.

Lua Before Integers

Not by chance, the introduction of integers in Lua 5.3 created few incompatibilities with previous Lua versions. As I said, programmers can mostly ignore the difference between integers and floats. When we ignore these differences, we also can ignore the differences between Lua 5.3 and Lua 5.2, where all numbers are floats. (Regarding numbers, Lua 5.0 and Lua 5.1 are exactly like Lua 5.2.)

Of course, the main incompatibility between Lua 5.3 and Lua 5.2 is the representation limits for integers. Lua 5.2 can represent exact integers only up to 2^{53}, while in Lua 5.3 the limit is 2^{63}. When counting things, this difference is seldom an issue. However, when the number represents some generic bit pattern (e.g., three 20-bit integers packed together), the difference can be crucial.

Although Lua 5.2 did not support integers, they sneaked into the language in several ways. For instance, library functions implemented in C often get integer arguments. Lua 5.2 does not specify how it converts floats to integers in these places: the manual says only that "[the number] is truncated in some non-specified way". This is not a hypothetical issue; Lua 5.2 indeed can convert -3.2 to -3 or -4, depending on the platform. Lua 5.3, on the other hand, defines precisely these conversions, doing them only when the number has an exact integer representation.

Lua 5.2 does not offer the function `math.type`, as all numbers have the same subtype. Lua 5.2 does not offer the constants `math.maxinteger` and `math.mininteger`, as it has no integers. Lua 5.2 also does not offer floor division, although it could. (After all, its modulo operator is already defined in terms of floor division.)

Surprisingly, the main source of problems related to the introduction of integers was how Lua converts numbers to strings. Lua 5.2 formats any integral value as an integer, without a decimal point. Lua 5.3 formats all floats as floats, either with a decimal point or an exponent. So, Lua 5.2 formats `3.0` as `"3"`, while Lua 5.3 formats it as `"3.0"`. Although Lua has never specified how it formats numbers in conversions, many programs relied on the previous behavior. We can fix this kind of problem by using an explicit format when converting numbers to strings. However, more often than not, this problem indicates a deeper flaw somewhere else, where an integer becomes a float with no good reason. (In fact, this was the main motivation for the new format rules in version 5.3. Integral values being represented as floats usually is a bad smell in a program. The new format rule exposes these smells.)

Exercises

Exercise 3.1: Which of the following are valid numerals? What are their values?

```
.0e12   .e12   0.0e   0x12   0xABFG   0xA   FFFF   0xFFFFFFFF
0x   0x1P10   0.1e1   0x0.1p1
```

Exercise 3.2: Explain the following results:

```
> math.maxinteger * 2                  --> -2
> math.mininteger * 2                  --> 0
> math.maxinteger * math.maxinteger    --> 1
> math.mininteger * math.mininteger    --> 0
```

(Remember that integer arithmetic always wraps around.)

Exercise 3.3: What will the following program print?

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

Exercise 3.4: What is the result of the expression `2^3^4`? What about `2^-3^4`?

Exercise 3.5: The number *12.7* is equal to the fraction *127/10*, where the denominator is a power of ten. Can you express it as a common fraction where the denominator is a power of two? What about the number *5.5*?

Exercise 3.6: Write a function to compute the volume of a right circular cone, given its height and the angle between a generatrix and the axis.

Exercise 3.7: Using `math.random`, write a function to produce a pseudo-random number with a standard normal (Gaussian) distribution.

Chapter 4. Strings

Strings represent text. A string in Lua can contain a single letter or an entire book. Programs that manipulate strings with 100K or 1M characters are not unusual in Lua.

Strings in Lua are sequences of bytes. The Lua core is agnostic about how these bytes encode text. Lua is eight-bit clean and its strings can contain bytes with any numeric code, including embedded zeros. This means that we can store any binary data into a string. We can also store Unicode strings in any representation (UTF-8, UTF-16, etc.); however, as we will discuss, there are several good reasons to use UTF-8 whenever possible. The standard string library that comes with Lua assumes one-byte characters, but it can handle UTF-8 strings quite reasonably. Moreover, since version 5.3, Lua comes with a small library to help the use of UTF-8 encoding.

Strings in Lua are immutable values. We cannot change a character inside a string, as we can in C; instead, we create a new string with the desired modifications, as in the next example:

```
a = "one string"
b = string.gsub(a, "one", "another")  -- change string parts
print(a)       --> one string
print(b)       --> another string
```

Strings in Lua are subject to automatic memory management, like all other Lua objects (tables, functions, etc.). This means that we do not have to worry about allocation and deallocation of strings; Lua handles it for us.

We can get the length of a string using the *length operator* (denoted by #):

```
a = "hello"
print(#a)             --> 5
print(#"good bye")    --> 8
```

This operator always counts the length in bytes, which is not the same as characters in some encodings.

We can concatenate two strings with the concatenation operator `..` (two dots). If any operand is a number, Lua converts this number to a string:

```
> "Hello " .. "World"     --> Hello World
> "result is " .. 3       --> result is 3
```

(Some languages use the plus sign for concatenation, but `3 + 5` is different from `3 .. 5`.)

Remember that strings in Lua are immutable values. The concatenation operator always creates a new string, without any modification to its operands:

```
> a = "Hello"
> a .. " World"           --> Hello World
> a                       --> Hello
```

Literal strings

We can delimit literal strings by single or double matching quotes:

```
a = "a line"
b = 'another line'
```

They are equivalent; the only difference is that inside each kind of quote we can use the other quote without escapes.

As a matter of style, most programmers always use the same kind of quotes for the same kind of strings, where the “kinds” of strings depend on the program. For instance, a library that manipulates XML may reserve single-quoted strings for XML fragments, because those fragments often contain double quotes.

Strings in Lua can contain the following C-like escape sequences:

`\a`	bell
`\b`	back space
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\\`	backslash
`\"`	double quote
`\'`	single quote

The following examples illustrate 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 literal string also by its numeric value through the escape sequences `\`*`ddd`* and `\x`*`hh`*, where *`ddd`* is a sequence of up to three decimal digits and *`hh`* is a sequence of exactly two hexadecimal digits. As a somewhat artificial example, the two literals `"ALO\n123\""` and `'\x41LO \10\04923"'` have the same value in a system using ASCII: 0x41 (65 in decimal) is the ASCII code for `A`, 10 is the code for newline, and 49 is the code for the digit `1`. (In this example we must write 49 with three digits, as `\049`, because it is followed by another digit; otherwise Lua would read the escape as `\492`.) We could also write that same string as `'\x41\x4c\x4f\x0a\x31\x32\x33\x22'`, representing each character by its hexadecimal code.

Since Lua 5.3, we can also specify UTF-8 characters with the escape sequence `\u{`*`h... h`*`}`; we can write any number of hexadecimal digits inside the brackets:

```
> "\u{3b1} \u{3b2} \u{3b3}"          --> # # #
```

(The above example assumes an UTF-8 terminal.)

Long strings

We can delimit literal strings also by matching double square brackets, as we do with long comments. Literals in this bracketed form can run for several lines and do not interpret escape sequences. Moreover, it

ignores the first character of the string when this character is a newline. This form is especially convenient for writing strings that contain large pieces of code, as in the following example:

```
page = [[
<html>
<head>
  <title>An HTML Page</title>
</head>
<body>
  <a href="http://www.lua.org">Lua</a>
</body>
</html>
]]

write(page)
```

Sometimes, we may need to enclose a piece of code containing something like `a = b[c[i]]` (notice the `]]` in this code), or we may need to enclose some code that already has some code commented out. To handle such cases, we can add any number of equals signs between the two opening brackets, as in `[===[`. After this change, the literal string ends only at the next closing brackets with the same number of equals signs in between (`]===]`, in our example). The scanner ignores any pairs of brackets with a different number of equals signs. By choosing an appropriate number of signs, we can enclose any literal string without having to modify it in any way.

This same facility is valid for comments, too. For instance, if we start a long comment with `--[=[`, it extends until the next `]=]`. This facility allows us to comment out easily a piece of code that contains parts already commented out.

Long strings are the ideal format to include literal text in our code, but we should not use them for non-text literals. Although literal strings in Lua can contain arbitrary bytes, it is not a good idea to use this feature (e.g., you may have problems with your text editor); moreover, end-of-line sequences like `"\r\n"` may be normalized to `"\n"` when read. Instead, it is better to code arbitrary binary data using numeric escape sequences either in decimal or in hexadecimal, such as `"\x13\x01\xA1\xBB"`. However, this poses a problem for long strings, because they would result in quite long lines. For those situations, since version 5.2 Lua offers the escape sequence `\z`: it skips all subsequent space characters in the string until the first non-space character. The next example illustrates its use:

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

The `\z` at the end of the first line skips the following end-of-line and the indentation of the second line, so that the byte `\x08` directly follows `\x07` in the resulting string.

Coercions

Lua provides automatic conversions between numbers and strings at run time. Any numeric operation applied to a string tries to convert the string to a number. Lua applies such coercions not only in arithmetic operators, but also in other places that expect a number, such as the argument to `math.sin`.

Conversely, whenever Lua finds a number where it expects a string, it converts the number to a string:

```
print(10 .. 20)           --> 1020
```

(When we write the concatenation operator right after a numeral, we must separate them with a space; otherwise, Lua thinks that the first dot is a decimal point.)

Many people argue that these automatic coercions were not a good idea in the design of Lua. As a rule, it is better not to count on them. They are handy in a few places, but add complexity both to the language and to programs that use them.

As a reflection of this “second-class status”, Lua 5.3 did not implement a full integration of coercions and integers, favoring instead a simpler and faster implementation. The rule for arithmetic operations is that the result is an integer only when both operands are integers; a string is not an integer, so any arithmetic operation with strings is handled as a floating-point operation:

```
> "10" + 1             --> 11.0
```

To convert a string to a number explicitly, we can use the function `tonumber`, which returns nil if the string does not denote a proper number. Otherwise, it returns integers or floats, following the same rules of the Lua scanner:

```
> tonumber("  -3 ")        --> -3
> tonumber(" 10e4 ")       --> 100000.0
> tonumber("10e")          --> nil   (not a valid number)
> tonumber("0x1.3p-4")     --> 0.07421875
```

By default, `tonumber` assumes decimal notation, but we can specify any base between 2 and 36 for the conversion:

```
> tonumber("100101", 2)       --> 37
> tonumber("fff", 16)         --> 4095
> tonumber("-ZZ", 36)         --> -1295
> tonumber("987", 8)          --> nil
```

In the last line, the string does not represent a proper numeral in the given base, so `tonumber` returns nil.

To convert a number to a string, we can call the function `tostring`:

```
print(tostring(10) == "10")   --> true
```

These conversions are always valid. Remember, however, that we have no control over the format (e.g., the number of decimal digits in the resulting string). For full control, we should use `string.format`, which we will see in the next section.

Unlike arithmetic operators, order operators never coerce their arguments. Remember that `"0"` is different from `0`. Moreover, `2 < 15` is obviously true, but `"2" < "15"` is false (alphabetical order). To avoid inconsistent results, Lua raises an error when we mix strings and numbers in an order comparison, such as `2 < "15"`.

The String Library

The power of a raw Lua interpreter to manipulate strings is quite limited. A program can create string literals, concatenate them, compare them, and get string lengths. However, it cannot extract substrings or examine their contents. The full power to manipulate strings in Lua comes from its string library.

As I mentioned before, the string library assumes one-byte characters. This equivalence is true for several encodings (e.g., ASCII or ISO-8859-1), but it breaks in any Unicode encoding. Nevertheless, as we will see, several parts of the string library are quite useful for UTF-8.

Some functions in the string library are quite simple: the call `string.len(s)` returns the length of a string `s`; it is equivalent to `#s`. The call `string.rep(s, n)` returns the string `s` repeated `n`

times; we can create a string of 1 MB (e.g., for tests) with `string.rep("a", 2^20)`. The function `string.reverse` reverses a string. The call `string.lower(s)` returns a copy of `s` with the upper-case letters converted to lower case; all other characters in the string are unchanged. The function `string.upper` converts to upper case.

```
> string.rep("abc", 3)             --> abcabcabc
> string.reverse("A Long Line!")   --> !eniL gnoL A
> string.lower("A Long Line!")     --> a long line!
> string.upper("A Long Line!")     --> A LONG LINE!
```

As a typical use, if we want to compare two strings regardless of case, we can write something like this:

```
string.lower(a) < string.lower(b)
```

The call `string.sub(s, i, j)` extracts a piece of the string `s`, from the `i`-th to the `j`-th character inclusive. (The first character of a string has index 1.) We can also use negative indices, which count from the end of the string: index -1 refers to the last character, -2 to the previous one, and so on. Therefore, the call `string.sub(s, 1, j)` gets a prefix of the string `s` with length `j`; `string.sub(s, j, -1)` gets a suffix of the string, starting at the `j`-th character; and `string.sub(s, 2, -2)` returns a copy of the string `s` with the first and last characters removed:

```
> s = "[in brackets]"
> string.sub(s, 2, -2)      --> in brackets
> string.sub(s, 1, 1)       --> [
> string.sub(s, -1, -1)     --> ]
```

Remember that strings in Lua are immutable. Like any other function in Lua, `string.sub` does not change the value of a string, but returns a new string. A common mistake is to write something like `string.sub(s, 2, -2)` and assume that it will modify the value of `s`. If we want to modify the value of a variable, we must assign the new value to it:

```
s = string.sub(s, 2, -2)
```

The functions `string.char` and `string.byte` convert between characters and their internal numeric representations. The function `string.char` gets zero or more integers, converts each one to a character, and returns a string concatenating all these characters. The call `string.byte(s, i)` returns the internal numeric representation of the `i`-th character of the string `s`; the second argument is optional; the call `string.byte(s)` returns the internal numeric representation of the first (or single) character of `s`. The following examples assume the ASCII encoding for characters:

```
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
```

In the last line, we used a negative index to access the last character of the string.

A call like `string.byte(s, i, j)` returns multiple values with the numeric representation of all characters between indices `i` and `j` (inclusive):

```
print(string.byte("abc", 1, 2))           --> 97 98
```

A nice idiom is `{string.byte(s, 1, -1)}`, which creates a list with the codes of all characters in `s`. (This idiom only works for strings somewhat shorter than 1 MB. Lua limits its stack size, which in turn limits the maximum number of returns from a function. The default stack limit is one million entries.)

The function `string.format` is a powerful tool for formatting strings and converting numbers to strings. It returns a copy of its first argument, the so-called *format string*, with each *directive* in that string replaced by a formatted version of its correspondent argument. The directives in the format string have rules similar to those of the C function `printf`. A directive is a percent sign plus a letter that tells how to format the argument: `d` for a decimal integer, `x` for hexadecimal, `f` for a floating-point number, `s` for strings, plus several others.

```
> string.format("x = %d  y = %d", 10, 20)   --> x = 10  y = 20
> string.format("x = %x", 200)              --> x = c8
> string.format("x = 0x%X", 200)            --> x = 0xC8
> string.format("x = %f", 200)              --> x = 200.000000
> tag, title = "h1", "a title"
> string.format("<%s>%s</%s>", tag, title, tag)
  --> <h1>a title</h1>
```

Between the percent sign and the letter, a directive can include other options that control the details of the formatting, such as the number of decimal digits of 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
```

In the first example, the `%.4f` means a floating-point number with four digits after the decimal point. In the second example, the `%02d` means a decimal number with zero padding and at least two digits; the directive `%2d`, without the zero, would use blanks for padding. For a complete description of these directives, see the documentation of the C function `printf`, as Lua calls the standard C library to do the hard work here.

We can call all functions from the string library as methods on strings, using the colon operator. For instance, we can rewrite the call `string.sub(s, i, j)` as `s:sub(i, j)`; `string.upper(s)` becomes `s:upper()`. (We will discuss the colon operator in detail in Chapter 21, *Object-Oriented Programming*.)

The string library includes also several functions based on pattern matching. The function `string.find` searches for a pattern in a given string:

```
> string.find("hello world", "wor")   --> 7   9
> string.find("hello world", "war")   --> nil
```

It returns the initial and final positions of the pattern in the string, or nil if it cannot find the pattern. The function `string.gsub` (Global SUBstitution) replaces all occurrences of a pattern in a string with another string:

```
> string.gsub("hello world", "l", ".")     --> he..o wor.d    3
> string.gsub("hello world", "ll", "..")   --> he..o world    1
> string.gsub("hello world", "a", ".")     --> hello world    0
```

It also returns, as a second result, the number of replacements it made.

We will discuss more about these functions and all about pattern matching in Chapter 10, *Pattern Matching*.

Unicode

Since version 5.3, Lua includes a small library to support operations on Unicode strings encoded in UTF-8. Even before that library, Lua already offered a reasonable support for UTF-8 strings.

UTF-8 is the dominant encoding for Unicode on the Web. Because of its compatibility with ASCII, UTF-8 is also the ideal encoding for Lua. That compatibility is enough to ensure that several string-manipulation techniques that work on ASCII strings also work on UTF-8 with no modifications.

UTF-8 represents each Unicode character using a variable number of bytes. For instance, it represents `A` with one byte, 65; it represents the Hebrew character Aleph, which has code 1488 in Unicode, with the two-byte sequence 215–144. UTF-8 represents all characters in the ASCII range as in ASCII, that is, with a single byte smaller than 128. It represents all other characters using sequences of bytes where the first byte is in the range *[194,244]* and the continuation bytes are in the range *[128,191]*. More specifically, the range of the starting bytes for two-byte sequences is *[194,223]*; for three-byte sequences, the range is *[224,239]*; and for four-byte sequences, it is *[240,244]*. None of those ranges overlap. This property ensures that the code sequence of any character never appears as part of the code sequence of any other character. In particular, a byte smaller than 128 never appears in a multibyte sequence; it always represents its corresponding ASCII character.

Several things in Lua "just work" for UTF-8 strings. Because Lua is 8-bit clean, it can read, write, and store UTF-8 strings just like other strings. Literal strings can contain UTF-8 data. (Of course, you probably will want to edit your source code as a UTF-8 file in a UTF-8–aware editor.) The concatenation operation works correctly for UTF-8 strings. String order operators (less than, less equal, etc.) compare UTF-8 strings following the order of their character codes in Unicode.

Lua's operating-system library and I/O library are mainly interfaces to the underlying system, so their support for UTF-8 strings depends on that underlying system. On Linux, for instance, we can use UTF-8 for file names, but Windows uses UTF-16. Therefore, to manipulate Unicode file names on Windows, we need either extra libraries or changes to the standard Lua libraries.

Let us now see how functions from the string library handle UTF-8 strings. The functions `reverse`, `upper`, `lower`, `byte`, and `char` do not work for UTF-8 strings, as all of them assume that one character is equivalent to one byte. The functions `string.format` and `string.rep` work without problems with UTF-8 strings except for the format option `'%c'`, which assumes that one character is one byte. The functions `string.len` and `string.sub` work correctly with UTF-8 strings, with indices referring to byte counts (not character counts). More often than not, this is what we need.

Let us now have a look at the new `utf8` library. The function `utf8.len` returns the number of UTF-8 characters (codepoints) in a given string. Moreover, it validates the string: if it finds any invalid byte sequence, it returns false plus the position of the first invalid byte:

```
> utf8.len("résumé")              --> 6
> utf8.len("ação")                --> 4
> utf8.len("Månen")               --> 5
> utf8.len("ab\x93")              --> nil    3
```

(Of course, to run these examples we need a terminal that understands UTF-8.)

The functions `utf8.char` and `utf8.codepoint` are the equivalent of `string.char` and `string.byte` in the UTF-8 world:

```
> utf8.char(114, 233, 115, 117, 109, 233)    --> résumé
> utf8.codepoint("résumé", 6, 7)             --> 109    233
```

Note the indices in the last line. Most functions in the `utf8` library work with indices in bytes. For instance, the call `string.codepoint(s, i, j)` considers both `i` and `j` to be byte positions in string `s`. If we want to use character indices, the function `utf8.offset` converts a character position to a byte position:

```
> s = "Nähdään"
> utf8.codepoint(s, utf8.offset(s, 5))    --> 228
```

```
> utf8.char(228)                                --> ä
```

In this example, we used `utf8.offset` to get the byte index of the fifth character in the string, and then provided that index to `codepoint`.

As in the string library, the character index for `utf8.offset` can be negative, in which case the counting is from the end of the string:

```
> s = "ÃøÆËÐ"
> string.sub(s, utf8.offset(s, -2))    --> ËÐ
```

The last function in the `utf8` library is `utf8.codes`. It allows us to iterate over the characters in a UTF-8 string:

```
for i, c in utf8.codes("Ação") do
  print(i, c)
end
  --> 1    65
  --> 2    231
  --> 4    227
  --> 6    111
```

This construction traverses all characters in the given string, assigning its position in bytes and its numeric code to two local variables. In our example, the loop body only prints the values of those variables. (We will discuss iterators in more detail in Chapter 18, *Iterators and the Generic* ***for***.)

Unfortunately, there is not much more that Lua can offer. Unicode has too many peculiarities. It is virtually impossible to abstract almost any concept from specific languages. Even the concept of what is a character is vague, because there is no one-to-one correspondence between Unicode coded characters and graphemes. For instance, the common grapheme é can be represented by a single codepoint (`"\u{E9}"`) or by two codepoints, an `e` followed by a diacritical mark (`"e\u{301}"`). Other apparently basic concepts, such as what is a letter, also change across different languages. Because of this complexity, complete support for Unicode demands huge tables, which are incompatible with the small size of Lua. So, for anything fancier, the best approach is an external library.

Exercises

Exercise 4.1: How can you embed the following fragment of XML as a string in a Lua program?

```
<![CDATA[
  Hello world
]]>
```

Show at least two different ways.

Exercise 4.2: Suppose you need to write a long sequence of arbitrary bytes as a literal string in Lua. What format would you use? Consider issues like readability, maximum line length, and size.

Exercise 4.3: Write a function to insert a string into a given position of another one:

```
> insert("hello world", 1, "start: ")    --> start: hello world
> insert("hello world", 7, "small ")     --> hello small world
```

Exercise 4.4: Redo the previous exercise for UTF-8 strings:

```
> insert("ação", 5, "!")     --> ação!
```

(Note that the position now is counted in codepoints.)

Exercise 4.5: Write a function to remove a slice from a string; the slice should be given by its initial position and its length:

```
> remove("hello world", 7, 4)     --> hello d
```

Exercise 4.6: Redo the previous exercise for UTF-8 strings:

```
> remove("ação", 2, 2)     --> ao
```

(Here, both the initial position and the length should be counted in codepoints.)

Exercise 4.7: Write a function to check whether a given string is a palindrome:

```
> ispali("step on no pets")     --> true
> ispali("banana")              --> false
```

Exercise 4.8: Redo the previous exercise so that it ignores differences in spaces and punctuation.

Exercise 4.9: Redo the previous exercise for UTF-8 strings.

Chapter 5. Tables

Tables are the main (in fact, the only) data structuring mechanism in Lua, and a powerful one. We use tables to represent arrays, sets, records, and many other data structures in a simple, uniform, and efficient way. Lua uses tables to represent packages and objects as well. When we write `math.sin`, we think about "the function `sin` from the `math` library". For Lua, this expression means "index the table `math` using the string `"sin"` as the key".

A table in Lua is essentially an associative array. A table is an array that accepts not only numbers as indices, but also strings or any other value of the language (except nil).

Tables in Lua are neither values nor variables; they are *objects*. If you are familiar with arrays in Java or Scheme, then you have a fair idea of what I mean. You may think of a table as a dynamically-allocated object; programs manipulate only references (or pointers) to them. Lua never does hidden copies or creation of new tables behind the scenes.

We create tables by means of a *constructor expression*, which in its simplest form is written as `{}`:

```
> a = {}           -- create a table and assign its reference
> k = "x"
> a[k] = 10        -- new entry, with key="x" and value=10
> a[20] = "great"  -- new entry, with key=20 and value="great"
> a["x"]                --> 10
> k = 20
> a[k]                  --> "great"
> a["x"] = a["x"] + 1   -- increments entry "x"
> a["x"]                --> 11
```

A table is always anonymous. There is no fixed relationship between a variable that holds a table and the table itself:

```
> a = {}
> a["x"] = 10
> b = a            -- 'b' refers to the same table as 'a'
> b["x"]           --> 10
> b["x"] = 20
> a["x"]           --> 20
> a = nil          -- only 'b' still refers to the table
> b = nil          -- no references left to the table
```

When a program has no more references to a table, the garbage collector will eventually delete the table and reuse its memory.

Table Indices

Each table can store values with different types of indices, and it grows as needed to accommodate new entries:

```
> a = {}     -- empty table
> -- create 1000 new entries
> for i = 1, 1000 do a[i] = i*2 end
> a[9]              --> 18
> a["x"] = 10
> a["x"]            --> 10
```

```
> a["y"]              --> nil
```

Note the last line: like global variables, table fields evaluate to nil when not initialized. Also like global variables, we can assign nil to a table field to delete it. This is not a coincidence: Lua stores global variables in ordinary tables. (We will discuss this subject further in Chapter 22, *The Environment*.)

To represent structures, we use the field name as an index. Lua supports this representation by providing `a.name` as syntactic sugar for `a["name"]`. Therefore, we could write the last lines of the previous example in a cleaner manner as follows:

```
> a = {}                        -- empty table
> a.x = 10                      -- same as a["x"] = 10
> a.x       --> 10              -- same as a["x"]
> a.y       --> nil             -- same as a["y"]
```

For Lua, the two forms are equivalent and can be intermixed freely. For a human reader, however, each form may signal a different intention. The dot notation clearly shows that we are using the table as a structure, where we have some set of fixed, predefined keys. The string notation gives the idea that the table can have any string as a key, and that for some reason we are manipulating that specific key.

A common mistake for beginners is to confuse `a.x` with `a[x]`. The first form represents `a["x"]`, that is, a table indexed by the string `"x"`. The second form is a table indexed by the value of the variable `x`. See the difference:

```
> a = {}
> x = "y"
> a[x] = 10                 -- put 10 in field "y"
> a[x]          --> 10      -- value of field "y"
> a.x           --> nil     -- value of field "x" (undefined)
> a.y           --> 10      -- value of field "y"
```

Because we can index a table with any type, when indexing a table we have the same subtleties that arise in equality. Although we can index a table both with the number `0` and with the string `"0"`, these two values are different and therefore denote different entries in a table. Similarly, the strings `"+1"`, `"01"`, and `"1"` all denote different entries. When in doubt about the actual types of your indices, use an explicit conversion to be sure:

```
> i = 10; j = "10"; k = "+10"
> a = {}
> a[i] = "number key"
> a[j] = "string key"
> a[k] = "another string key"
> a[i]              --> number key
> a[j]              --> string key
> a[k]              --> another string key
> a[tonumber(j)]    --> number key
> a[tonumber(k)]    --> number key
```

You can introduce subtle bugs in your program if you do not pay attention to this point.

Integers and floats do not have the above problem. In the same way that `2` compares equal to `2.0`, both values refer to the same table entry, when used as keys:

```
> a = {}
> a[2.0] = 10
> a[2.1] = 20
```

```
> a[2]              --> 10
> a[2.1]            --> 20
```

More specifically, when used as a key, any float value that can be converted to an integer is converted. For instance, when Lua executes `a[2.0] = 10`, it converts the key `2.0` to `2`. Float values that cannot be converted to integers remain unaltered.

Table Constructors

Constructors are expressions that create and initialize tables. They are a distinctive feature of Lua and one of its most useful and versatile mechanisms.

The simplest constructor is the empty constructor, `{}`, as we have seen. Constructors also initialize lists. For instance, the following statement will initialize `days[1]` with the string `"Sunday"` (the first element of the constructor has index 1, not 0), `days[2]` with `"Monday"`, and so on:

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

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

Lua also offers a special syntax to initialize a record-like table, as in the next example:

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

This previous line is equivalent to these commands:

```
a = {}; a.x = 10; a.y = 20
```

The original expression, however, is faster, because Lua creates the table already with the right size.

No matter what constructor we use to create a table, we can always add fields to and remove fields from the result:

```
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'
x.f = w                  -- add key "f" to table 'x'
print(w["x"])            --> 0
print(w[1])              --> another field
print(x.f[1])            --> another field
w.x = nil                -- remove field "x"
```

However, as I just mentioned, creating a table with a proper constructor is more efficient, besides being cleaner.

We can mix record-style and list-style initializations 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 above example also illustrates how we can nest tables (and constructors) to represent more complex data structures. Each of the elements `polyline[i]` is a table representing a record:

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

Those two constructor forms have their limitations. For instance, we cannot initialize fields with negative indices, nor with string indices that are not proper identifiers. For such needs, there is another, more general, format. In this format, we explicitly write each 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 cumbersome, but more flexible too: both the list-style and the record-style forms are special cases of this more general syntax, as we show in the following equivalences:

```
{x = 0, y = 0}    <-->    {["x"] = 0, ["y"] = 0}
{"r", "g", "b"}   <-->    {[1] = "r", [2] = "g", [3] = "b"}
```

We can always put a comma after the last entry. These trailing commas are optional, but are always valid:

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

This flexibility frees programs that generate Lua constructors from the need to handle the last element as a special case.

Finally, we can always use a semicolon instead of a comma in a constructor. This facility is a leftover from older Lua versions and I guess it is seldom used nowadays.

Arrays, Lists, and Sequences

To represent a conventional array or a list, we simply use a table with integer keys. There is neither a way nor a need to declare a size; we just initialize the elements we need:

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

Given that we can index a table with any value, we can start the indices of an array with any number that pleases us. However, it is customary in Lua to start arrays with one (and not with zero, as in C) and many facilities in Lua stick to this convention.

Usually, when we manipulate a list we must know its length. It can be a constant or it can be stored somewhere. Often we store the length of a list in a non-numeric field of the table; for historical reasons, several programs use the field `"n"` for this purpose. Often, however, the length is implicit. Remember that any non-initialized index results in nil; we can use this value as a sentinel to mark the end of the list. For instance, after we read 10 lines into a list, it is easy to know that its length is 10, because its numeric keys

are *1, 2, ..., 10*. This technique only works when the list does not have *holes*, which are nil elements inside it. We call such a list without holes a *sequence*.

For sequences, Lua offers the length operator (#). As we have seen, on strings it gives the number of bytes in the string. On tables, it gives the length of the *sequence* represented by the table. For instance, we could print the lines read in the last example with the following code:

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

The length operator also provides a useful idiom for manipulating sequences:

```
a[#a + 1] = v        -- appends 'v' to the end of the sequence
```

The length operator is unreliable for lists with holes (nils). It only works for sequences, which we defined as lists without holes. More precisely, a *sequence* is a table where the positive numeric keys comprise a set *{1,...,n}* for some *n*. (Remember that any key with value nil is actually not in the table.) In particular, a table with no numeric keys is a sequence with length zero.

The behavior of the length operator for lists with holes is one of the most contentious features of Lua. Over the years, there have been many proposals either to raise an error when we apply the length operator to a list with holes, or to extend its meaning to those lists. However, these proposals are easier said than done. The problem is that, because a list is actually a table, the concept of “length” is somewhat fuzzy. For instance, consider the list resulting from the following code:

```
a = {}
a[1] = 1
a[2] = nil    -- does nothing, as 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 that is has a hole at index 2. However, what can we say about the next similar example?

```
a = {}
a[1] = 1
a[10000] = 1
```

Should we consider `a` as a list with 10000 elements, with 9998 holes? Now, the program does this:

```
a[10000] = nil
```

What is the list length now? Should it be 9999, because the program deleted the last element? Or maybe still 10000, as the program only changed the last element to nil? Or should the length collapse to one?

Another common proposal is to make the # operator return the total number of elements in the table. This semantics is clear and well defined, but not very useful or intuitive. Consider all the examples we are discussing here and think how useful would be such operator for them.

Yet more troubling are nils at the end of the list. What should be the length of the following list?

```
a = {10, 20, 30, nil, nil}
```

Remember that, for Lua, a field with nil is indistinct from an absent field. Therefore, the previous table is equal to `{10, 20, 30}`; its length is 3, not 5.

You may consider that a nil at the end of a list is a very special case. However, many lists are built by adding elements one by one. Any list with holes that was built that way must have had nils at its end along the way.

Despite all these discussions, most lists we use in our programs are sequences (e.g., a file line cannot be nil) and, therefore, most of the time the use of the length operator is safe. If you really need to handle lists with holes, you should store the length explicitly somewhere.

Table Traversal

We can traverse all key–value pairs in a table with the `pairs` iterator:

```
t = {10, print, x = 12, k = "hi"}
for k, v in pairs(t) do
  print(k, v)
end
  --> 1    10
  --> k    hi
  --> 2    function: 0x420610
  --> x    12
```

Due to the way that Lua implements tables, the order that elements appear in a traversal is undefined. The same program can produce different orders each time it runs. The only certainty is that each element will appear once during the traversal.

For lists, we can use the `ipairs` iterator:

```
t = {10, print, 12, "hi"}
for k, v in ipairs(t) do
  print(k, v)
end
  --> 1    10
  --> 2    function: 0x420610
  --> 3    12
  --> 4    hi
```

In this case, Lua trivially ensures the order.

Another way to traverse a sequence is with a numerical for:

```
t = {10, print, 12, "hi"}
for k = 1, #t do
  print(k, t[k])
end
  --> 1    10
  --> 2    function: 0x420610
  --> 3    12
  --> 4    hi
```

Safe Navigation

Suppose the following situation: we want to know whether a given function from a given library is present. If we know for sure that the library itself exists, we can write something like `if lib.foo then ...`. Otherwise, we have to write something like `if lib and lib.foo then ...`.

When the level of nested tables gets deeper, this notation becomes problematic, as the next example illustrates:

```
zip = company and company.director and
        company.director.address and
          company.director.address.zipcode
```

This notation is not only cumbersome, but inefficient, too. It performs six table accesses in a successful access, instead of three.

Some programming languages, such as C#, offer a *safe navigation operator* (written as `?.` in C#) for this task. When we write `a ?. b` and `a` is nil, the result is also nil, instead of an error. Using that operator, we could write our previous example like this:

```
zip = company?.director?.address?.zipcode
```

If any component in the path were nil, the safe operator would propagate that nil until the final result.

Lua does not offer a safe navigation operator, and we do not think it should. Lua is minimalistic. Moreover, this operator is quite controversial, with many people arguing —not without some reason— that it promotes careless programming. However, we can emulate it in Lua with a bit of extra notation.

If we execute `a or {}` when `a` is nil, the result is the empty table. So, if we execute `(a or {}).b` when `a` is nil, the result will be also nil. Using this idea, we can rewrite our original expression like this:

```
zip = (((company or {}).director or {}).address or {}).zipcode
```

Still better, we can make it a little shorter and slightly more efficient:

```
E = {}     -- can be reused in other similar expressions
...
zip = (((company or E).director or E).address or E).zipcode
```

Granted, this syntax is more complex than the one with the safe navigation operator. Nevertheless, we write each field name only once, it performs the minimum required number of table accesses (three, in this example), and it requires no new operators in the language. In my personal opinion, it is a good enough substitute.

The Table Library

The table library offers several useful functions to operate over lists and sequences.[1]

The function `table.insert` inserts an element in a given position of a sequence, moving up other elements to open space. For instance, if `t` is the list `{10, 20, 30}`, after the call `table.insert(t, 1, 15)` it will become `{15, 10, 20, 30}`. As a special and frequent case, if we call `insert` without a position, it inserts the element in the last position of the sequence, moving no elements. As an example, the following code reads the input stream line by line, storing all lines in a sequence:

```
t = {}
for line in io.lines() do
  table.insert(t, line)
end
print(#t)         --> (number of lines read)
```

[1] You can think of it as “The Sequence Library” or “The List Library”; we have kept the original name for compatibility with old versions.

The function `table.remove` removes and returns an element from the given position in a sequence, moving subsequent elements down to fill the gap. When called without a position, it removes the last element of the sequence.

With these two functions, it is straightforward to implement stacks, queues, and double queues. We can initialize such structures as `t = {}`. A push operation is equivalent to `table.insert(t, x)`; a pop operation is equivalent to `table.remove(t)`. The call `table.insert(t, 1, x)` inserts at the other end of the structure (its beginning, actually), and `table.remove(t, 1)` removes from this end. The last two operations are not particularly efficient, as they must move elements up and down. However, because the `table` library implements these functions in C, these loops are not too expensive, so that this implementation is good enough for small arrays (up to a few hundred elements, say).

Lua 5.3 has introduced a more general function for moving elements in a table. The call `table.move(a, f, e, t)` moves the elements in table `a` from index `f` until `e` (both inclusive) to position `t`. For instance, to insert an element in the beginning of a list `a`, we can do the following:

```
table.move(a, 1, #a, 2)
a[1] = newElement
```

The next code removes the first element:

```
table.move(a, 2, #a, 1)
a[#a] = nil
```

Note that, as is common in computing, a *move* actually *copies* values from one place to another. In this last example, we must explicitly erase the last element after the move.

We can call `table.move` with an extra optional parameter, a table. In that case, the function moves the elements from the first table into the second one. For instance, the call `table.move(a, 1, #a, 1, {})` returns a clone of list `a` (by copying all its elements into a new list), while `table.move(a, 1, #a, #b + 1, b)` appends all elements from list `a` to the end of list `b`.

Exercises

Exercise 5.1: What will the following script print? Explain.

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

Exercise 5.2: Assume the following code:

```
a = {};  a.a = a
```

What would be the value of `a.a.a.a`? Is any `a` in that sequence somehow different from the others?

Now, add the next line to the previous code:

```
a.a.a.a = 3
```

What would be the value of `a.a.a.a` now?

Exercise 5.3: Suppose that you want to create a table that maps each escape sequence for strings (the section called “Literal strings”) to its meaning. How could you write a constructor for that table?

Exercise 5.4: We can represent a polynomial $a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x^1 + a_0$ in Lua as a list of its coefficients, such as $\{a_0, a_1, ..., a_n\}$.

Write a function that takes a polynomial (represented as a table) and a value for `x` and returns the polynomial value.

Exercise 5.5: Can you write the function from the previous item so that it uses at most *n* additions and *n* multiplications (and no exponentiations)?

Exercise 5.6: Write a function to test whether a given table is a valid sequence.

Exercise 5.7: Write a function that inserts all elements of a given list into a given position of another given list.

Exercise 5.8: The table library offers a function `table.concat`, which receives a list of strings and returns their concatenation:

```
print(table.concat({"hello", " ", "world"}))    --> hello world
```

Write your own version for this function.

Compare the performance of your implementation against the built-in version for large lists, with hundreds of thousands of entries. (You can use a **for** loop to create those large lists.)

Chapter 6. Functions

Functions are the main mechanism for abstraction of statements and expressions in Lua. Functions can both carry out a specific task (what is sometimes called a *procedure* or a *subroutine* in other languages) or compute and return values. In the first case, we use a function call as a statement; in the second case, we use it as an expression:

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

In both cases, a list of arguments enclosed in parentheses denotes the call; if the call has no arguments, we still must write an empty list `()` to denote it. There is a special case to this rule: if the function has one single argument and that argument is either a literal string or a table constructor, then 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 a special syntax for object-oriented calls, the colon operator. An expression like `o:foo(x)` calls the method `foo` in the object `o`. In Chapter 21, *Object-Oriented Programming*, we will discuss such calls and object-oriented programming in more detail.

A Lua program can use functions defined both in Lua and in C (or in any other language used by the host application). Typically, we resort to C functions both to achieve better performance and to access facilities not easily accessible directly from Lua, such as operating-system facilities. As an example, all functions from the standard Lua libraries are written in C. However, when calling a function, there is no difference between functions defined in Lua and functions defined in C.

As we saw in other examples, a function definition in Lua has a conventional syntax, like here:

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

In this syntax, a function definition has a *name* (`add`, in the example), a list of *parameters*, and a *body*, which is a list of statements. Parameters work exactly as local variables initialized with the values of the arguments passed in the function call.

We can call a function with a number of arguments different from its number of parameters. Lua adjusts the number of arguments to the number of parameters by throwing away extra arguments and supplying nils to extra parameters. For instance, consider the next function:

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

It has the following behavior:

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

Although this behavior can lead to programming errors (easily spotted with minimal tests), it is also useful, especially for default arguments. As an example, consider the following function, to increment a global counter:

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

This function has 1 as its default argument; the call `incCount()`, without arguments, increments `globalCounter` by one. When we call `incCount()`, Lua first initializes the parameter `n` with nil; the **or** expression results in its second operand and, as a result, Lua assigns a default 1 to `n`.

Multiple Results

An unconventional but quite convenient feature of Lua is that functions can return multiple results. Several predefined functions in Lua return multiple values. We have already seen the function `string.find`, which locates a pattern in a string. This function returns two indices when it finds the pattern: the index of the character where the match starts and the one where it ends. A multiple assignment allows the program to get both results:

```
s, e = string.find("hello Lua users", "Lua")
print(s, e)   --> 7      9
```

(Remember that the first character of a string has index 1.)

Functions that we write in Lua also can return multiple results, by listing them all after the **return** keyword. For instance, a function to find the maximum element in a sequence can return both the maximum value and its location:

```
function maximum (a)
  local mi = 1          -- index of the maximum value
  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          -- return the maximum and its index
end

print(maximum({8,10,23,12,5}))     --> 23   3
```

Lua always adjusts the number of results from a function to the circumstances of the call. When we call a function as a statement, Lua discards all results from the function. When we use a call as an expression (e.g., the operand of an addition), Lua keeps only the first result. We get all results only when the call is the last (or the only) expression in a list of expressions. These lists appear in four constructions in Lua: multiple assignments, arguments to function calls, table constructors, and **return** statements. To illustrate all these cases, we will assume the following definitions for the next examples:

```
function foo0 () end                    -- returns no results
function foo1 () return "a" end         -- returns 1 result
function foo2 () return "a", "b" end    -- returns 2 results
```

In a multiple assignment, a function call as the last (or only) expression produces as many results as needed to match the 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"
```

In a multiple assignment, if a function has fewer results than we need, Lua produces nils for the missing values:

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

Remember that multiple results only happen when the call is the last (or only) expression in a list. A function call that is not the last element in the list always produces exactly one result:

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

When a function call is the last (or the only) argument to another call, all results from the first call go as arguments. We saw examples of this construction already, with `print`. Because `print` can receive a variable number of arguments, the statement `print(g())` prints all results returned by `g`.

```
print(foo0())          -->            (no results)
print(foo1())          --> a
print(foo2())          --> a   b
print(foo2(), 1)       --> a   1
print(foo2() .. "x")   --> ax         (see next)
```

When the call to `foo2` appears inside an expression, Lua adjusts the number of results to one; so, in the last line, the concatenation uses only the first result, `"a"`.

If we write `f(g())`, and `f` has a fixed number of parameters, Lua adjusts the number of results from `g` to the number of parameters of `f`. Not by chance, this is exactly the same behavior that happens in a multiple assignment.

A constructor also collects all results from a call, without any adjustments:

```
t = {foo0()}         -- t = {}  (an empty table)
t = {foo1()}         -- t = {"a"}
t = {foo2()}         -- t = {"a", "b"}
```

As always, this behavior happens only when the call is the last expression in the list; calls in any other position produce exactly one result:

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

Finally, a statement like `return f()` returns all values returned by `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))     --> a  b
print(foo(0))     -- (no results)
print(foo(3))     -- (no results)
```

We can force a call to return exactly one result by enclosing it in an extra pair of parentheses:

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

Beware that a **return** statement does not need parentheses around the returned value; any pair of parentheses placed there counts as an extra pair. Therefore, a statement like `return (f(x))` always returns one single value, no matter how many values `f` returns. Sometimes this is what we want, sometimes not.

Variadic Functions

A function in Lua can be *variadic*, that is, it can take a variable number of arguments. For instance, we have already called `print` with one, two, and more arguments. Although `print` is defined in C, we can define variadic functions in Lua, too.

As a simple example, the following function returns the summation of all its arguments:

```
function add (...)
  local s = 0
  for _, v in ipairs{...} do
    s = s + v
  end
  return s
end

print(add(3, 4, 10, 25, 12))    --> 54
```

The three dots (`...`) in the parameter list indicate that the function is variadic. When we call this function, Lua collects all its arguments internally; we call these collected arguments the *extra arguments* of the function. A function accesses its extra arguments using again the three dots, now as an expression. In our example, the expression `{...}` results in a list with all collected arguments. The function then traverses the list to add its elements.

We call the three-dot expression a *vararg expression*. It behaves like a multiple return function, returning all extra arguments of the current function. For instance, the command `print(...)` prints all extra arguments of the function. Likewise, the next command creates two local variables with the values of the first two optional arguments (or nil if there are no such arguments):

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

Actually, we can emulate the usual parameter-passing mechanism of Lua translating

```
function foo (a, b, c)
```

to

```
function foo (...)
  local a, b, c = ...
```

Those who fancy Perl's parameter-passing mechanism may enjoy this second form.

A function like the next one simply returns all its arguments:

```
function id (...) return ... end
```

It is a multi-value identity function. The next function behaves exactly like another function `foo`, except that before the call it prints a message with its arguments:

```
function foo1 (...)
  print("calling foo:", ...)
  return foo(...)
end
```

This is a useful trick for tracing calls to a specific function.

Let us see another useful example. Lua provides separate functions for formatting text (`string.format`) and for writing text (`io.write`). It is straightforward to combine both functions into a single variadic function:

```
function fwrite (fmt, ...)
  return io.write(string.format(fmt, ...))
end
```

Note the presence of a fixed parameter `fmt` before the dots. Variadic functions can have any number of fixed parameters before the variadic part. Lua assigns the first arguments to these parameters; the rest (if any) goes as extra arguments.

To iterate over its extra arguments, a function can use the expression `{...}` to collect them all in a table, as we did in our definition of `add`. However, in the rare occasions when the extra arguments can be valid nils, the table created with `{...}` may not be a proper sequence. For instance, there is no way to detect in such a table whether there were trailing nils in the original arguments. For these occasions, Lua offers the function `table.pack`.[1] This function receives any number of arguments and returns a new table with all its arguments (just like `{...}`), but this table has also an extra field `"n"`, with the total number of arguments. As an example, the following function uses `table.pack` to test whether none of its arguments is nil:

```
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
```

Another option to traverse the variable arguments of a function is the `select` function. A call to `select` has always one fixed argument, the *selector*, plus a variable number of extra arguments. If the selector is

[1] This function was introduced in Lua 5.2.

a number *n*, `select` returns all arguments after the *n*-th argument; otherwise, the selector should be the string `"#"`, so that `select` returns the total number of extra arguments.

```
print(select(1, "a", "b", "c"))       --> a    b    c
print(select(2, "a", "b", "c"))       --> b    c
print(select(3, "a", "b", "c"))       --> c
print(select("#", "a", "b", "c"))     --> 3
```

More often than not, we use `select` in places where its number of results is adjusted to one, so we can think about `select(n, ...)` as returning its `n`-th extra argument.

As a typical example of the use of `select`, here is our previous `add` function using it:

```
function add (...)
  local s = 0
  for i = 1, select("#", ...) do
    s = s + select(i, ...)
  end
  return s
end
```

For few arguments, this second version of `add` is faster, because it avoids the creation of a new table at each call. For more arguments, however, the cost of multiple calls to `select` with many arguments outperforms the cost of creating a table, so the first version becomes a better choice. (In particular, the second version has a quadratic cost, because both the number of iterations and the number of arguments passed in each iteration grow with the number of arguments.)

The function `table.unpack`

A special function with multiple returns is `table.unpack`. It takes a list and returns as results all elements from the list:

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

As the name implies, `table.unpack` is the reverse of `table.pack`. While `pack` transforms a parameter list into a real Lua list (a table), `unpack` transforms a real Lua list (a table) into a return list, which can be given as the parameter list to another function.

An important use for `unpack` is in a generic call mechanism. A generic call mechanism allows us to call any function, with any arguments, dynamically. In ISO C, for instance, there is no way to code a generic call. We can declare a function that takes a variable number of arguments (with `stdarg.h`) and we can call a variable function, using pointers to functions. However, we cannot call a function with a variable number of arguments: each call you write in C has a fixed number of arguments, and each argument has a fixed type. In Lua, if we want to call a variable function `f` with variable arguments in an array `a`, we simply write this:

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

The call to `unpack` returns all values in `a`, which become the arguments to `f`. For instance, consider the following call:

```
print(string.find("hello", "ll"))
```

We can dynamically build an equivalent call with the following code:

```
f = string.find
a = {"hello", "ll"}

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

Usually, `table.unpack` uses the length operator to know how many elements to return, so it works only on proper sequences. If needed, however, we can provide explicit limits:

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

Although the predefined function `unpack` is written in C, we could write it also 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, the parameter `i` gets 1 and `n` gets the length of the sequence. Then the function returns `t[1]` followed by all results from `unpack(t, 2, n)`, which in turn returns `t[2]` followed by all results from `unpack(t, 3, n)`, and so on, stopping after `n` elements.

Proper Tail Calls

Another interesting feature of functions in Lua is that Lua does tail-call elimination. (This means that Lua is *properly tail recursive*, although the concept does not involve recursion directly; see Exercise 6.6.)

A *tail call* is a goto dressed as a call. A tail call happens when a function calls another as its last action, so it has nothing else to do. For instance, in the following code, the call to `g` is a tail call:

```
function f (x) x = x + 1; return g(x) end
```

After `f` calls `g`, it has nothing else to do. In such situations, the program does not need to return to the calling function when the called function ends. Therefore, after the tail call, the program does not need to keep any information about the calling function on the stack. When `g` returns, control can return directly to the point that called `f`. Some language implementations, such as the Lua interpreter, take advantage of this fact and actually do not use any extra stack space when doing a tail call. We say that these implementations do *tail-call elimination*.

Because tail calls use no stack space, the number of nested tail calls that a program can make is unlimited. For instance, we can call the following function passing any number as argument:

```
function foo (n)
  if n > 0 then return foo(n - 1) end
end
```

It will never overflow the stack.

A subtle point about tail-call elimination is what is a tail call. Some apparently obvious candidates fail the criterion that the calling function has nothing else to do after the call. For instance, in the following code, the call to `g` is not a tail call:

```
function f (x)  g(x)  end
```

The problem in this example is that, after calling `g`, `f` still has to discard any results from `g` before returning. Similarly, all the following calls fail the criterion:

```
return g(x) + 1     -- must do the addition
return x or g(x)    -- must adjust to 1 result
return (g(x))       -- must adjust to 1 result
```

In Lua, only a call with the form `return` *`func`*`(`*`args`*`)` is a tail call. However, both *`func`* and its arguments can be complex expressions, because Lua evaluates them before the call. For instance, the next call is a tail call:

```
return x[i].foo(x[j] + a*b, i + j)
```

Exercises

Exercise 6.1: Write a function that takes an array and prints all its elements.

Exercise 6.2: Write a function that takes an arbitrary number of values and returns all of them, except the first one.

Exercise 6.3: Write a function that takes an arbitrary number of values and returns all of them, except the last one.

Exercise 6.4: Write a function to shuffle a given list. Make sure that all permutations are equally probable.

Exercise 6.5: Write a function that takes an array and prints all combinations of the elements in the array. (Hint: you can use the recursive formula for combination: *C(n,m) = C(n -1, m -1) + C(n - 1, m)*. To generate all *C(n,m)* combinations of *n* elements in groups of size *m*, you first add the first element to the result and then generate all *C(n - 1, m - 1)* combinations of the remaining elements in the remaining slots; then you remove the first element from the result and then generate all *C(n - 1, m)* combinations of the remaining elements in the free slots. When *n* is smaller than *m*, there are no combinations. When *m* is zero, there is only one combination, which uses no elements.)

Exercise 6.6: Sometimes, a language with proper-tail calls is called *properly tail recursive*, with the argument that this property is relevant only when we have recursive calls. (Without recursive calls, the maximum call depth of a program would be statically fixed.)

Show that this argument does not hold in a dynamic language like Lua: write a program that performs an unbounded call chain without recursion. (Hint: see the section called “Compilation”.)

Chapter 7. The External World

Because of its emphasis on portability and embeddability, Lua itself does not offer much in terms of facilities to communicate with the external world. Most I/O in real Lua programs is done either by the host application or through external libraries not included in the main distribution, from graphics to databases and network access. Pure Lua offers only the functionalities that the ISO C standard offers —namely, basic file manipulation plus some extras. In this chapter, we will see how the standard libraries cover these functionalities.

The Simple I/O Model

The I/O library offers two different models for file manipulation. The simple model assumes a *current input stream* and a *current output stream*, and its I/O operations operate on these streams. The library initializes the current input stream to the process's standard input (`stdin`) and the current output stream to the process's standard output (`stdout`). Therefore, when we execute something like `io.read()`, we read a line from the standard input.

We can change these current streams with the functions `io.input` and `io.output`. A call like `io.input(filename)` opens a stream over the given file in read mode and sets it as the current input stream. From this point on, all input will come from this file, until another call to `io.input`. The function `io.output` does a similar job for output. In case of error, both functions raise the error. If you want to handle errors directly, you should use the complete I/O model.

As `write` is simpler than `read`, we will look at it first. The function `io.write` simply takes an arbitrary number of strings (or numbers) and writes them to the current output stream. Because we can call it with multiple arguments, we should avoid calls like `io.write(a..b..c)`; the call `io.write(a, b, c)` accomplishes the same effect with fewer resources, as it avoids the concatenations.

As a rule, you should use `print` only for quick-and-dirty programs or debugging; always use `io.write` when you need full control over your output. Unlike `print`, `write` adds no extra characters to the output, such as tabs or newlines. Moreover, `io.write` allows you to redirect your output, whereas `print` always uses the standard output. Finally, `print` automatically applies `tostring` to its arguments; this is handy for debugging, but it also can hide subtle bugs.

The function `io.write` converts numbers to strings following the usual conversion rules; for full control over this conversion, we should use `string.format`:

```
> 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
```

The function `io.read` reads strings from the current input stream. Its arguments control what to read:[1]

`"a"`	reads the whole file
`"l"`	reads the next line (dropping the newline)
`"L"`	reads the next line (keeping the newline)
`"n"`	reads a number
num	reads *num* characters as a string

[1] In Lua 5.2 and before, all string options should be preceded by an asterisk. Lua 5.3 still accepts the asterisk for compatibility.

The call `io.read("a")` reads the whole current input file, starting at its current position. If we are at the end of the file, or if the file is empty, the call returns an empty string.

Because Lua handles long strings efficiently, a simple technique for writing filters in Lua is to read the whole file into a string, process the string, and then write the string to the output:

```
t = io.read("a")                       -- read the whole file
t = string.gsub(t, "bad", "good")      -- do the job
io.write(t)                            -- write the file
```

As a more concrete example, the following chunk is a complete program to code a file's content using the MIME *quoted-printable* encoding. This encoding codes each non-ASCII byte as *=xx*, where *xx* is the value of the byte in hexadecimal. To keep the consistency of the encoding, it must encode the equals sign as well:

```
t = io.read("all")
t = string.gsub(t, "([\128-\255=])", function (c)
      return string.format("=%02X", string.byte(c))
    end)
io.write(t)
```

The function `string.gsub` will match all non-ASCII bytes (codes from 128 to 255), plus the equals sign, and call the given function to provide a replacement. (We will discuss pattern matching in detail in Chapter 10, *Pattern Matching*.)

The call `io.read("l")` returns the next line from the current input stream, without the newline character; the call `io.read("L")` is similar, but it keeps the newline (if present in the file). When we reach the end of file, the call returns nil, as there is no next line to return. Option `"l"` is the default for `read`. Usually, I use this option only when the algorithm naturally handles the data line by line; otherwise, I favor reading the whole file at once, with option `"a"`, or in blocks, as we will see later.

As a simple example of the use of line-oriented input, the following program copies its current input to the current output, numbering each line:

```
for count = 1, math.huge do
  local line = io.read("L")
  if line == nil then break end
  io.write(string.format("%6d  ", count), line)
end
```

However, to iterate on a whole file line by line, the `io.lines` iterator allows a simpler code:

```
local count = 0
for line in io.lines() do
  count = count + 1
  io.write(string.format("%6d  ", count), line, "\n")
end
```

As another example of line-oriented input, Figure 7.1, “A program to sort a file” shows a complete program to sort the lines of a file.

Figure 7.1. A program to sort a file

```
local lines = {}

-- read the lines in table 'lines'
for line in io.lines() do
  lines[#lines + 1] = line
end

-- sort
table.sort(lines)

-- write all the lines
for _, l in ipairs(lines) do
  io.write(l, "\n")
end
```

The call `io.read("n")` reads a number from the current input stream. This is the only case where `read` returns a number (integer or float, following the same rules of the Lua scanner) instead of a string. If, after skipping spaces, `io.read` cannot find a numeral at the current file position (because of bad format or end of file), it returns nil.

Besides the basic read patterns, we can call `read` with a number *n* as an argument: in this case, it tries to read *n* characters from the input stream. If it cannot read any character (end of file), the call returns nil; otherwise, it returns a string with at most *n* characters from the stream. As an example of this read pattern, the following program is an efficient way to copy a file from `stdin` to `stdout`:

```
while true do
  local block = io.read(2^13)          -- block size is 8K
  if not block then break end
  io.write(block)
end
```

As a special case, `io.read(0)` works as a test for end of file: it returns an empty string if there is more to be read or nil otherwise.

We can call `read` with multiple options; for each argument, the function will return the respective result. Suppose we have a file with three numbers per line:

```
6.0       -3.23     15e12
4.3       234       1000001
...
```

Now we want to print the maximum value of each line. We can read all three numbers of each line with a single call to `read`:

```
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
```

The Complete I/O Model

The simple I/O model is convenient for simple things, but it is not enough for more advanced file manipulation, such as reading from or writing to several files simultaneously. For these manipulations, we need the complete model.

To open a file, we use the function `io.open`, which mimics the C function `fopen`. It takes as arguments the name of the file to open plus a *mode* string. This mode string can contain an `r` for reading, a `w` for writing (which also erases any previous content of the file), or an `a` for appending, plus an optional `b` to open binary files. The function `open` returns a new stream over the file. In case of error, `open` returns nil, plus an error message and a system-dependent error number:

```
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
```

A typical idiom to check for errors is to use the function `assert`:

```
local f = assert(io.open(filename, mode))
```

If the `open` fails, the error message goes as the second argument to `assert`, which then shows the message.

After we open a file, we can read from or write to the resulting stream with the methods `read` and `write`. They are similar to the functions `read` and `write`, but we call them as methods on the stream object, using the colon operator. For instance, to open a file and read it all, we can use a fragment like this:

```
local f = assert(io.open(filename, "r"))
local t = f:read("a")
f:close()
```

(We will discuss the colon operator in detail in Chapter 21, *Object-Oriented Programming*.)

The I/O library offers handles for the three predefined C streams, called `io.stdin`, `io.stdout`, and `io.stderr`. For instance, we can send a message directly to the error stream with a code like this:

```
io.stderr:write(message)
```

The functions `io.input` and `io.output` allow us to mix the complete model with the simple model. We get the current input stream by calling `io.input()`, without arguments. We set this stream with the call `io.input(handle)`. (Similar calls are also valid for `io.output`.) For instance, if we want to change the current input stream temporarily, we can write something like this:

```
local temp = io.input()     -- save current stream
io.input("newinput")        -- open a new current stream
do something with new input
io.input():close()          -- close current stream
io.input(temp)              -- restore previous current stream
```

Note that `io.read(`*`args`*`)` is actually a shorthand for `io.input():read(`*`args`*`)`, that is, the `read` method applied over the current input stream. Similarly, `io.write(`*`args`*`)` is a shorthand for `io.output():write(`*`args`*`)`.

Instead of `io.read`, we can also use `io.lines` to read from a stream. As we saw in previous examples, `io.lines` gives an iterator that repeatedly reads from a stream. Given a file name, `io.lines` will open a stream over the file in read mode and will close it after reaching end of file. When called with no arguments, `io.lines` will read from the current input stream. We can also use `lines` as a method over handles. Moreover, since Lua 5.2 `io.lines` accepts the same options that `io.read` accepts. As an example, the next fragment copies the current input to the current output, iterating over blocks of 8 KB:

```
for block in io.input():lines(2^13) do
  io.write(block)
end
```

Other Operations on Files

The function `io.tmpfile` returns a stream over a temporary file, open in read/write mode. This file is automatically removed (deleted) when the program ends.

The function `flush` executes all pending writes to a file. Like the function `write`, we can call it as a function —`io.flush()`— to flush the current output stream, or as a method —`f:flush()`— to flush the stream `f`.

The `setvbuf` method sets the buffering mode of a stream. Its first argument is a string: `"no"` means no buffering; `"full"` means that the stream data is only written out when the buffer is full or when we explicitly flush the file; and `"line"` means that the output is buffered until a newline is output or there is any input from some special files (such as a terminal device). For the last two options, `setvbuf` accepts an optional second argument with the buffer size.

In most systems, the standard error stream (`io.stderr`) is not buffered, while the standard output stream (`io.stdout`) is buffered in line mode. So, if we write incomplete lines to the standard output (e.g., a progress indicator), we may need to flush the stream to see that output.

The `seek` method can both get and set the current position of a stream in a file. Its general form is `f:seek(whence, offset)`, where the `whence` parameter is a string that specifies how to interpret the offset. Its valid values are `"set"`, for offsets relative to the beginning of the file; `"cur"`, for offsets relative to the current position in the file; and `"end"`, for offsets relative to the end of the file. Independently of the value of `whence`, the call returns the new current position of the stream, measured in bytes from the beginning of the file.

The default value for `whence` is `"cur"` and for `offset` is zero. Therefore, the call `file:seek()` returns the current stream position, without changing it; the call `file:seek("set")` resets the position to the beginning of the file (and returns zero); and the call `file:seek("end")` sets the position to the end of the file and returns its size. The following function gets the file size without changing its current position:

```
function fsize (file)
  local current = file:seek()      -- save current position
  local size = file:seek("end")    -- get file size
  file:seek("set", current)        -- restore position
  return size
end
```

To complete the set, `os.rename` changes the name of a file and `os.remove` removes (deletes) a file. Note that these functions come from the `os` library, not the `io` library, because they manipulate real files, not streams.

All these functions return nil plus an error message and an error code in case of errors.

Other System Calls

The function `os.exit` terminates the execution of a program. Its optional first argument is the return status of the program. It can be a number (zero means a successful execution) or a Boolean (**true** means a successful execution). An optional second argument, if true, closes the Lua state, calling all finalizers and releasing all memory used by that state. (Usually this finalization is not necessary, because most operating systems release all resources used by a process when it exits.)

The function `os.getenv` gets the value of an environment variable. It takes the name of the variable and returns a string with its value:

```
print(os.getenv("HOME"))    --> /home/lua
```

The call returns nil for undefined variables.

Running system commands

The function `os.execute` runs a system command; it is equivalent to the C function `system`. It takes a string with the command and returns information regarding how the command terminated. The first result is a Boolean: **true** means the program exited with no errors. The second result is a string: `"exit"` if the program terminated normally or `"signal"` if it was interrupted by a signal. A third result is the return status (if the program terminated normally) or the number of the signal that terminated the program. As an example, both in POSIX and Windows we can use the following function to create new directories:

```
function createDir (dirname)
  os.execute("mkdir " .. dirname)
end
```

Another quite useful function is `io.popen`.[2] Like `os.execute`, it runs a system command, but it also connects the command output (or input) to a new local stream and returns that stream, so that our script can read data from (or write to) the command. For instance, the following script builds a table with the entries in the current directory:

```
-- for POSIX systems, use 'ls' instead of 'dir'
local f = io.popen("dir /B", "r")
local dir = {}
for entry in f:lines() do
  dir[#dir + 1] = entry
end
```

The second parameter (`"r"`) to `io.popen` means that we intend to read from the command. The default is to read, so this parameter is optional in the example.

The next example sends an email message:

```
local subject = "some news"
local address = "someone@somewhere.org"

local cmd = string.format("mail -s '%s' '%s'", subject, address)
local f = io.popen(cmd, "w")
f:write([[
Nothing important to say.
```

[2]This function is not available in all Lua installations, because the corresponding functionality is not part of ISO C. Despite not being standard in C, we included it in the standard libraries due to its generality and presence in major operating systems.

```
-- me
]])
f:close()
```

(This script only works on POSIX systems, with the appropriate packages installed.) The second parameter to `io.popen` now is `"w"`, meaning that we intend to write to the command.

As we can see from those two previous examples, both `os.execute` and `io.popen` are powerful functions, but they are also highly system dependent.

For extended OS access, your best option is to use an external Lua library, such as `LuaFileSystem`, for basic manipulation of directories and file attributes, or `luaposix`, which offers much of the functionality of the POSIX.1 standard.

Exercises

Exercise 7.1: Write a program that reads a text file and rewrites it with its lines sorted in alphabetical order. When called with no arguments, it should read from standard input and write to standard output. When called with one file-name argument, it should read from that file and write to standard output. When called with two file-name arguments, it should read from the first file and write to the second.

Exercise 7.2: Change the previous program so that it asks for confirmation if the user gives the name of an existing file for its output.

Exercise 7.3: Compare the performance of Lua programs that copy the standard input stream to the standard output stream in the following ways:

- byte by byte;
- line by line;
- in chunks of 8 kB;
- the whole file at once.

For the last option, how large can the input file be?

Exercise 7.4: Write a program that prints the last line of a text file. Try to avoid reading the entire file when the file is large and seekable.

Exercise 7.5: Generalize the previous program so that it prints the last *n* lines of a text file. Again, try to avoid reading the entire file when the file is large and seekable.

Exercise 7.6: Using `os.execute` and `io.popen`, write functions to create a directory, to remove a directory, and to collect the entries in a directory.

Exercise 7.7: Can you use `os.execute` to change the current directory of your Lua script? Why?

Chapter 8. Filling some Gaps

We have already used most of Lua's syntactical constructions in previous examples, but it is easy to miss some details. For completeness, this chapter closes the first part of the book with more details about them.

Local Variables and Blocks

By default, variables in Lua are global. All local variables must be declared as such. Unlike global variables, a local variable has its scope limited to the block where it is declared. A *block* is the body of a control structure, the body of a function, or a chunk (the file or string where the variable is declared):

```
x = 10
local i = 1        -- local to the chunk

while i <= x do
  local x = i * 2  -- local to the while body
  print(x)         --> 2, 4, 6, 8, ...
  i = i + 1
end

if i > 20 then
  local x          -- local to the "then" body
  x = 20
  print(x + 2)     -- (would print 22 if test succeeded)
else
  print(x)         --> 10  (the global one)
end

print(x)           --> 10  (the global one)
```

Beware that this last example will not work as expected if you enter it in interactive mode. In interactive mode, each line is a chunk by itself (unless it is not a complete command). As soon as you enter the second line of the example (`local i = 1`), Lua runs it and starts a new chunk in the next line. By then, the **local** declaration is already out of scope. To solve this problem, we can delimit the whole block explicitly, bracketing it with the keywords **do**–**end**. Once you enter the **do**, the command completes only at the corresponding **end**, so Lua will not execute each line by itself.

These **do** blocks are useful also when we need finer control over the scope of some local variables:

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

It is good programming style to use local variables whenever possible. Local variables avoid cluttering the global environment with unnecessary names; they also avoid name clashes between different parts of a program. Moreover, the access to local variables is faster than to global ones. Finally, a local variable vanishes as soon as its scope ends, allowing the garbage collector to release its value.

Given that local variables are “better” than global ones, some people argue that Lua should use local by default. However, local by default has its own set of problems (e.g., issues with accessing non-local variables). A better approach would be no default, that is, all variables should be declared before used. The Lua distribution comes with a module `strict.lua` for global-variable checks; it raises an error if we try to assign to a non-existent global inside a function or to use a non-existent global. It is a good habit to use it when developing Lua code.

Each local declaration can include an initial assignment, which works the same way as a conventional multiple assignment: extra values are thrown away, extra variables get nil. If a declaration has no initial assignment, it initializes all its variables with nil:

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

A common idiom in Lua is

```
local foo = foo
```

This code creates a local variable, `foo`, and initializes it with the value of the global variable `foo`. (The local `foo` becomes visible only *after* its declaration.) This idiom is useful to speed up the access to `foo`. It is also useful when the chunk needs to preserve the original value of `foo` even if later some other function changes the value of the global `foo`; in particular, it makes the code resistant to monkey patching. Any piece of code preceded by `local print = print` will use the original function `print` even if `print` is monkey patched to something else.

Some people think it is a bad practice to use declarations in the middle of a block. Quite the opposite: by declaring a variable only when we need it, we seldom need to declare it without an initial value (and therefore we seldom forget to initialize it). Moreover, we shorten the scope of the variable, which increases readability.

Control Structures

Lua provides a small and conventional set of control structures, with **if** for conditional execution and **while**, **repeat**, and **for** for iteration. All control structures have a syntax with an explicit terminator: **end** terminates **if**, **for** and **while** structures; **until** terminates **repeat** structures.

The condition expression of a control structure can result in any value. Remember that Lua treats as true all values different from **false** and nil. (In particular, Lua treats both zero and the empty string as true.)

if then else

An **if** statement tests its condition and executes its *then-part* or its *else-part* accordingly. 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
```

To write nested **if**s we can use **elseif**. It is similar to an **else** followed by an **if**, but it avoids the need for multiple **end**s:

```
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
```

Because Lua has no switch statement, such chains are somewhat common.

while

As the name implies, a **while** loop repeats its body while a condition is true. As usual, Lua first tests the **while** condition; if the condition is false, then the loop ends; otherwise, Lua executes the body of the loop and repeats the process.

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

repeat

As the name implies, a **repeat**–**until** statement repeats its body until its condition is true. This statement does the test after the body, so that it always executes the body at least once.

```
-- print the first non-empty input line
local line
repeat
  line = io.read()
until line ~= ""
print(line)
```

Differently from most other languages, in Lua the scope of a local variable declared inside the loop includes the condition:

```
-- computes the square root of 'x' using Newton-Raphson method
local sqr = x / 2
repeat
  sqr = (sqr + x/sqr) / 2
  local error = math.abs(sqr^2 - x)
until error < x/10000      -- local 'error' still visible here
```

Numerical for

The **for** statement has two variants: the *numerical* **for** and the *generic* **for**.

A numerical **for** has the following syntax:

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

This loop will execute `something` for each value of `var` from `exp1` to `exp2`, using `exp3` as the *step* to increment `var`. This third expression is optional; when absent, Lua assumes one as the step value. If we want a loop without an upper limit, we can use the 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 should learn in order to make good use of it. First, all three expressions are evaluated once, before the loop starts. Second, the control variable is a local variable automatically declared by the **for** statement, and it is visible only inside the loop. A typical mistake is to assume that the variable still exists after the loop ends:

```
for i = 1, 10 do print(i) end
max = i      -- probably wrong!
```

If you need the value of the control variable after the loop (usually when you break the loop), you must save its value into another variable:

```
-- find a value in a 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 not change the value of the control variable: the effect of such changes is unpredictable. If you want to end a **for** loop before its normal termination, use **break** (as we did in the previous example).

Generic for

The generic **for** loop traverses all values returned by an iterator function. We saw some examples already, with `pairs`, `ipairs`, `io.lines`, etc. Despite its apparent simplicity, the generic **for** is powerful. With proper iterators, we can traverse almost anything in a readable fashion.

Of course, we can write our own iterators. Although the use of the generic **for** is easy, the task of writing iterator functions has its subtleties; hence, we will cover this topic later, in Chapter 18, *Iterators and the Generic **for***.

Unlike the numerical **for**, the generic **for** can have multiple variables, which are all updated at each iteration. The loop stops when the first variable gets nil. As in the numerical loop, the loop variables are local to the loop body and you should not change their values inside each iteration.

break, return, and goto

The **break** and **return** statements allow us to jump out of a block. The **goto** statement allows us to jump to almost any point in a function.

We use the **break** statement to finish a loop. This statement breaks the inner loop (**for**, **repeat**, or **while**) that contains it; it cannot be used outside a loop. After the break, the program continues running from the point immediately after the broken loop.

A **return** statement returns the results from a function or simply finishes the function. There is an implicit return at the end of any function, so we do not need to write one for functions that end naturally, without returning any value.

For syntactic reasons, a **return** can appear only as the last statement of a block: in other words, as the last statement in our chunk or just before an **end**, an **else**, or an **until**. For instance, in the next example, **return** is the last statement of the **then** block:

```
local i = 1
while a[i] do
  if a[i] == v then return i end
  i = i + 1
end
```

Usually, these are the places where we use a **return**, because any statement following it would be unreachable. Sometimes, however, it may be useful to write a **return** in the middle of a block; for instance, we may be debugging a function and want to avoid its execution. In such cases, we can use an explicit **do** block around the statement:

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

A **goto** statement jumps the execution of a program to a corresponding label. There has been a long going debate about goto, with some people arguing even today that they are harmful to programming and should be banned from programming languages. Nonetheless, several current languages offer goto, with good reason. They are a powerful mechanism and, when used with care, can only improve the quality of our code.

In Lua, the syntax for a goto statement is quite conventional: it is the reserved word **goto** followed by the label name, which can be any valid identifier. The syntax for a label is a little more convoluted: it has two colons followed by the label name followed by more two colons, like in `::name::`. This convolution is intentional, to highlight labels in a program.

Lua poses some restrictions to where we can jump with a goto. First, labels follow the usual visibility rules, so we cannot jump into a block (because a label inside a block is not visible outside it). Second, we cannot jump out of a function. (Note that the first rule already excludes the possibility of jumping *into* a function.) Third, we cannot jump into the scope of a local variable.

A typical and well-behaved use of a goto is to simulate some construction that you learned from another language but that is absent from Lua, such as continue, multi-level break, multi-level continue, redo, local error handling, etc. A continue statement is simply a goto to a label at the end of a loop block; a 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 detail in the specification of Lua is that the scope of a local variable ends on the last *non-void* statement of the block where the variable is defined; labels are considered void statements. To see the usefulness of this detail, consider the next fragment:

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

You may think that this goto jumps into the scope of the variable `var`. However, the `continue` label appears after the last non-void statement of the block, and therefore it is not inside the scope of `var`.

The goto is also useful for writing state machines. As an example, Figure 8.1, “An example of a state machine with **goto**” shows a program that checks whether its input has an even number of zeros.

Figure 8.1. An example of a state machine with 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
```

There are better ways to write this specific program, but this technique is useful if we want to translate a finite automaton into Lua code automatically (think about dynamic code generation).

As another example, let us consider a simple maze game. The maze has several rooms, each with up to four doors: north, south, east, and west. At each step, the user enters a movement direction. If there is a

door in this direction, the user goes to the corresponding room; otherwise, the program prints a warning. The goal is to go from an initial room to a final room.

This game is a typical state machine, where the current room is the state. We can implement this maze with one block for each room, using a goto to move from one room to another. Figure 8.2, “A maze game” shows how we could write a small maze with four rooms.

Figure 8.2. A maze game

```
goto room1      -- initial room

::room1:: do
  local move = io.read()
  if move == "south" then goto room3
  elseif move == "east" then goto room2
  else
    print("invalid 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("invalid 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("invalid move")
    goto room3
  end
end

::room4:: do
  print("Congratulations, you won!")
end
```

For this simple game, you may find that a data-driven program, where you describe the rooms and movements with tables, is a better design. However, if the game has several special situations in each room, then this state-machine design is quite appropriate.

Exercises

Exercise 8.1: Most languages with a C-like syntax do not offer an **elseif** construct. Why does Lua need this construct more than those languages?

Exercise 8.2: Describe four different ways to write an unconditional loop in Lua. Which one do you prefer?

Exercise 8.3: Many people argue that **repeat**--**until** is seldom used, and therefore it should not be present in a minimalistic language like Lua. What do you think?

Exercise 8.4: As we saw in the section called “Proper Tail Calls”, a tail call is a goto in disguise. Using this idea, reimplement the simple maze game from the section called “**break**, **return**, and **goto**” using tail calls. Each block should become a new function, and each goto becomes a tail call.

Exercise 8.5: Can you explain why Lua has the restriction that a goto cannot jump out of a function? (Hint: how would you implement that feature?)

Exercise 8.6: Assuming that a goto could jump out of a function, explain what the program in Figure 8.3, “A strange (and invalid) use of a **goto**” would do.

Figure 8.3. A strange (and invalid) use of a goto

```
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()
```

(Try to reason about the label using the same scoping rules used for local variables.)

Part II. Real Programming

Table of Contents

Chapter 9. Closures

Functions in Lua are first-class values with proper lexical scoping.

What does it mean for functions to be “first-class values”? It means that, in Lua, a function is a value with the same rights as more conventional values like numbers and strings. A program can store functions in variables (both global and local) and in tables, pass functions as arguments to other functions, and return functions as results.

What does it mean for functions to have “lexical scoping”? It means that functions can access variables of their enclosing functions. (It also means that Lua properly contains the lambda calculus.)

Together, these two features give great flexibility to the language; for instance, a program can redefine a function to add new functionality or erase a function to create a secure environment when running a piece of untrusted code (such as code received through a network). More importantly, these features allow us to apply in Lua many powerful programming techniques from the functional-language world. Even if you have no interest at all in functional programming, it is worth learning a little about how to explore these techniques, because they can make your programs smaller and simpler.

Functions as First-Class Values

As we just saw, functions in Lua are first-class values. The following example illustrates what that means:

```
a = {p = print}      -- 'a.p' refers to the 'print' function
a.p("Hello World")   --> Hello World
print = math.sin     -- 'print' now refers to the sine function
a.p(print(1))        --> 0.8414709848079
math.sin = a.p       -- 'sin' now refers to the print function
math.sin(10, 20)     --> 10      20
```

If functions are values, are there expressions that create functions? Sure. In fact, the usual way to write a function in Lua, such as

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

is an instance of what we call *syntactic sugar*; it is simply a pretty way to write the following code:

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

The expression in the right-hand side of the assignment (`function (x)` *body* `end`) is a function constructor, in the same way that `{}` is a table constructor. Therefore, a function definition is in fact a statement that creates a value of type `"function"` and assigns it to a variable.

Note that, in Lua, all functions are anonymous. Like any other value, they do not have names. When we talk about a function name, such as `print`, we are actually talking about a variable that holds that function. Although we often assign functions to global variables, giving them something like a name, there are several occasions when functions remain anonymous. Let us see some examples.

The table library provides the function `table.sort`, which receives a table and sorts its elements. Such a function must allow unlimited variations in the sort order: ascending or descending, numeric or alphabetical, tables sorted by a key, and so on. Instead of trying to provide all kinds of options, `sort` provides a single optional parameter, which is the *order function*: a function that takes two elements and returns

whether the first must come before the second in the sorted list. For instance, suppose we have a table of records like this:

```
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 we want to sort the table by the field `name`, in reverse alphabetical order, we just write this:

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

See how handy the anonymous function is in this statement.

A function that takes another function as an argument, such as `sort`, is what we call a *higher-order function*. Higher-order functions are a powerful programming mechanism, and the use of anonymous functions to create their function arguments is a great source of flexibility. Nevertheless, remember that higher-order functions have no special rights; they are a direct consequence of the fact that Lua handles functions as first-class values.

To further illustrate the use of higher-order functions, we will write a naive implementation of a common higher-order function, the derivative. In an informal definition, the derivative of a function *f* is the function *f'(x) = (f(x + d) - f(x)) / d* when *d* becomes infinitesimally small. According to this definition, we can compute an approximation of the derivative as follows:

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

Given a function `f`, the call `derivative(f)` returns (an approximation 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
```

Non-Global Functions

An obvious consequence of first-class functions is that we can store functions not only in global variables, but also in table fields and in local variables.

We have already seen several examples of functions in table fields: most Lua libraries use this mechanism (e.g., `io.read`, `math.sin`). To create such functions in Lua, we only have to put together what we have learned so far:

```
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 offers a specific syntax to define such functions:

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

As we will see in Chapter 21, *Object-Oriented Programming*, the use of functions in table fields is a key ingredient for object-oriented programming in Lua.

When we store a function into a local variable, we get a *local function*, that is, a function that is restricted to a given scope. Such definitions are particularly useful for packages: because Lua handles each chunk as a function, a chunk can declare local functions, which are visible only inside the chunk. Lexical scoping ensures that other functions in the chunk can use these local functions.

Lua supports such uses of local functions with a syntactic sugar for them:

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

A subtle point arises in the definition of recursive local functions, because the naive approach does not work here. Consider the next definition:

```
local fact = function (n)
  if n == 0 then return 1
  else return n*fact(n-1)   -- buggy
  end
end
```

When Lua compiles the call `fact(n - 1)` in the function body, the local `fact` is not yet defined. Therefore, this expression will try to call a global `fact`, not the local one. We can solve this problem by first defining the local variable and then the function:

```
local fact
fact = function (n)
  if n == 0 then return 1
  else return n*fact(n-1)
  end
end
```

Now the `fact` inside the function refers to the local variable. Its value when the function is defined does not matter; by the time the function executes, `fact` already has the right value.

When Lua expands its syntactic sugar for local functions, it does not use the naive definition. Instead, a definition like

```
local function foo (params)  body  end
```

expands to

```
local foo; foo = function (params)  body  end
```

Therefore, we can use this syntax for recursive functions without worrying.

Of course, this trick does not work if we have indirect recursive functions. In such cases, we must use the equivalent of an explicit forward declaration:

```
local f      -- "forward" declaration

local function g ()
  some code  f()  some code
end

function f ()
  some code  g()  some code
end
```

Beware not to write `local` in the last definition. Otherwise, Lua would create a fresh local variable `f`, leaving the original `f` (the one that `g` is bound to) undefined.

Lexical Scoping

When we write a function enclosed in another function, it has full access to local variables from the enclosing function; we call this feature *lexical scoping*. Although this visibility rule may sound obvious, it is not. Lexical scoping plus nested first-class functions give great power to a programming language, but many do not support the combination.

Let us start with a simple example. Suppose we have a list of student names and a table that maps names to grades; we want to sort the list of names according to their grades, with higher grades first. We can do this task as follows:

```
names = {"Peter", "Paul", "Mary"}
grades = {Mary = 10, Paul = 7, Peter = 8}
table.sort(names, function (n1, n2)
  return grades[n1] > grades[n2]        -- compare the grades
end)
```

Now, suppose we want to create a function to do this task:

```
function sortbygrade (names, grades)
  table.sort(names, function (n1, n2)
    return grades[n1] > grades[n2]      -- compare the grades
  end)
end
```

The interesting point in this last example is that the anonymous function given to `sort` accesses `grades`, which is a parameter to the enclosing function `sortbygrade`. Inside this anonymous function, `grades`

is neither a global variable nor a local variable, but what we call a *non-local variable*. (For historical reasons, non-local variables are also called *upvalues* in Lua.)

Why is this point so interesting? Because functions, being first-class values, can *escape* the original scope of their variables. Consider the following code:

```
function newCounter ()
  local count = 0
  return function ()      -- anonymous function
           count = count + 1
           return count
         end
end

c1 = newCounter()
print(c1())  --> 1
print(c1())  --> 2
```

In this code, the anonymous function refers to a non-local variable (`count`) to keep its counter. However, by the time we call the anonymous function, the variable `count` seems to be out of scope, because the function that created this variable (`newCounter`) has already returned. Nevertheless, Lua handles this situation correctly, using the concept of *closure*. Simply put, a closure is a function plus all it needs to access non-local variables correctly. If we call `newCounter` again, it will create a new local variable `count` and a new closure, acting over this new variable:

```
c2 = newCounter()
print(c2())  --> 1
print(c1())  --> 3
print(c2())  --> 2
```

So, `c1` and `c2` are different closures. Both are built over the same function, but each acts upon an independent instantiation of the local variable `count`.

Technically speaking, what is a value in Lua is the closure, not the function. The function itself is a kind of a prototype for closures. Nevertheless, we will continue to use the term “function” to refer to a closure whenever there is no possibility for confusion.

Closures provide a valuable tool in many contexts. As we have seen, they are useful as arguments to higher-order functions such as `sort`. Closures are valuable for functions that build other functions too, like our `newCounter` example or the derivative example; this mechanism allows Lua programs to incorporate sophisticated programming techniques from the functional world. Closures are useful for *callback* functions, too. A typical example here occurs when we create buttons in a conventional GUI toolkit. Each button has a callback function to be called when the user presses the button; we want different buttons to do slightly different things when pressed.

For instance, a digital calculator needs ten similar buttons, one for each digit. We can create each of them with a function like this:

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

```
end
```

In this example, we pretend that `Button` is a toolkit function that creates new buttons; `label` is the button label; and `action` is the callback function to be called when the button is pressed. The callback can be called a long time after `digitButton` did its task, but it can still access the `digit` variable.

Closures are valuable also in a quite different context. Because functions are stored in regular variables, we can easily redefine functions in Lua, even predefined functions. This facility is one of the reasons why Lua is so flexible. Frequently, when we redefine a function, we need the original function in the new implementation. As an example, suppose we want to redefine the function `sin` to operate in degrees instead of radians. This new function converts its argument and then calls the original function `sin` to do the real work. Our code could look like this:

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

A slightly cleaner way to do this redefinition is as follows:

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

This code uses a **do** block to limit the scope of the local variable `oldSin`; following conventional visibility rules, the variable is only visible inside the block. So, the only way to access it is through the new function.

We can use this same technique to create secure environments, also called sandboxes. Secure environments are essential when running untrusted code, such as code received through the Internet by a server. For instance, to restrict the files that a program can access, we can redefine `io.open` 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 nice is that, after this redefinition, there is no way for the program to call the unrestricted version of function `io.open` except through the new, restricted version. It keeps the insecure version as a private variable in a closure, inaccessible from the outside. With this technique, we can build Lua sandboxes in Lua itself, with the usual benefits: simplicity and flexibility. Instead of a one-size-fits-all solution, Lua offers us a meta-mechanism, so that we can tailor our environment for our specific security needs. (Real sandboxes do more than protecting external files. We will return to this subject in the section called “Sandboxing”.)

A Taste of Functional Programming

To give a more concrete example of functional programming, in this section we will develop a simple system for geometric regions.[1] The goal is to develop a system to represent geometric regions, where a region is a set of points. We want to be able to represent all kinds of shapes and to combine and modify shapes in several ways (rotation, translation, union, etc.).

To implement this system, we could start looking for good data structures to represent shapes; we could try an object-oriented approach and develop some hierarchy of shapes. Or we can work on a higher level of abstraction and represent our sets directly by their characteristic (or indicator) function. (The characteristic function of a set *A* is a function f_A such that $f_A(x)$ is true if and only if *x* belongs to *A*.) Given that a geometric region is a set of points, we can represent a region by its characteristic function; that is, we represent a region by a function that, given a point, returns true if and only if the point belongs to the region.

As an example, the next function represents a disk (a circular region) with center *(1.0, 3.0)* and radius *4.5*:

```
function disk1 (x, y)
  return (x - 1.0)^2 + (y - 3.0)^2 <= 4.5^2
end
```

With higher-order functions and lexical scoping, it is easy to define a disk factory, which creates disks with given centers and radius:

```
function disk (cx, cy, r)
  return function (x, y)
           return (x - cx)^2 + (y - cy)^2 <= r^2
         end
end
```

A call like `disk(1.0, 3.0, 4.5)` will create a disk equal to `disk1`.

The next function creates axis-aligned rectangles, given the bounds:

```
function rect (left, right, bottom, up)
  return function (x, y)
           return left <= x and x <= right and
                  bottom <= x and x <= up
         end
end
```

In a similar fashion, we can define functions to create other basic shapes, such as triangles and non–axis-aligned rectangles. Each shape has a completely independent implementation, needing only a correct characteristic function.

Now let us see how to modify and combine regions. To create the complement of any region is trivial:

```
function complement (r)
  return function (x, y)
           return not r(x, y)
         end
end
```

[1] This example is adapted from the research report *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity*, by Paul Hudak and Mark P. Jones.

Union, intersection, and difference are equally simple, as we show in Figure 9.1, “Union, intersection, and difference of regions”.

Figure 9.1. Union, intersection, and difference of regions

```
function union (r1, r2)
  return function (x, y)
           return r1(x, y) or r2(x, y)
         end
end

function intersection (r1, r2)
  return function (x, y)
           return r1(x, y) and r2(x, y)
         end
end

function difference (r1, r2)
  return function (x, y)
           return r1(x, y) and not r2(x, y)
         end
end
```

The following function translates a region by a given delta:

```
function translate (r, dx, dy)
  return function (x, y)
           return r(x - dx, y - dy)
         end
end
```

To visualize a region, we can traverse the viewport testing each pixel; pixels in the region are painted black, pixels outside it are painted white. To illustrate this process in a simple way, we will write a function to generate a PBM (*portable bitmap*) file with the drawing of a given region.

PBM files have a quite simple structure. (This structure is also highly inefficient, but our emphasis here is simplicity.) In its text-mode variant, it starts with a one-line header with the string `"P1"`; then there is one line with the width and height of the drawing, in pixels. Finally, there is a sequence of digits, one for each image pixel (1 for black, 0 for white), separated by optional spaces and end of lines. The function `plot` in Figure 9.2, “Drawing a region in a PBM file” creates a PBM file for a given region, mapping a virtual drawing area *(-1,1), [-1,1)* to the viewport area *[1,M], [1,N]*.

Figure 9.2. Drawing a region in a PBM file

```
function plot (r, M, N)
  io.write("P1\n", M, " ", N, "\n")     -- header
  for i = 1, N do                 -- for each line
    local y = (N - i*2)/N
    for j = 1, M do            -- for each column
      local x = (j*2 - M)/M
      io.write(r(x, y) and "1" or "0")
    end
    io.write("\n")
  end
end
```

To complete our example, the following command draws a waxing crescent moon (as seen from the Southern Hemisphere):

```
c1 = disk(0, 0, 1)
plot(difference(c1, translate(c1, 0.3, 0)), 500, 500)
```

Exercises

Exercise 9.1: Write a function `integral` that takes a function `f` and returns an approximation of its integral.

Exercise 9.2: What will be the output of the following chunk:

```
function F (x)
  return {
    set = function (y) x = y end,
    get = function () return x end
  }
end

o1 = F(10)
o2 = F(20)
print(o1.get(), o2.get())
o2.set(100)
o1.set(300)
print(o1.get(), o2.get())
```

Exercise 9.3: Exercise 5.4 asked you to write a function that receives a polynomial (represented as a table) and a value for its variable, and returns the polynomial value. Write the *curried* version of that function. Your function should receive a polynomial and return a function that, when called with a value for `x`, returns the value of the polynomial for that `x`. See the example:

```
f = newpoly({3, 0, 1})
print(f(0))    --> 3
print(f(5))    --> 28
print(f(10))   --> 103
```

Exercise 9.4: Using our system for geometric regions, draw a waxing crescent moon as seen from the Northern Hemisphere.

Exercise 9.5: In our system for geometric regions, add a function to rotate a given region by a given angle.

Chapter 10. Pattern Matching

Unlike several other scripting languages, Lua uses neither POSIX regex nor Perl regular expressions for pattern matching. The main reason for this decision is size: a typical implementation of POSIX regular expressions takes more than 4000 lines of code, which is more than half the size of all Lua standard libraries together. In comparison, the implementation of pattern matching in Lua has less than 600 lines. Of course, pattern matching in Lua cannot do all that a full POSIX implementation does. Nevertheless, pattern matching in Lua is a powerful tool, and includes some features that are difficult to match with standard POSIX implementations.

The Pattern-Matching Functions

The string library offers four functions based on *patterns*. We have already had a glimpse at `find` and `gsub`; the other two are `match` and `gmatch` (*Global Match*). Now we will see all of them in detail.

The function `string.find`

The function `string.find` searches for a pattern inside a given subject string. The simplest form of a pattern is a word, which matches only a copy of itself. For instance, the pattern '`hello`' will search for the substring `"hello"` inside the subject string. When `string.find` finds its pattern, it returns two values: the index where the match begins and the index where the match ends. If it does not find a match, it returns nil:

```
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 a match succeeds, we can call `string.sub` with the values returned by `find` to get the part of the subject string that matched the pattern. For simple patterns, this is necessarily the pattern itself.

The function `string.find` has two optional parameters. The third parameter is an index that tells where in the subject string to start the search. The fourth parameter, a Boolean, indicates a plain search. A plain search, as the name implies, does a plain “find substring” search in the subject, ignoring patterns:

```
> string.find("a [word]", "[")
stdin:1: malformed pattern (missing ']')
> string.find("a [word]", "[", 1, true)   --> 3   3
```

In the first call, the function complains because '`[`' has a special meaning in patterns. In the second call, the function treats '`[`' as a simple string. Note that we cannot pass the fourth optional parameter without the third one.

The function `string.match`

The function `string.match` is similar to `find`, in the sense that it also searches for a pattern in a string. However, instead of returning the positions where it found the pattern, it returns the part of the subject string that matched the pattern:

```
print(string.match("hello world", "hello"))   --> hello
```

For fixed patterns such as '`hello`', this function is pointless. It shows its power when used with variable patterns, as in the next example:

```
date = "Today is 17/7/1990"
d = string.match(date, "%d+/%d+/%d+")
print(d)  --> 17/7/1990
```

Shortly we will discuss the meaning of the pattern '`%d+/%d+/%d+`' and more advanced uses for `string.match`.

The function `string.gsub`

The function `string.gsub` has three mandatory parameters: a subject string, a pattern, and a replacement string. Its basic use is to substitute the replacement string for all occurrences of the pattern inside the subject 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
```

An optional fourth parameter limits the number of substitutions to be made:

```
s = string.gsub("all lii", "l", "x", 1)
print(s)          --> axl lii
s = string.gsub("all lii", "l", "x", 2)
print(s)          --> axx lii
```

Instead of a replacement string, the third argument to `string.gsub` can be also a function or a table, which is called (or indexed) to produce the replacement string; we will cover this feature in the section called “Replacements”.

The function `string.gsub` also returns as a second result the number of times it made the substitution.

The function `string.gmatch`

The function `string.gmatch` returns a function that iterates over all occurrences of a pattern in a string. For instance, the following example collects all words of a given string `s`:

```
s = "some string"
words = {}
for w in string.gmatch(s, "%a+") do
  words[#words + 1] = w
end
```

As we will discuss shortly, the pattern '`%a+`' matches sequences of one or more alphabetic characters (that is, words). So, the **for** loop will iterate over all words of the subject string, storing them in the list `words`.

Patterns

Most pattern-matching libraries use the backslash as an escape. However, this choice has some annoying consequences. For the Lua parser, patterns are regular strings. They have no special treatment and follow

the same rules as other strings. Only the pattern-matching functions interpret them as patterns. Because the backslash is the escape character in Lua, we have to escape it to pass it to any function. Patterns are naturally hard to read, and writing `"\\"` instead of `"\"` everywhere does not help.

We could ameliorate this problem with long strings, enclosing patterns between double brackets. (Some languages recommend this practice.) However, the long-string notation seems cumbersome for patterns, which are usually short. Moreover, we would lose the ability to use escapes inside patterns. (Some pattern-matching tools work around this limitation by reimplementing the usual string escapes.)

Lua's solution is simpler: patterns in Lua use the percent sign as an escape. (Several functions in C, such as `printf` and `strftime`, adopt the same solution.) In general, any escaped alphanumeric character has some special meaning (e.g., '`%a`' matches any letter), while any escaped non-alphanumeric character represents itself (e.g., '`%.`' matches a dot).

We will start our discussion about patterns with *character classes*. A character class is an item in a pattern that can match any character in a specific set. For instance, the class `%d` matches any digit. Therefore, we can search for a date in the format `dd/mm/yyyy` with the pattern '`%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.match(s, date))   --> 30/05/1999
```

The following table lists the predefined character classes with their meanings:

`.`	all characters
`%a`	letters
`%c`	control characters
`%d`	digits
`%g`	printable characters except spaces
`%l`	lower-case letters
`%p`	punctuation characters
`%s`	space characters
`%u`	upper-case letters
`%w`	alphanumeric characters
`%x`	hexadecimal digits

An upper-case version of any of these classes represents the complement of the class. For instance, '`%A`' represents all non-letter characters:

```
print((string.gsub("hello, up-down!", "%A", ".")))
  --> hello..up.down.
```

(When printing the results of `gsub`, I am using extra parentheses to discard its second result, which is the number of substitutions.)

Some characters, called *magic characters*, have special meanings when used in a pattern. Patterns in Lua use the following magic characters:

```
( ) . % + - * ? [ ] ^ $
```

As we have seen, the percent sign works as an escape for these magic characters. So, '`%?`' matches a question mark and '`%%`' matches a percent sign itself. We can escape not only the magic characters, but also any non-alphanumeric character. When in doubt, play safe and use an escape.

A *char-set* allows us to create our own character classes, grouping single characters and classes inside square brackets. For instance, the char-set '`[%w_]`' matches both alphanumeric characters and underscores, '`[01]`' matches binary digits, and '`[%[%]]`' matches square brackets. To count the number of vowels in a text, we can write this code:

```
_, nvow = string.gsub(text, "[AEIOUaeiou]", "")
```

We can also include character ranges in a char-set, by writing the first and the last characters of the range separated by a hyphen. I seldom use this feature, because most useful ranges are predefined; for instance, '`%d`' substitutes '`[0-9]`', and '`%x`' substitutes '`[0-9a-fA-F]`'. However, if you need to find an octal digit, you may prefer '`[0-7]`' instead of an explicit enumeration like '`[01234567]`'.

We can get the complement of any char-set by starting it with a caret: the pattern '`[^0-7]`' finds any character that is not an octal digit and '`[^\n]`' matches any character different from newline. Nevertheless, remember that you can negate simple classes with its upper-case version: '`%S`' is simpler than '`[^%s]`'.

We can make patterns still more useful with modifiers for repetitions and optional parts. Patterns in Lua offer four modifiers:

`+`	1 or more repetitions
`*`	0 or more repetitions
`-`	0 or more lazy repetitions
`?`	optional (0 or 1 occurrence)

The plus modifier matches one or more characters of the original class. It will always get the longest sequence that matches the pattern. For instance, the pattern '`%a+`' means one or more letters (a word):

```
print((string.gsub("one, and two; and three", "%a+", "word")))
  --> word, word word; word word
```

The pattern '`%d+`' matches one or more digits (an integer numeral):

```
print(string.match("the number 1298 is even", "%d+"))   --> 1298
```

The asterisk modifier is similar to plus, but it also accepts zero occurrences of characters of the class. A typical use is to match optional spaces between parts of a pattern. For instance, to match an empty parenthesis pair, such as `()` or `( )`, we can use the pattern '`%(%s*%)`', where the pattern '`%s*`' matches zero or more spaces. (Parentheses have a special meaning in a pattern, so we must escape them.) As another example, the pattern '`[_%a][_%w]*`' matches identifiers in a Lua program: a sequence starting with a letter or an underscore, followed by zero or more underscores or alphanumeric characters.

Like an asterisk, the minus modifier also matches zero or more occurrences of characters of the original class. However, instead of matching the longest sequence, it matches the shortest one. Sometimes there is no difference between asterisk and minus, but usually they give rather different results. For instance, if we try to find an identifier with the pattern '`[_%a][_%w]-`', we will find only the first letter, because the '`[_%w]-`' will always match the empty sequence. On the other hand, suppose we want to erase comments in a C program. Many people would first try '`/%*.*%*/`' (that is, a `"/*"` followed by a sequence of any characters followed by `"*/"`, written with the appropriate escapes). However, because the '`.*`' expands as far as it can, the first `"/*"` in the program would close only with the last `"*/"`:

```
test = "int x; /* x */  int y; /* y */"
print((string.gsub(test, "/%*.*%*/", "")))
  --> int x;
```

The pattern '`.-`', instead, will expand only as much as necessary to find the first `"*/"`, so that we get the desired result:

```
test = "int x; /* x */  int y; /* y */"
print((string.gsub(test, "/%*.-%*/", "")))
  --> int x;   int y;
```

The last modifier, the question mark, matches an optional character. As an example, suppose we want to find an integer in a text, where the number can contain an optional sign. The pattern '`[+-]?%d+`' does the job, matching numerals like `"-12"`, `"23"`, and `"+1009"`. The character class '`[+-]`' matches either a plus or a minus sign; the following `?` makes this sign optional.

Unlike some other systems, in Lua we can apply a modifier only to a character class; there is no way to group patterns under a modifier. For instance, there is no pattern that matches an optional word (unless the word has only one letter). Usually, we can circumvent this limitation using some of the advanced techniques that we will see in the end of this chapter.

If a pattern begins with a caret, it will match only at the beginning of the subject string. Similarly, if it ends with a dollar sign, it will match only at the end of the subject string. We can use these marks both to restrict the matches that we find and to anchor patterns. For instance, the next test checks whether the string `s` starts with a digit:

```
if string.find(s, "^%d") then ...
```

The next one checks whether that string represents an integer number, without any other leading or trailing characters:

```
if string.find(s, "^[+-]?%d+$") then ...
```

The caret and dollar signs are magic only when used in the beginning or end of the pattern. Otherwise, they act as regular characters matching themselves.

Another item in a pattern is '`%b`', which matches balanced strings. We write this item as '`%b`*xy*', where *x* and *y* are any two distinct characters; the *x* acts as an opening character and the *y* as the closing one. For instance, the pattern '`%b()`' matches parts of the string that start with a left parenthesis and finish at the respective right one:

```
s = "a (enclosed (in) parentheses) line"
print((string.gsub(s, "%b()", "")))     --> a  line
```

Typically, we use this pattern as '`%b()`', '`%b[]`', '`%b{}`', or '`%b<>`', but we can use any two distinct characters as delimiters.

Finally, the item '`%f[`*char-set*`]`' represents a *frontier pattern*. It matches an empty string only if the next character is in *char-set* but the previous one is not:

```
s = "the anthem is the theme"
print((string.gsub(s, "%f[%w]the%f[%W]", "one")))
  --> one anthem is one theme
```

The pattern '`%f[%w]`' matches a frontier between a non-alphanumeric and an alphanumeric character, and the pattern '`%f[%W]`' matches a frontier between an alphanumeric and a non-alphanumeric character. Therefore, the given pattern matches the string `"the"` only as an entire word. Note that we must write the char-set inside brackets, even when it is a single class.

The frontier pattern treats the positions before the first and after the last characters in the subject string as if they had the null character (ASCII code zero). In the previous example, the first `"the"` starts with a frontier between a null character, which is not in the set '`[%w]`', and a `t`, which is.

Captures

The *capture* mechanism allows a pattern to yank parts of the subject string that match parts of the pattern for further use. We specify a capture by writing the parts of the pattern that we want to capture between parentheses.

When a pattern has captures, the function `string.match` returns each captured value as a separate result; in other words, it breaks a string into its captured parts.

```
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; the pattern '`%s*`' specifies a possibly empty sequence of spaces. So, in the example above, the whole pattern specifies a sequence of letters, followed by a sequence of spaces, followed by an equals sign, again followed by spaces, plus another sequence of letters. Both sequences of letters have their patterns enclosed in parentheses, so that they will be captured if a match occurs. Below is a similar example:

```
date = "Today is 17/7/1990"
d, m, y = string.match(date, "(%d+)/(%d+)/(%d+)")
print(d, m, y)  --> 17  7  1990
```

In this example, we use three captures, one for each sequence of digits.

In a pattern, an item like '`%`*n*', where *n* is a single digit, matches only a copy of the *n*-th capture. As a typical use, suppose we want to find, inside a string, a substring enclosed between single or double quotes. We could try a pattern such as '`["']`.-`["']`', that is, a quote followed by anything followed by another quote; but we would have problems with strings like `"it's all right"`. To solve this problem, we can capture the first quote and use it to specify the second one:

```
s = [[then he said: "it's all right"!]]
q, quotedPart = string.match(s, "([\"'])(.-)%1")
print(quotedPart)   --> it's all right
print(q)            --> "
```

The first capture is the quote character itself and the second capture is the contents of the quote (the substring matching the '`.-`').

A similar example is this pattern, which matches long strings in Lua:

```
%[(=*)%[(.-)%]%1%]
```

It will match an opening square bracket followed by zero or more equals signs, followed by another opening square bracket, followed by anything (the string content), followed by a closing square bracket, followed by the same number of equals signs, followed by another closing square bracket:

```
p = "%[(=*)%[(.-)%]%1%]"
s = "a = [=[[[ something ]] ]==] ]=]; print(a)"
print(string.match(s, p))    --> =       [[ something ]] ]==]
```

The first capture is the sequence of equals signs (only one sign in this example); the second is the string content.

The third use of captured values is in the replacement string of `gsub`. Like the pattern, the replacement string can also contain items like `"%`*n*`"`, which are changed to the respective captures when the substitu-

tion is made. In particular, the item `"%0"` becomes the whole match. (By the way, a percent sign in the replacement string must be escaped as `"%%"`.) As an example, the following command duplicates every letter in a string, with a hyphen between the copies:

```
print((string.gsub("hello Lua!", "%a", "%0-%0")))
  --> h-he-el-ll-lo-o L-Lu-ua-a!
```

This one interchanges adjacent characters:

```
print((string.gsub("hello Lua", "(.)(.)", "%2%1")))
  --> ehll ouLa
```

As a more useful example, let us write a primitive format converter, which gets a string with commands written in a LaTeX style and changes them to a format in XML style:

```
\command{some text}    -->    <command>some text</command>
```

If we disallow nested commands, the following call to `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 <em>change</em> that.
```

(In the next section, we will see how to handle nested commands.)

Another useful example is how to trim a string:

```
function trim (s)
  s = string.gsub(s, "^%s*(.-)%s*$", "%1")
  return s
end
```

Note the judicious use of pattern modifiers. The two anchors (`^` and `$`) ensure that we get the whole string. Because the `'.-'` in the middle tries to expand as little as possible, the two enclosing patterns `'%s*'` match all spaces at both extremities.

Replacements

As we have seen already, we can use either a function or a table as the third argument to `string.gsub`, instead of a string. When invoked with a function, `string.gsub` calls the function every time it finds a match; the arguments to each call are the captures, and the value that the function returns becomes the replacement string. When invoked with a table, `string.gsub` looks up the table using the first capture as the key, and the associated value is used as the replacement string. If the result from the call or from the table lookup is nil, `gsub` does not change the match.

As a first example, the following function does variable expansion: it substitutes the value of the global variable `varname` for every occurrence of `$varname` in a string:

```
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?
```

(As we will discuss in detail in Chapter 22, *The Environment*, `_G` is a predefined table containing all global variables.) For each match with '`$(%w+)`' (a dollar sign followed by a name), `gsub` looks up the captured name in the global table `_G`; the result replaces the match. When the table does not have the key, there is no replacement:

```
print(expand("$othername is $status, isn't it?"))
  --> $othername is great, isn't it?
```

If we are not sure whether the given variables have string values, we may want to apply `tostring` to their values. In this case, we can use a function as the replacement value:

```
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
```

Inside `expand`, for each match with '`$(%w+)`', `gsub` calls the given function with the captured name as argument; the returned string replaces the match.

The last example goes back to our format converter from the previous section. Again, we want to convert commands in LaTeX style (`\example{text}`) to XML style (`<example>text</example>`), but allowing nested commands this time. The following function uses recursion to do the job:

```
function toxml (s)
  s = string.gsub(s, "\\(%a+)(%b{})", function (tag, body)
        body = string.sub(body, 2, -2)  -- remove the brackets
        body = toxml(body)              -- handle 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

Our next example will use *URL encoding*, which is the encoding used by HTTP to send parameters embedded in a URL. This encoding represents special characters (such as `=`, `&`, and `+`) as `"%`*xx*`"`, where *xx* is the character code in hexadecimal. After that, it changes spaces to plus signs. For instance, it encodes the string `"a+b = c"` as `"a%2Bb+%3D+c"`. Finally, it writes each parameter name and parameter value with an equals sign in between and appends all resulting pairs `name = value` with an ampersand in between. For instance, the values

```
name = "al";  query = "a+b = c"; q="yes or no"
```

are encoded as `"name=al&query=a%2Bb+%3D+c&q=yes+or+no"`.

Now, suppose we want to decode this URL and store each value in a table, indexed by its corresponding name. The following function does the basic 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

print(unescape("a%2Bb+%3D+c"))  --> a+b = c
```

The first `gsub` changes each plus sign in the string to a space. The second `gsub` matches all two-digit hexadecimal numerals preceded by a percent sign and calls an anonymous function for each match. This function converts the hexadecimal numeral into a number (using `tonumber` with base 16) and returns the corresponding character (`string.char`).

To decode the pairs `name=value`, we use `gmatch`. Because neither names nor values can contain either ampersands or equals signs, we can match them with 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 call to `gmatch` matches all pairs in the form `name=value`. For each pair, the iterator returns the corresponding captures (as marked by the parentheses in the matching string) as the values for `name` and `value`. The loop body simply applies `unescape` to both strings and stores the pair in the `cgi` table.

The corresponding encoding is also easy to write. First, we write the `escape` function; this function encodes all special characters as a percent sign followed by the character code in hexadecimal (the `format` option `"%02X"` makes a hexadecimal number with two digits, using 0 for padding), and then changes spaces to plus signs:

```
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 table to be encoded, building the resulting string:

```
function encode (t)
  local b = {}
  for k,v in pairs(t) do
    b[#b + 1] = (escape(k) .. "=" .. escape(v))
  end
  -- concatenates all entries in 'b', separated by "&"
  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
```

Tab expansion

An empty capture like '`()`' has a special meaning in Lua. Instead of capturing nothing (a useless task), this pattern captures its position in the subject string, as a number:

```
print(string.match("hello", "()ll()"))   --> 3  5
```

(Note that the result of this example is not the same as what we get from `string.find`, because the position of the second empty capture is *after* the match.)

A nice example of the use of position captures is for expanding tabs in a string:

```
function expandTabs (s, tab)
  tab = tab or 8       -- tab "size" (default is 8)
  local corr = 0       -- correction
  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` pattern matches all tabs in the string, capturing their positions. For each tab, the anonymous function uses this position to compute the number of spaces needed to arrive at a column that is a multiple of `tab`: it subtracts one from the position to make it relative to zero and adds `corr` to compensate for previous tabs. (The expansion of each tab affects the position of the following ones.) It then updates the correction for the next tab: minus one for the tab being removed, plus `sp` for the spaces being added. Finally, it returns a string with the appropriate number of spaces to replace the tab.

Just for completeness, let us see how to reverse this operation, converting spaces to tabs. A first approach could also involve the use of empty captures to manipulate positions, but there is a simpler solution: at every eighth character, we insert a mark in the string. Then, wherever the mark is preceded by spaces, we replace the sequence spaces–mark by a tab:

```
function unexpandTabs (s, tab)
  tab = tab or 8
  s = expandTabs(s, tab)
  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
```

The function starts by expanding the string to remove any previous tabs. Then it computes an auxiliary pattern for matching all sequences of eight characters, and uses this pattern to add a mark (the control character `\1`) after every eight characters. It then substitutes a tab for all sequences of one or more spaces followed by a mark. Finally, it removes the marks left (those not preceded by spaces).

Tricks of the Trade

Pattern matching is a powerful tool for manipulating strings. We can perform many complex operations with only a few calls to `string.gsub`. However, as with any power, we must use it carefully.

Pattern matching is not a replacement for a proper parser. For quick-and-dirty programs, we can do useful manipulations on source code, but it may be hard to build a product with quality. As a good example, consider the pattern we used to match comments in a C program: '`/%*.-%*/`'. If the program has a literal string containing `"/*"`, we may get a wrong result:

```
test = [[char s[] = "a /* here";  /* a tricky string */]]
print((string.gsub(test, "/%*.-%*/", "<COMMENT>")))
  --> char s[] = "a <COMMENT>
```

Strings with such contents are rare. For our own use, that pattern will probably do its job, but we should not distribute a program with such a flaw.

Usually, pattern matching is efficient enough for Lua programs: my new machine takes less than 0.2 seconds to count all words in a 4.4 MB text (850 K-words).[1] But we can take precautions. We should always make the pattern as specific as possible; loose patterns are slower than specific ones. An extreme example is '`(.-)%$`', to get all text in a string up to the first dollar sign. If the subject string has a dollar sign, everything goes fine, but suppose that the string does not contain any dollar signs. The algorithm will first try to match the pattern starting at the first position of the string. It will go through all the string, looking for a dollar. When the string ends, the pattern fails *for the first position* of the string. Then, the algorithm will do the whole search again, starting at the second position of the string, only to discover that the pattern does not match there, too, repeating the search for every position in the string. This will take a quadratic time, resulting in more than four minutes in my new machine for a string of 200K characters. We can correct this problem simply by anchoring the pattern at the first position of the string, with '`^(.-)%$`'. The anchor tells the algorithm to stop the search if it cannot find a match at the first position. With the anchor, the match runs in a hundredth of a second.

Beware also of empty patterns, that is, patterns that match the empty string. For instance, if we try to match names with a pattern like '`%a*`', we will find names everywhere:

```
i, j = string.find(";$%  **#$hello13", "%a*")
print(i,j)   --> 1  0
```

In this example, the call to `string.find` has correctly found an empty sequence of letters at the beginning of the string.

It never makes sense to write a pattern that ends with the minus modifier, because it will match only the empty string. This modifier always needs something after it to anchor its expansion. Similarly, patterns that include '`.*`' are tricky, because this construction can expand much more than we intended.

Sometimes, it is useful to use Lua itself to build a pattern. We already used this trick in our function to convert spaces to tabs. As another example, let us see how we can find long lines in a text, for instance lines with more than 70 characters. A long line is a sequence of 70 or more characters different from newline. We can match a single character different from newline with the character class '`[^\n]`'. Therefore, we can match a long line with a pattern that repeats 70 times the pattern for one character, finishing in a repetition for that pattern (to match the rest of the line). Instead of writing this pattern by hand, we can create it with `string.rep`:

```
pattern = string.rep("[^\n]", 70) .. "+"
```

As another example, suppose we want to make a case-insensitive search. A way of doing this is to change any letter *x* in the pattern to the class '`[xX]`', that is, a class including both the lower and the upper-case versions of the original letter. We can automate this conversion with a function:

```
function nocase (s)
```

[1] "My new machine" is an Intel Core i7-4790 3.6 GHz, with 8 GB of RAM. I measured all performance data in this book on that machine.

```
  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, we want to replace every plain occurrence of `s1` with `s2`, without regarding any character as magic. If the strings `s1` and `s2` are literals, we can add proper escapes to magic characters while we write the strings. If these strings are variable values, we can use another `gsub` to put the escapes for us:

```
s1 = string.gsub(s1, "(%W)", "%%%1")
s2 = string.gsub(s2, "%%", "%%%%")
```

In the search string, we escape all non-alphanumeric characters (thus the upper-case `W`). In the replacement string, we escape only the percent sign.

Another useful technique for pattern matching is to preprocess the subject string before the real work. Suppose we want to change to upper case all quoted strings in a text, where a quoted string starts and ends with a double quote (`"`), but may contain escaped quotes (`"\""`):

```
follows a typical string: "This is \"great\"!".
```

One approach for handling such cases is to preprocess the text to encode the problematic sequence as something else. For instance, we could code `"\""` as `"\1"`. However, if the original text already contains a `"\1"`, we are in trouble. An easy way to do the encoding and avoid this problem is to code all sequences `"\`*x*`"` as `"\`*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*`"` in the encoded string must have come from the coding, because any `"\`*ddd*`"` in the original string has been coded, too. So, the decoding is an easy task:

```
function decode (s)
  return (string.gsub(s, "\\(%d%d%d)", function (d)
            return "\\" .. string.char(tonumber(d))
          end))
end
```

Now we can complete our task. As the encoded string does not contain any escaped quote (`"\""`), we can search for quoted strings simply with `'".-"'`:

```
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\"!".
```

We can also write it like here:

```
print(decode(string.gsub(code(s), '".-"', string.upper)))
```

The applicability of pattern-matching functions to UTF-8 strings depends on the pattern. Literal patterns work without problems, due to the key property of UTF-8 that the encoding of any character never appears inside the encoding of any other character. Character classes and character sets work only for ASCII characters. For instance, the pattern `'%s'` works on UTF-8 strings, but it will match only the ASCII white spaces; it will not match extra Unicode white spaces such as a non-break space (U+00A0) or a Mongolian vowel separator (U+180E).

Judicious patterns can bring some extra power to Unicode handling. A good example is the predefined pattern `utf8.charpattern`, which matches exactly one UTF-8 character. The `utf8` library defines this pattern as follows:

```
utf8.charpattern = [\0-\x7F\xC2-\xF4][\x80-\xBF]*
```

The first part is a class that matches either ASCII characters (range [0, 0x7F]) or initial bytes for multibyte sequences (range [0xC2, 0xF4]). The second part matches zero or more continuation bytes (range [0x80, 0xBF]).

Exercises

Exercise 10.1: Write a function `split` that receives a string and a delimiter pattern and returns a sequence with the chunks in the original string separated by the delimiter:

```
t = split("a whole new world", " ")
-- t = {"a", "whole", "new", "world"}
```

How does your function handle empty strings? (In particular, is an empty string an empty sequence or a sequence with one empty string?)

Exercise 10.2: The patterns `'%D'` and `'[^%d]'` are equivalent. What about the patterns `'[^%d%u]'` and `'[%D%U]'`?

Exercise 10.3: Write a function transliterate. This function receives a string and replaces each character in that string with another character, according to a table given as a second argument. If the table maps `a` to `b`, the function should replace any occurrence of `a` with `b`. If the table maps `a` to **false**, the function should remove occurrences of `a` from the resulting string.

Exercise 10.4: At the end of the section called “Captures”, we defined a `trim` function. Because of its use of backtracking, this function can take a quadratic time for some strings. (For instance, in my new machine, a match for a 100 KB string can take 52 seconds.)

- Create a string that triggers this quadratic behavior in function `trim`.
- Rewrite that function so that it always works in linear time.

Exercise 10.5: Write a function to format a binary string as a literal in Lua, using the escape sequence `\x` for all bytes:

```
print(escape("\0\1hello\200"))
  --> \x00\x01\x68\x65\x6C\x6C\x6F\xC8
```

As an improved version, use also the escape sequence `\z` to break long lines.

Exercise 10.6: Rewrite the function transliterate for UTF-8 characters.

Exercise 10.7: Write a function to reverse a UTF-8 string.

Chapter 11. Interlude: Most Frequent Words

In this interlude we will develop a program that reads a text and prints the most frequent words in that text. As in the previous interlude, the program here is quite simple, but it uses some more advanced features, such as iterators and anonymous functions.

The main data structure of our program is a table that maps each word found in the text to its frequency counter. With this data structure, the program has three main tasks:

- Read the text, counting the number of occurrences of each word.
- Sort the list of words in descending order of frequencies.
- Print the first *n* entries in the sorted list.

To read the text, we can iterate over all its lines and, for each line, we iterate over all its words. For each word that we read, we increment its respective counter:

```
local counter = {}

for line in io.lines() do
  for word in string.gmatch(line, "%w+") do
    counter[word] = (counter[word] or 0) + 1
  end
end
```

Here, we describe a “word” using the pattern `'%w+'`, that is, one or more alphanumeric characters.

The next step is to sort the list of words. However, as the attentive reader may have noticed already, we do not have a list of words to sort! Nevertheless, it is easy to create one, using the words that appear as keys in table `counter`:

```
local words = {}    -- list of all words found in the text

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

Once we have the list, we can sort it using `table.sort`:

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

Remember that the order function must return true when `w1` must come before `w2` in the result. Words with larger counters come first; words with equal counters come in alphabetical order.

Figure 11.1, “Word-frequency program” presents the complete program.

Figure 11.1. Word-frequency program

```
local counter = {}

for line in io.lines() do
  for word in string.gmatch(line, "%w+") do
    counter[word] = (counter[word] or 0) + 1
  end
end

local words = {}    -- list of all words found in the text

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)

-- number of words to print
local n = math.min(tonumber(arg[1]) or math.huge, #words)

for i = 1, n do
  io.write(words[i], "\t", counter[words[i]], "\n")
end
```

The last loop prints the result, which is the first `n` words and their respective counters. The program assumes that its first argument is the number of words to be printed; by default, it prints all words if no argument is given.

As an example, we show the result of applying this program over this book:

```
$ lua wordcount.lua 10 < book.of
the 5996
a 3942
to 2560
is 1907
of 1898
in 1674
we 1496
function 1478
and 1424
x 1266
```

Exercises

Exercise 11.1: When we apply the word-frequency program to a text, usually the most frequent words are uninteresting small words like articles and prepositions. Change the program so that it ignores words with less than four letters.

Exercise 11.2: Repeat the previous exercise but, instead of using length as the criterion for ignoring a word, the program should read from a text file a list of words to be ignored.

Chapter 12. Date and Time

The standard libraries offer few functions to manipulate date and time in Lua. As usual, all it offers is what is available in the standard C libraries. Nevertheless, despite its apparent simplicity, we can make quite a lot with this basic support.

Lua uses two representations for date and time. The first one is through a single number, usually an integer. Although not required by ISO C, on most systems this number is the number of seconds since some fixed date, called the *epoch*. In particular, both in POSIX and Windows systems the epoch is Jan 01, 1970, 0:00 UTC.

The second representation that Lua uses for dates and times is a table. Such *date tables* have the following significant fields: `year`, `month`, `day`, `hour`, `min`, `sec`, `wday`, `yday`, and `isdst`. All fields except `isdst` have integer values. The first six fields have obvious meanings. The `wday` field is the day of the week (one is Sunday); the `yday` field is the day of the year (one is January 1st). The `isdst` field is a Boolean, true if daylight saving is in effect. As an example, Sep 16, 1998, 23:48:10 (a Wednesday) corresponds to the following table:

```
{year = 1998, month = 9, day = 16, yday = 259, wday = 4,
 hour = 23, min = 48, sec = 10, isdst = false}
```

Date tables do not encode a time zone. It is up to the program to interpret them correctly with respect to time zones.

The Function `os.time`

The function `os.time`, when called without arguments, returns the current date and time, coded as a number:

```
> os.time()          --> 1439653520
```

This date corresponds to Aug 15, 2015, 12:45:20.[1] In a POSIX system, we can use some basic arithmetic to decompose that number:

```
local date = 1439653520
local day2year = 365.242                 -- days in a year
local sec2hour = 60 * 60                 -- seconds in an hour
local sec2day = sec2hour * 24            -- seconds in a day
local sec2year = sec2day * day2year      -- seconds in a year

-- year
print(date // sec2year + 1970)          --> 2015.0

-- hour (in UTC)
print(date % sec2day // sec2hour)       --> 15

-- minutes
print(date % sec2hour // 60)            --> 45

-- seconds
print(date % 60)                        --> 20
```

[1] Unless otherwise stated, my dates are from a POSIX system running in Rio de Janeiro.

We can also call `os.time` with a date table, to convert the table representation to a number. The `year`, `month`, and `day` fields are mandatory. The `hour`, `min`, and `sec` fields default to noon (12:00:00) when not provided. Other fields (including `wday` and `yday`) are ignored.

```
> os.time({year=2015, month=8, day=15, hour=12, min=45, sec=20})
  --> 1439653520
> os.time({year=1970, month=1, day=1, hour=0})    --> 10800
> os.time({year=1970, month=1, day=1, hour=0, sec=1})
  --> 10801
> os.time({year=1970, month=1, day=1})            --> 54000
```

Note that 10800 is three hours (the time zone) in seconds and 54000 is 10800 plus 12 hours in seconds.

The Function `os.date`

The function `os.date`, despite its name, is a kind of reverse of `os.time`: it converts a number representing the date and time to some higher-level representation, either a date table or a string. Its first parameter is a *format string*, describing the representation we want. The second parameter is the numeric date–time; it defaults to the current date and time if not provided.

To produce a date table, we use the format string `"*t"`. For instance, the call `os.date("*t", 906000490)` returns the following table:

```
{year = 1998, month = 9, day = 16, yday = 259, wday = 4,
 hour = 23, min = 48, sec = 10, isdst = false}
```

In general, we have that `os.time(os.date("*t", t)) == t`, for any valid time `t`.

Except for `isdst`, the resulting fields are integers in the following ranges:

`year`	a full year
`month`	1–12
`day`	1–31
`hour`	0–23
`min`	0–59
`sec`	0–60
`wday`	1–7
`yday`	1–366

(Seconds can go up to 60 to allow for leap seconds.)

For other format strings, `os.date` returns a copy of the string with specific directives replaced by information about the given time and date. A directive consists of a percent sign followed by a letter, as in the next example:

```
print(os.date("a %A in %B"))            --> a Tuesday in May
print(os.date("%d/%m/%Y", 906000490))   --> 16/09/1998
```

When relevant, representations follow the current locale. For instance, in a locale for Brazil–Portuguese, `%A` would result in `"terça-feira"` and `%B` in `"maio"`.

Figure 12.1, “Directives for function `os.date`” shows the main directives. For each directive, it presents its meaning and its value for September 16, 1998 (a Wednesday), at 23:48:10.

Figure 12.1. Directives for function `os.date`

%a	abbreviated weekday name (e.g., Wed)
%A	full weekday name (e.g., Wednesday)
%b	abbreviated month name (e.g., Sep)
%B	full month name (e.g., September)
%c	date and time (e.g., 09/16/98 23:48:10)
%d	day of the month (16) [01–31]
%H	hour, using a 24-hour clock (23) [00–23]
%I	hour, using a 12-hour clock (11) [01–12]
%j	day of the year (259) [001–365]
%m	month (09) [01–12]
%M	minute (48) [00–59]
%p	either "am" or "pm" (pm)
%S	second (10) [00–60]
%w	weekday (3) [0–6 = Sunday–Saturday]
%W	week of the year (37) [00–53]
%x	date (e.g., 09/16/98)
%X	time (e.g., 23:48:10)
%y	two-digit year (98) [00–99]
%Y	full year (1998)
%z	timezone (e.g., -0300)
%%	a percent sign

For numerical values, the table shows also their range of possible values. Here are some examples, showing how to create some ISO 8601 formats:

```
t = 906000490
-- ISO 8601 date
print(os.date("%Y-%m-%d", t))           --> 1998-09-16
-- ISO 8601 combined date and time
print(os.date("%Y-%m-%dT%H:%M:%S", t))  --> 1998-09-16T23:48:10
-- ISO 8601 ordinal date
print(os.date("%Y-%j", t))              --> 1998-259
```

If the format string starts with an exclamation mark, then `os.date` interprets the time in UTC:

```
-- the Epoch
print(os.date("!%c", 0))     --> Thu Jan  1 00:00:00 1970
```

If we call `os.date` without any arguments, it uses the `%c` format, that is, date and time information in a reasonable format. Note that the representations for `%x`, `%X`, and `%c` change according to the locale and the system. If you want a fixed representation, such as `dd/mm/yyyy`, use an explicit format string, such as `"%d/%m/%Y"`.

Date–Time Manipulation

When `os.date` creates a date table, its fields are all in the proper ranges. However, when we give a date table to `os.time`, its fields do not need to be normalized. This feature is an important tool to manipulate dates and times.

As a simple example, suppose we want to know the date 40 days from now. We can compute that date as follows:

```
t = os.date("*t")        -- get current time
print(os.date("%Y/%m/%d", os.time(t)))     --> 2015/08/18
t.day = t.day + 40
print(os.date("%Y/%m/%d", os.time(t)))     --> 2015/09/27
```

If we convert the numeric time back to a table, we get a normalized version of that date–time:

```
t = os.date("*t")
print(t.day, t.month)             -->  26    2
t.day = t.day - 40
print(t.day, t.month)             --> -14    2
t = os.date("*t", os.time(t))
print(t.day, t.month)             -->  17    1
```

In this example, Feb -14 has been normalized to Jan 17, which is 40 days before Feb 26.

In most systems, we could also add or subtract 3456000 (40 days in seconds) to the numeric time. However, the C standard does not guarantee the correctness of this operation, because it does not require numeric times to denote seconds from some epoch. Moreover, if we want to add some months instead of days, the direct manipulation of seconds becomes problematic, as different months have different durations. The normalization method, on the other hand, has none of these problems:

```
t = os.date("*t")        -- get current time
print(os.date("%Y/%m/%d", os.time(t)))     --> 2015/08/18
t.month = t.month + 6    -- six months from now
print(os.date("%Y/%m/%d", os.time(t)))     --> 2016/02/18
```

We have to be careful when manipulating dates. Normalization works in a somewhat obvious way, but it may have some non-obvious consequences. For instance, if we compute one month after March 31, that would give April 31, which is normalized to May 1 (one day after April 30). That sounds quite natural. However, if we take one month back from that result (May 1), we arrive on April 1, not the original March 31. Note that this mismatch is a consequence of the way our calendar works; it has nothing to do with Lua.

To compute the difference between two times, there is the function `os.difftime`. It returns the difference, in seconds, between two given numeric times. For most systems, this difference is exactly the result of subtracting on time from the other. Unlike the subtraction, however, the behavior of `os.difftime` is guaranteed in any system. The next example computes the number of days passed between the release of Lua 5.2 and Lua 5.3:

```
local t5_3 = os.time({year=2015, month=1, day=12})
local t5_2 = os.time({year=2011, month=12, day=16})
local d = os.difftime(t5_3, t5_2)
print(d // (24 * 3600))         --> 1123.0
```

With `difftime`, we can express dates as number of seconds since any arbitrary epoch:

```
> myepoch = os.time{year = 2000, month = 1, day = 1, hour = 0}
```

```
> now = os.time{year = 2015, month = 11, day = 20}
> os.difftime(now, myepoch)     --> 501336000.0
```

Using normalization, it is easy to convert that number of seconds back to a legitimate numeric time: we create a table with the epoch and set its seconds as the number we want to convert, as in the next example.

```
> T = {year = 2000, month = 1, day = 1, hour = 0}
> T.sec = 501336000
> os.date("%d/%m/%Y", os.time(T))   --> 20/11/2015
```

We can also use `os.difftime` to compute the running time of a piece of code. For this task, however, it is better to use `os.clock`. The function `os.clock` returns the number of seconds of CPU time used by the program. Its typical use is to benchmark a piece of code:

```
local x = os.clock()
local s = 0
for i = 1, 100000 do s = s + i end
print(string.format("elapsed time: %.2f\n", os.clock() - x))
```

Unlike `os.time`, `os.clock` usually has sub-second precision, so its result is a float. The exact precision depends on the platform; in POSIX systems, it is typically one microsecond.

Exercises

Exercise 12.1: Write a function that returns the date–time exactly one month after a given date–time. (Assume the numeric coding of date–time.)

Exercise 12.2: Write a function that returns the day of the week (coded as an integer, one is Sunday) of a given date.

Exercise 12.3: Write a function that takes a date–time (coded as a number) and returns the number of seconds passed since the beginning of its respective day.

Exercise 12.4: Write a function that takes a year and returns the day of its first Friday.

Exercise 12.5: Write a function that computes the number of complete days between two given dates.

Exercise 12.6: Write a function that computes the number of complete months between two given dates.

Exercise 12.7: Does adding one month and then one day to a given date give the same result as adding one day and then one month?

Exercise 12.8: Write a function that produces the system's time zone.

Chapter 13. Bits and Bytes

Lua handles binary data similarly to text. A string in Lua can contain any bytes, and almost all library functions that handle strings can handle arbitrary bytes. We can even do pattern matching on binary data. On top of that, Lua 5.3 introduced extra facilities to manipulate binary data: besides integer numbers, it brought bitwise operators and functions to pack and unpack binary data. In this chapter, we will cover these and other facilities for handling binary data in Lua.

Bitwise Operators

Starting with version 5.3, Lua offers a standard set of bitwise operators on numbers. Unlike arithmetic operations, bitwise operators only work on integer values. The bitwise operators are & (bitwise AND), | (bitwise OR), ~ (bitwise exclusive-OR), >> (logical right shift), << (left shift), and the unary ~ (bitwise NOT). (Note that, in several languages, the exclusive-OR operator is denoted by ^. In Lua, ^ means exponentiation.)

```
> string.format("%x", 0xff & 0xabcd)    --> cd
> string.format("%x", 0xff | 0xabcd)    --> abff
> string.format("%x", 0xaaaa ~ -1)      --> ffffffffffff5555
> string.format("%x", ~0)               --> ffffffffffffffff
```

(Several examples in this chapter will use `string.format` to show results in hexadecimal.)

All bitwise operators work on all bits of integers. In Standard Lua, that means 64 bits. That can be a problem when implementing algorithms that assume 32-bit integers (e.g., the cryptographic hash SHA-2). However, it is not difficult to perform 32-bit integer manipulation. Except for the right-shift operation, all bitwise operations on 64 bits agree with the same operations on 32 bits, if we simply ignore the higher half bits. The same is true for addition, subtraction, and multiplication. So, all we have to do to operate on 32-bit integers is to erase the higher 32 bits of an integer before a right shift. (We seldom do divisions on that kind of computations.)

Both shift operators fill with zeros the vacant bits. This is usually called logical shifts. Lua does not offer an arithmetic right shift, which fills vacant bits with the signal bit. We can perform the equivalent to arithmetic shifts with a floor division by an appropriate power of two. (For instance, `x // 16` is the same as an arithmetic shift by four.)

Negative displacements shift in the other direction, that is, `a >> n` is the same as `a << -n`:

```
> string.format("%x", 0xff << 12)    --> ff000
> string.format("%x", 0xff >> -12)   --> ff000
```

If the displacement is equal to or larger than the number of bits in the integer representation (64 in Standard Lua, 32 in Small Lua), the result is zero, as all bits are shifted out of the result:

```
> string.format("%x", -1 << 80)    --> 0
```

Unsigned Integers

The representation of integers uses one bit to store the signal. Therefore, the maximum integer that we can represent with 64-bit integers is $2^{63} - 1$, instead of $2^{64} - 1$. Usually, this difference is irrelevant, as $2^{63} - 1$ is quite large already. However, sometimes we cannot waste a bit for the signal, because we are either handling external data with unsigned integers or implementing some algorithm that needs integers with all their 64 bits. Moreover, in Small Lua the difference can be quite significant. For instance, if we use a 32-bit signed integer as a position in a file, we are limited to 2 GB files; an unsigned integer doubles that limit.

Lua does not offer explicit support for unsigned integers. Nevertheless, with some care, it is not difficult to handle unsigned integers in Lua, as we will see now.

We can write constants larger than $2^{63} - 1$ directly, despite appearances:

```
> x = 13835058055282163712          -- 3 << 62
> x                                 --> -4611686018427387904
```

The problem here is not the constant, but the way Lua prints it: the standard way to print numbers interprets them as signed integers. We can use the `%u` or `%x` options in `string.format` to see integers as unsigned:

```
> string.format("%u", x)            --> 13835058055282163712
> string.format("0x%X", x)          --> 0xC000000000000000
```

Due to the way signed integers are represented (two's complement), the operations of addition, subtraction, and multiplication work the same way for signed and unsigned integers:

```
> string.format("%u", x)              --> 13835058055282163712
> string.format("%u", x + 1)          --> 13835058055282163713
> string.format("%u", x - 1)          --> 13835058055282163711
```

(With such a large value, multiplying `x` even by two would overflow, so we did not include that operation in the example.)

Order operators work differently for signed and unsigned integers. The problem appears when we compare two integers with a difference in the higher bit. For signed integers, the integer with that bit set is the smaller, because it represents a negative number:

```
> 0x7fffffffffffffff < 0x8000000000000000    --> false
```

This result is clearly incorrect if we regard both integers as unsigned. So, we need a different operation to compare unsigned integers. Lua 5.3 provides `math.ult` (*unsigned less than*) for that need:

```
> math.ult(0x7fffffffffffffff, 0x8000000000000000)   --> true
```

Another way to do the comparison is to flip the signal bit of both operands before doing a signed comparison:

```
> mask = 0x8000000000000000
> (0x7fffffffffffffff ~ mask) < (0x8000000000000000 ~ mask)
  --> true
```

Unsigned division is also different from its signed version. Figure 13.1, “Unsigned division” shows an algorithm for unsigned division.

Figure 13.1. Unsigned division

```
function udiv (n, d)
  if d < 0 then
    if math.ult(n, d) then return 0
    else return 1
    end
  end
  local q = ((n >> 1) // d) << 1
  local r = n - q * d
  if not math.ult(r, d) then q = q + 1 end
  return q
end
```

The first test (`d < 0`) is equivalent to testing whether `d` is larger than 2^{63}. In that case, the quotient can only be 1 (if `n` is equal to or larger than `d`) or 0. Otherwise, we do the equivalent of dividing the dividend by two, then dividing the result by the divisor, and then multiplying the result by two. The right shift is equivalent to an unsigned division by two; the result will be a non-negative signed integer. The subsequent left shift corrects the quotient, undoing this previous division.

In general, `floor(floor(n / 2) / d) * 2` (the computation done by the algorithm) is not equal to `floor(((n / 2) / d) * 2)` (the correct result). However, it is not difficult to prove that the difference is at most one. So, the algorithm computes the rest of the division (in the variable `r`) and checks whether it is greater than the divisor: if so, it corrects the quotient (adding one to it) and it is done.

Converting an unsigned integer to/from a float needs some adjustments. To convert an unsigned integer to a float, we can convert it as a signed integer and correct the result with a modulo operator:

```
> u = 11529215046068469760          -- an example
> f = (u + 0.0) % 2^64
> string.format("%.0f", f)          --> 11529215046068469760
```

The value of `u + 0.0` is -6917529027641081856, because the standard conversion sees `u` as a signed integer. The modulo operation brings the value back to the range of unsigned integers. (In real code we do not need the addition, because the modulo with a float would do the conversion anyway.)

To convert from a float to an unsigned integer, we can use the following code:

```
> f = 0xA000000000000000.0              -- an example
> u = math.tointeger(((f + 2^63) % 2^64) - 2^63)
> string.format("%x", u)                --> a000000000000000
```

The addition transforms a value greater than 2^{63} in a value greater than 2^{64}. The modulo operator then projects this value to the range $[0,2^{63})$, and the subtraction makes it a “negative” value (that is, a value with the highest bit set). For a value smaller than 2^{63}, the addition keeps it smaller than 2^{64}, the modulo operator does not affect it, and the subtraction restores its original value.

Packing and Unpacking Binary Data

Lua 5.3 also introduced functions for converting between binary data and basic values (numbers and strings). The function `string.pack` “packs” values into a binary string; `string.unpack` extracts those values from the string.

Both `string.pack` and `string.unpack` get as their first argument a format string, which describes how the data is packed. Each letter in this string describes how to pack/unpack one value; see the following example:

```
> s = string.pack("iii", 3, -27, 450)
> #s                                --> 12
> string.unpack("iii", s)           --> 3   -27   450   13
```

This call to `string.pack` creates a string with the binary codes of three integers (according to the description `"iii"`), each encoding its corresponding argument. The string length will be three times the native size of an integer (3 times 4 bytes in my machine). The call to `string.unpack` decodes three integers (again according to `"iii"`) from the given string and returns the decoded values.

The function `string.unpack` also returns the position in the string after the last item read, to simplify iterations. (This explains the 13 in the results of the last example.) Accordingly, it accepts an optional third argument, which tells where to start reading. For instance, the next example prints all strings packed inside a given string:

```
s = "hello\0Lua\0world\0"
local i = 1
while i <= #s do
  local res
  res, i = string.unpack("z", s, i)
  print(res)
end
  --> hello
  --> Lua
  --> world
```

As we will see, the `z` option means a zero-terminated string, so that the call to `unpack` extracts the string at position `i` from `s` and returns that string plus the next position for the loop.

There are several options for coding an integer, each corresponding to a native integer size: `b` (`char`), `h` (`short`), `i` (`int`), and `l` (`long`); the option `j` uses the size of a Lua integer. To use a fixed, machine-independent size, we can suffix the `i` option with a number from one to 16. For instance, `i7` will produce seven-byte integers. All sizes check for overflows:

```
> x = string.pack("i7", 1 << 54)
> string.unpack("i7", x)             --> 18014398509481984   8
> x = string.pack("i7", -(1 << 54))
> string.unpack("i7", x)             --> -18014398509481984  8
> x = string.pack("i7", 1 << 55)
stdin:1: bad argument #2 to 'pack' (integer overflow)
```

We can pack and unpack integers wider than native Lua integers but, when unpacking, their actual values must fit into Lua integers:

```
> x = string.pack("i12", 2^61)
> string.unpack("i12", x)      --> 2305843009213693952    13
> x = "aaaaaaaaaaaa"           -- fake a large 12-byte number
> string.unpack("i12", x)
stdin:1: 12-byte integer does not fit into Lua Integer
```

Each integer option has an upper-case version corresponding to an unsigned integer of the same size:

```
> s = "\xFF"
> string.unpack("b", s)       --> -1    2
> string.unpack("B", s)       --> 255   2
```

Moreover, unsigned integers have an extra option `T` for `size_t`. (The `size_t` type in ISO C is an unsigned integer larger enough to hold the size of any object.)

We can pack strings in three representations: zero-terminated strings, fixed-length strings, and strings with explicit length. Zero-terminated strings use the `z` option. For fixed-length strings, we use the option `c`*n*, where `n` is the number of bytes in the packed string. The last option for strings stores the string preceded by its length. In that case, the option has the format `s`*n*, where `n` is the size of the unsigned integer used to store the length. For instance, the option `s1` stores the string length in one byte:

```
s = string.pack("s1", "hello")
for i = 1, #s do print((string.unpack("B", s, i))) end
  --> 5                          (length)
  --> 104                        ('h')
  --> 101                        ('e')
  --> 108                        ('l')
  --> 108                        ('l')
```

```
  --> 111                           ('o')
```

Lua raises an error if the length does not fit into the given size. We can also use a pure `s` as the option; in that case, the length is stored as a `size_t`, which is large enough to hold the length of any string. (In 64-bit machines, `size_t` usually is an eight-byte unsigned integer, which may be a waste of space for small strings.)

For floating-point numbers, there are three options: `f` for single precision, `d` for double precision, and `n` for a Lua float.

The format string also has options to control the endianess and the alignment of the binary data. By default, a format uses the machine's native endianess. The `>` option turns all subsequent encodings in that format to big endian, or *network byte order*:

```
s = string.pack(">i4", 1000000)
for i = 1, #s do print((string.unpack("B", s, i))) end
  --> 0
  --> 15
  --> 66
  --> 64
```

The `<` option turns to little endian:

```
s = string.pack("<i2 i2", 500, 24)
for i = 1, #s do print((string.unpack("B", s, i))) end
  --> 244
  --> 1
  --> 24
  --> 0
```

Finally, the `=` option turns back to the default machine's native endianess.

For alignment, the `!`*n* option forces data to align at indices that are multiples of `n`. More specifically, if the item is smaller than `n`, it is aligned at its own size; otherwise, it is aligned at `n`. For instance, suppose we start the format string with `!4`. Then, one-byte integers will be written in indices multiple of one (that is, any index), two-byte integers will be written in indices multiple of two, and four-byte or larger integers will be written in indices multiple of four. The `!` option (without a number) sets the alignment to the machine's native alignment.

The function `string.pack` does the alignment by adding zeros to the resulting string until the index has a proper value. The function `string.unpack` simply skips the padding when reading the string. Alignment only works for powers of two: if we set the alignment to four and try to manipulate a three-byte integer, Lua will raise an error.

Any format string works as if prefixed by `"=!1"`, which means native endianess and no alignment (as every index is a multiple of one). We can change the endianess and the alignment at any point during the translation.

If needed, we can add padding manually. The option `x` means one byte of padding; `string.pack` adds a zero byte to the resulting string; `string.unpack` skips one byte from the subject string.

Binary files

The functions `io.input` and `io.output` always open a file in *text mode*. In POSIX, there is no difference between binary files and text files. In some systems like Windows, however, we must open binary files in a special way, using the letter `b` in the mode string of `io.open`.

Typically, we read binary data either with the `"a"` pattern, that reads the whole file, or with the pattern *n*, that reads *n* bytes. (Lines make no sense in a binary file.) As a simple example, the following program converts a text file from Windows format to POSIX format —that is, it translates sequences of carriage return–newlines to newlines:

```
local inp = assert(io.open(arg[1], "rb"))
local out = assert(io.open(arg[2], "wb"))

local data = inp:read("a")
data = string.gsub(data, "\r\n", "\n")
out:write(data)

assert(out:close())
```

It cannot use the standard I/O streams (`stdin`/`stdout`), because these streams are open in text mode. Instead, it assumes that the names of the input file and the output file are arguments to the program. We can call this program with the following command line:

```
> lua prog.lua file.dos file.unix
```

As another example, the following program prints all strings found in a binary file:

```
local f = assert(io.open(arg[1], "rb"))
local data = f:read("a")
local validchars = "[%g%s]"
local pattern = "(" .. string.rep(validchars, 6) .. "+)\0"
for w in string.gmatch(data, pattern) do
  print(w)
end
```

The program assumes that a string is any zero-terminated sequence of six or more valid characters, where a valid character is any character accepted by the pattern `validchars`. In our example, this pattern comprises the printable characters. We use `string.rep` and concatenation to create a pattern that matches all sequences of six or more `validchars` ended by a zero. The parentheses in the pattern capture the string without the zero.

Our last example is a program to make a dump of a binary file, showing its contents in hexadecimal. Figure 13.2, “Dumping the `dump` program” shows the result of applying this program to itself on a POSIX machine.

Figure 13.2. Dumping the `dump` program

```
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74  local f = assert
28 69 6F 2E 6F 70 65 6E 28 61 72 67 5B 31 5D 2C  (io.open(arg[1],
20 22 72 62 22 29 29 0A 6C 6F 63 61 6C 20 62 6C   "rb")).local bl
6F 63 6B 73 69 7A 65 20 3D 20 31 36 0A 66 6F 72  ocksize = 16.for
20 62 79 74 65 73 20 69 6E 20 66 3A 6C 69 6E 65   bytes in f:line
  ...
25 63 22 2C 20 22 2E 22 29 0A 20 20 69 6F 2E 77  %c", ".").  io.w
72 69 74 65 28 22 20 22 2C 20 62 79 74 65 73 2C  rite(" ", bytes,
20 22 5C 6E 22 29 0A 65 6E 64 0A 0A               "\n").end..
```

The complete program is here:

```
local f = assert(io.open(arg[1], "rb"))
```

```
local blocksize = 16
for bytes in f:lines(blocksize) do
  for i = 1, #bytes do
    local b = string.unpack("B", bytes, i)
    io.write(string.format("%02X ", b))
  end
  io.write(string.rep("   ", blocksize - #bytes))
  bytes = string.gsub(bytes, "%c", ".")
  io.write(" ", bytes, "\n")
end
```

Again, the first program argument is the input file name; the output is regular text, so it can go to the standard output. The program reads the file in chunks of 16 bytes. For each chunk, it writes the hexadecimal representation of each byte, and then it writes the chunk as text, changing control characters to dots. We use `string.rep` to fill with blanks the last line (which in general will not have exactly 16 bytes), keeping the alignment.

Exercises

Exercise 13.1: Write a function to compute the modulo operation for unsigned integers.

Exercise 13.2: Implement different ways to compute the number of bits in the representation of a Lua integer.

Exercise 13.3: How can you test whether a given integer is a power of two?

Exercise 13.4: Write a function to compute the Hamming weight of a given integer. (The *Hamming weight* of a number is the number of ones in its binary representation.)

Exercise 13.5: Write a function to test whether the binary representation of a given integer is a palindrome.

Exercise 13.6: Implement a *bit array* in Lua. It should support the following operations:

- `newBitArray(n)` (creates an array with `n` bits),
- `setBit(a, n, v)` (assigns the Boolean value `v` to bit `n` of array `a`),
- `testBit(a, n)` (returns a Boolean with the value of bit `n`).

Exercise 13.7: Suppose we have a binary file with a sequence of records, each one with the following format:

```
struct Record {
  int x;
  char[3] code;
  float value;
};
```

Write a program that reads that file and prints the sum of the `value` fields.

Chapter 14. Data Structures

Tables in Lua are not a data structure; they are *the* data structure. We can represent all structures that other languages offer —arrays, records, lists, queues, sets— with tables in Lua. Moreover, Lua tables implement all these structures efficiently.

In more traditional languages, such as C and Pascal, we implement most data structures with arrays and lists (where lists = records + pointers). Although we can implement arrays and lists using Lua tables (and sometimes we do this), tables are more powerful than arrays and lists; many algorithms are simplified to the point of triviality with the use of tables. For instance, we seldom write a search in Lua, because tables offer direct access to any type.

It takes a while to learn how to use tables efficiently. Here, we will see how to implement typical data structures with tables and cover some examples of their uses. We will start with arrays and lists, not because we need them for the other structures, but because most programmers are already familiar with them. (We have already seen the basics of this material in Chapter 5, *Tables*, but I will repeat it here for completeness.) Then we will continue with more advanced examples, such as sets, bags, and graphs.

Arrays

We implement arrays in Lua simply by indexing tables with integers. Therefore, arrays do not have a fixed size, but grow as needed. Usually, when we initialize the array we define its size indirectly. For instance, after the following code, any attempt to access a field outside the range 1–1000 will return nil, instead of zero:

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

The length operator (#) uses this fact to find the size of an array:

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

We can start an array at index zero, one, 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 customary in Lua to start arrays with index one. The Lua libraries adhere to this convention; so does the length operator. If our arrays do not start with one, we will not be able to use these facilities.

We can use a constructor to create and initialize arrays in a single expression:

```
squares = {1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Such constructors can be as large as we need. In Lua, it is not uncommon data-description files with constructors with a few million elements.

Matrices and Multi-Dimensional Arrays

There are two main ways to represent matrices in Lua. The first one is with a *jagged array* (an array of arrays), that is, a table wherein each element is another table. For instance, we can create a matrix of zeros with dimensions `N` by `M` with the following code:

```
local mt = {}          -- create the matrix
for i = 1, N do
  local row = {}       -- create a new row
  mt[i] = row
  for j = 1, M do
    row[j] = 0
  end
end
```

Because tables are objects in Lua, we have to create each row explicitly to build a matrix. On the one hand, this is certainly more verbose than simply declaring a matrix, as we do in C. On the other hand, it gives us more flexibility. For instance, we can create a triangular matrix by changing the inner loop in the previous example to `for j=1,i do ... end`. With this code, the triangular matrix uses only half the memory of the original one.

The second way to represent a matrix is by composing the two indices into a single one. Typically, we do this by multiplying the first index by a suitable constant and then adding the second index. With this approach, the following code would create our matrix of zeros with dimensions `N` by `M`:

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

Quite often, applications use a *sparse matrix*, a matrix wherein most elements are zero or nil. For instance, we can represent a graph by its adjacency matrix, which has the value *x* in position *(m,n)* when there is a connection with cost *x* between nodes *m* and *n*. When these nodes are not connected, the value in position *(m,n)* is nil. To represent a graph with ten thousand nodes, where each node has about five neighbors, we will need a matrix with a hundred million entries (a square matrix with 10000 columns and 10000 rows), but approximately only fifty thousand of them will not be nil (five non-nil columns for each row, corresponding to the five neighbors of each node). Many books on data structures discuss at length how to implement such sparse matrices without wasting 800 MB of memory, but we seldom need these techniques when programming in Lua. Because we represent arrays with tables, they are naturally sparse. With our first representation (tables of tables), we will need ten thousand tables, each one with about five elements, with a grand total of fifty thousand entries. With the second representation, we will have a single table, with fifty thousand entries in it. Whatever the representation, we need space only for the non-nil elements.

We cannot use the length operator over sparse matrices, because of the holes (nil values) between active entries. This is not a big loss; even if we could use it, we probably would not. For most operations, it would be quite inefficient to traverse all those empty entries. Instead, we can use `pairs` to traverse only the non-nil elements. As an example, let us see how to do matrix multiplication with sparse matrices represented by jagged arrays.

Suppose we want to multiply a matrix `a[M,K]` by a matrix `b[K,N]`, producing the matrix `c[M,N]`. The usual matrix-multiplication algorithm goes like this:

```
for i = 1, M do
  for j = 1, N do
    c[i][j] = 0
    for k = 1, K do
      c[i][j] = c[i][j] + a[i][k] * b[k][j]
    end
  end
end
```

The two outer loops traverse the entire resulting matrix, and for each element, the inner loop computes its value.

For sparse matrices with jagged arrays, this inner loop is a problem. Because it traverses a column of `b`, instead of a row, we cannot use something like `pairs` here: the loop has to visit each row looking whether that row has an element in that column. Instead of visiting only a few non-zero elements, the loop visits all zero elements, too. (Traversing a column can be an issue in other contexts, too, because of its loss of spatial locality.)

The following algorithm is quite similar to the previous one, but it reverses the order of the two inner loops. With this simple change, it avoids traversing columns:

```
-- assumes 'c' has zeros in all elements
for i = 1, M do
  for k = 1, K do
    for j = 1, N do
      c[i][j] = c[i][j] + a[i][k] * b[k][j]
    end
  end
end
```

Now, the middle loop traverses the row `a[i]`, and the inner loop traverses the row `b[k]`. Both can use `pairs`, visiting only the non-zero elements. The initialization of the resulting matrix `c` is not an issue here, because an empty sparse matrix is naturally filled with zeros.

Figure 14.1. Multiplication of sparse matrices

```
function mult (a, b)
  local c = {}          -- resulting matrix
  for i = 1, #a do
    local resultline = {}               -- will be 'c[i]'
    for k, va in pairs(a[i]) do         -- 'va' is a[i][k]
      for j, vb in pairs(b[k]) do       -- 'vb' is b[k][j]
        local res = (resultline[j] or 0) + va * vb
        resultline[j] = (res ~= 0) and res or nil
      end
    end
    c[i] = resultline
  end
  return c
end
```

Figure 14.1, “Multiplication of sparse matrices” shows the complete implementation of the above algorithm, using `pairs` and taking care of sparse entries. This implementation visits only the non-nil elements, and the result is naturally sparse. Moreover, the code deletes resulting entries that by chance evaluate to zero.

Linked Lists

Because tables are dynamic entities, it is easy to implement linked lists in Lua. We represent each node with a table (what else?); links are simply table fields that contain references to other tables. For instance, let us implement a singly-linked list, where each node has two fields, `value` and `next`. A simple variable is the list root:

```
list = nil
```

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

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

To traverse the list, we write this:

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

We can also implement easily other kinds of lists, such as doubly-linked lists or circular lists. However, we seldom need those structures in Lua, because usually there is a simpler way to represent our data without using linked lists. For instance, we can represent a stack with an (unbounded) array.

Queues and Double-Ended Queues

A simple way to implement queues in Lua is with functions `insert` and `remove` from the `table` library. As we saw in the section called “The Table Library”, these functions insert and remove elements in any position of an array, moving other elements to accommodate the operation. However, these moves can be expensive for large structures. A more efficient implementation uses two indices, one for the first element and another for the last. With that representation, we can insert or remove an element at both ends in constant time, as shown in Figure 14.2, “A double-ended queue”.

Figure 14.2. A double-ended queue

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

function pushFirst (list, value)
  local first = list.first - 1
  list.first = first
  list[first] = value
end

function pushLast (list, value)
  local last = list.last + 1
  list.last = last
  list[last] = value
end

function popFirst (list)
  local first = list.first
  if first > list.last then error("list is empty") end
  local value = list[first]
  list[first] = nil        -- to allow garbage collection
  list.first = first + 1
  return value
end

function popLast (list)
  local last = list.last
  if list.first > last then error("list is empty") end
  local value = list[last]
  list[last] = nil         -- to allow garbage collection
  list.last = last - 1
  return value
end
```

If we use this structure in a strict queue discipline, calling only `pushLast` and `popFirst`, both `first` and `last` will increase continually. However, because we represent arrays in Lua with tables, we can index them either from 1 to 20 or from 16777201 to 16777220. With 64-bit integers, such a queue can run for thirty thousand years, doing ten million insertions per second, before it has problems with overflows.

Reverse Tables

As I said, before, we seldom do searches in Lua. Instead, we use what we call an index table or a reverse table.

Suppose we have a table with the names of the days of the week:

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

Now we want to translate a name into its position in the week. We can search the table, looking for the given name. A more efficient approach, however, is to build a reverse table, say `revDays`, which has the names as indices and the numbers as values. This table would look like this:

```
revDays = {["Sunday"] = 1,   ["Monday"] = 2,
           ["Tuesday"] = 3,  ["Wednesday"] = 4,
           ["Thursday"] = 5, ["Friday"] = 6,
           ["Saturday"] = 7}
```

Then, all we have to do to find the order of a name is to index this reverse table:

```
x = "Tuesday"
print(revDays[x])    --> 3
```

Of course, we do not need to declare the reverse table manually. We can build it automatically from the original one:

```
revDays = {}
for k,v in pairs(days) do
  revDays[v] = k
end
```

The loop will do the assignment for each element of `days`, with the variable `k` getting the keys (1, 2, ...) and `v` the values (`"Sunday"`, `"Monday"`, ...).

Sets and Bags

Suppose we want to list all identifiers used in a program source; for that, we need to filter the reserved words out of our listing. Some C programmers could be tempted to represent the set of reserved words as an array of strings and search this array to know whether a given word is in the set. To speed up the search, they could even use a binary tree to represent the set.

In Lua, an efficient and simple way to represent such sets is to put the set elements as *indices* in a table. Then, instead of searching the table for a given element, we just index the table and test whether the result is nil. In our example, we could write the following code:

```
reserved = {
  ["while"] = true,     ["if"] = true,
  ["else"] = true,      ["do"] = true,
}

for w in string.gmatch(s, "[%a_][%w_]*") do
  if not reserved[w] then
    do something with 'w'   -- 'w' is not a reserved word
  end
end
```

(In the definition of `reserved`, we cannot write `while = true`, because `while` is not a valid name in Lua. Instead, we use the notation `["while"] = true`.)

We can have a clearer initialization using an auxiliary function to build 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", }
```

We can also use another set to collect the identifiers:

```
local ids = {}
for w in string.gmatch(s, "[%a_][%w_]*") do
  if not reserved[w] then
    ids[w] = true
  end
end

-- print each identifier once
for w in pairs(ids) do print(w) end
```

Bags, also called *multisets*, differ from regular sets in that each element can appear multiple times. An easy representation for bags in Lua is similar to the previous representation for sets, but with a counter associated with each key.[1] To insert an element, we increment its 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 only keep the counter if it already exists and it is still greater than zero.

String Buffers

Suppose we are building a string piecemeal, for instance reading a file line by line. Our typical code could look like this:

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

Despite its innocent look, this code in Lua can cause a huge performance penalty for large files: for instance, it takes more than 30 seconds to read a 4.5 MB file on my new machine.

Why is that? To understand what happens, let us imagine that we are in the middle of the read loop; each line has 20 bytes and we have already read some 2500 lines, so `buff` is a string with 50 kB. When Lua concatenates `buff..line.."\n"`, it allocates a new string with 50020 bytes and copies the 50000 bytes from `buff` into this new string. That is, for each new line, Lua moves around 50 kB of memory, and growing. The algorithm is quadratic. After reading 100 new lines (only 2 kB), Lua has already moved more than 5 MB of memory. When Lua finishes reading 350 kB, it has moved around more than 50 GB. (This problem is not peculiar to Lua: other languages wherein strings are immutable values present a similar behavior, Java being a famous example.)

Before we continue, we should remark that, despite all I said, this situation is not a common problem. For small strings, the above loop is fine. To read an entire file, Lua provides the `io.read("a")` option,

[1] We already used this representation for the most-frequent-words program in Chapter 11, *Interlude: Most Frequent Words*.

which reads the file at once. However, sometimes we must face this problem. Java offers the `StringBuffer` class to ameliorate the problem. In Lua, we can use a table as the string buffer. The key to this approach is the function `table.concat`, which returns the concatenation of all the strings of a given list. Using `concat`, we can write our previous loop as follows:

```
local t = {}
for line in io.lines() do
  t[#t + 1] = line .. "\n"
end
local s = table.concat(t)
```

This algorithm takes less than 0.05 seconds to read the same file that took more than half a minute to read with the original code. (Nevertheless, for reading a whole file it is still better to use `io.read` with the `"a"` option.)

We can do even better. The function `concat` accepts an optional second argument, which is a separator to be inserted between the strings. Using this separator, we do not need to insert a newline after each line:

```
local t = {}
for line in io.lines() do
  t[#t + 1] = line
end
s = table.concat(t, "\n") .. "\n"
```

The function inserts the separator between the strings, but we still have to add the last newline. This last concatenation creates a new copy of the resulting string, which can be quite long. There is no option to make `concat` insert this extra separator, but we can deceive it, inserting an extra empty string in `t`:

```
t[#t + 1] = ""
s = table.concat(t, "\n")
```

Now, the extra newline that `concat` adds before this empty string is at the end of the resulting string, as we wanted.

Graphs

Like any decent language, Lua allows multiple implementations for graphs, each one better adapted to some particular algorithms. Here we will see a simple object-oriented implementation, where we represent nodes as objects (actually tables, of course) and arcs as references between nodes.

We will represent each node as a table with two fields: `name`, with the node's name; and `adj`, with the set of nodes adjacent to this one. Because we will read the graph from a text file, we need a way to find a node given its name. So, we will use an extra table mapping names to nodes. Given a name, function `name2node` returns the corresponding node:

```
local function name2node (graph, name)
  local node = graph[name]
  if not node then
    -- node does not exist; create a new one
    node = {name = name, adj = {}}
    graph[name] = node
  end
  return node
end
```

Figure 14.3, “Reading a graph from a file” shows the function that builds a graph.

Figure 14.3. Reading a graph from a file

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

It reads a file where each line has two node names, meaning that there is an arc from the first node to the second. For each line, the function uses `string.match` to split the line in two names, finds the nodes corresponding to these names (creating the nodes if needed), and connects the nodes.

Figure 14.4, “Finding a path between two nodes” illustrates an algorithm using such graphs.

Figure 14.4. Finding a path between two nodes

```
function findpath (curr, to, path, visited)
  path = path or {}
  visited = visited or {}
  if visited[curr] then     -- node already visited?
    return nil              -- no path here
  end
  visited[curr] = true      -- mark node as visited
  path[#path + 1] = curr    -- add it to path
  if curr == to then        -- final node?
    return path
  end
  -- try all adjacent nodes
  for node in pairs(curr.adj) do
    local p = findpath(node, to, path, visited)
    if p then return p end
  end
  table.remove(path)        -- remove node from path
end
```

The function `findpath` searches for a path between two nodes using a depth-first traversal. Its first parameter is the current node; the second is its goal; the third parameter keeps the path from the origin to the current node; the last parameter is a set with all the nodes already visited, to avoid loops. Note how the algorithm manipulates nodes directly, without using their names. For instance, `visited` is a set of nodes, not of node names. Similarly, `path` is a list of nodes.

To test this code, we add a function to print a path and some code to put it all to 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 14.1: Write a function to add two sparse matrices.

Exercise 14.2: Modify the queue implementation in Figure 14.2, “A double-ended queue” so that both indices return to zero when the queue is empty.

Exercise 14.3: Modify the graph structure so that it can keep a label for each arc. The structure should represent each arc by an object, too, with two fields: its label and the node it points to. Instead of an adjacent set, each node keeps an incident set that contains the arcs that originate at that node.

Adapt the function `readgraph` to read two node names plus a label from each line in the input file. (Assume that the label is a number.)

Exercise 14.4: Assume the graph representation of the previous exercise, where the label of each arc represents the distance between its end nodes. Write a function to find the shortest path between two given nodes, using Dijkstra's algorithm.

Chapter 15. Data Files and Serialization

When dealing with data files, it is usually much easier to write the data than to read it back. When we write a file, we have full control of what is going on. When we read a file, on the other hand, we do not know what to expect. Besides all the kinds of data that a correct file can contain, a robust program should also handle bad files gracefully. Therefore, coding robust input routines is always difficult. In this chapter, we will see how we can use Lua to eliminate all code for reading data from our programs, simply by writing the data in an appropriate format. More specifically, we write data as Lua programs that, when run, rebuild the data.

Data description has been one of the main applications of Lua since its creation in 1993. At that time, the main alternative for a textual data-description language would be SGML. For many people (including us), SGML is bloated and complex. In 1998, some people simplified it to create XML, which in our view is still bloated and complex. Other people shared our view, and some of them created JSON (in 2001). JSON is based on Javascript and quite similar to restricted Lua data files. On the one hand, JSON has a big advantage of being an international standard, and several languages (including Lua) have libraries to manipulate JSON files. On the other hand, Lua files are trivial to read and more flexible.

Using a full programming language for data description is surely flexible, but it brings two problems. One is security, as "data" files can run amok inside our program. We can solve that by running the file in a sandbox, which we will discuss in the section called "Sandboxing".

The other problem is performance. Lua not only runs fast, but it also compiles fast. For instance, in my new machine, Lua 5.3 reads, compiles, and runs a program with ten million assignments in four seconds, using 240 MB. For comparison, Perl 5.18 takes 21 seconds and 6 GB, Python 2.7 and Python 3.4 trash the machine, Node.js 0.10.25 gives an "out of memory" error after eight seconds, and Rhino 1.7 also gives an "out of memory" error, after six minutes.

Data Files

Table constructors provide an interesting alternative for file formats. With a little extra work when writing data, reading becomes trivial. The technique is to write our data file as Lua code that, when run, rebuilds the data into the program. With table constructors, these chunks can look remarkably like a plain data file.

Let us see an example to make things clear. If our data file is in a predefined format, such as CSV (Comma-Separated Values) or XML, we have little choice. However, if we are going to create the file for our own use, we can use Lua constructors as our format. In this format, we represent each data record as a Lua constructor. Instead of writing in our data file something like

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

we write this:

```
Entry{"Donald E. Knuth",
      "Literate Programming",
      "CSLI",
      1992}

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

Remember that `Entry{`*`code`*`}` is the same as `Entry({`*`code`*`})`, that is, a call to some function `Entry` with a table as its single argument. So, that previous piece of data is a Lua program. To read that file, we

only need to run it, with a sensible definition for `Entry`. For instance, the following program counts the number of entries in a data file:

```
local count = 0
function Entry () count = count + 1 end
dofile("data")
print("number of entries: " .. count)
```

The next program collects in a set the names of all authors found in the file, and then prints them:

```
local authors = {}      -- a set to collect authors
function Entry (b) authors[b[1]] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

Note the event-driven approach in these program fragments: the function `Entry` acts as a callback function, which is called during the `dofile` for each entry in the data file.

When file size is not a big concern, we can use name-value pairs for our representation:[1]

```
Entry{
  author = "Donald E. Knuth",
  title = "Literate Programming",
  publisher = "CSLI",
  year = 1992
}

Entry{
  author = "Jon Bentley",
  title = "More Programming Pearls",
  year = 1990,
  publisher = "Addison-Wesley",
}
```

This format is what we call a *self-describing data* format, because each piece of data has attached to it a short description of its meaning. Self-describing data are more readable (by humans, at least) than CSV or other compact notations; they are easy to edit by hand, when necessary; and they allow us to make small modifications in the basic format without having to change the data file. For instance, if we add a new field we need only a small change in the reading program, so that it supplies a default value when the field is absent.

With the name-value format, our program to collect authors becomes this:

```
local authors = {}      -- a set to collect authors
function Entry (b) authors[b.author] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

Now the order of fields is irrelevant. Even if some entries do not have an author, we have to adapt only the function `Entry`:

```
function Entry (b)
  authors[b.author or "unknown"] = true
end
```

[1] If this format reminds you of BibTeX, it is not a coincidence. BibTeX was one of the inspirations for the constructor syntax in Lua.

Serialization

Frequently we need to serialize some data, that is, to convert the data into a stream of bytes or characters, so that we can save it into a file or send it through a network connection. We can represent serialized data as Lua code in such a way that, when we run the code, it reconstructs the saved values into the reading program.

Usually, if we want to restore the value of a global variable, our chunk will be something like `varname = exp`, where *exp* is the Lua code to create the value. The `varname` is the easy part, so let us see how to write the code that creates a value. For a numeric value, the task is easy:

```
function serialize (o)
  if type(o) == "number" then
    io.write(tostring(o))
  else other cases
  end
end
```

By writing a float in decimal format, however, we risk losing some precision. We can use a hexadecimal format to avoid this problem. With format (`"%a"`), the read float will have exactly the same bits of the original one. Moreover, since Lua 5.3 we should distinguish between integers and floats, so that they can be restored with the correct subtype:

```
local fmt = {integer = "%d", float = "%a"}

function serialize (o)
  if type(o) == "number" then
    io.write(string.format(fmt[math.type(o)], o))
  else other cases
```

For a string value, a naive approach would be something like this:

```
if type(o) == "string" then
  io.write("'", o, "'")
```

However, if the string contains special characters (such as quotes or newlines) the resulting code will not be a valid Lua program.

You may be tempted to solve this problem changing quotes:

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

Beware of code injection! If a malicious user manages to direct your program to save something like `" ]]..os.execute('rm *')..[[ "` (for instance, she can supply this string as her address), your final chunk will be like this one:

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

You will have a bad surprise trying to load this “data”.

A simple way to quote a string in a secure way is with the option `"%q"` from `string.format`. This option was designed to save the string in a way that it can be safely read back by Lua. It surrounds the string with double quotes and properly escapes double quotes, newlines, and some other characters inside the string:

```
a = 'a "problematic" \\string'
```

```
print(string.format("%q", a))    --> "a \"problematic\" \\string"
```

Using this feature, our `serialize` function now looks like this:

```
function serialize (o)
  if type(o) == "number" then
    io.write(string.format(fmt[math.type(o)], o))
  elseif type(o) == "string" then
    io.write(string.format("%q", o))
  else other cases
  end
end
```

Lua 5.3.3 extended the format option `"%q"` to work also with numbers (plus nil and Booleans), again writing them in a proper way to be read back by Lua. (In particular, it formats floats in hexadecimal, to ensure full precision.) Thus, since that version, we can simplify and extend `serialize` even more:

```
function serialize (o)
  local t = type(o)
  if t == "number" or t == "string" or t == "boolean" or
     t == "nil" then
    io.write(string.format("%q", o))
  else other cases
  end
end
```

Another way to save strings is the notation `[=[...]=]` for long strings. However, this notation is mainly intended for hand-written code, where we do not want to change a literal string in any way. In automatically generated code, it is easier to escape problematic characters, as the option `"%q"` from `string.format` does.

If you nevertheless want to use the long-string notation for automatically generated code, you must take care of some details. The first one is that you must choose a proper number of equals signs. A good proper number is one more than the maximum that appears in the original string. Because strings containing long sequences of equals signs are common (e.g., comments delimiting parts of a source code), we should limit our attention to sequences of equals signs enclosed by square bracket. The second detail is that Lua always ignores a newline at the beginning of a long string; a simple way to avoid this problem is to add always a newline to be ignored.

The function `quote` in Figure 15.1, “Quoting arbitrary literal strings” is the result of our previous remarks.

Figure 15.1. Quoting arbitrary literal strings

```
function quote (s)
  -- find maximum length of sequences of equals signs
  local n = -1
  for w in string.gmatch(s, "]=*%f[%]]") do
    n = math.max(n, #w - 1)   -- -1 to remove the ']'
  end

  -- produce a string with 'n' plus one equals signs
  local eq = string.rep("=", n + 1)

  -- build quoted string
  return string.format(" [%s[\n%s]%s] ", eq, s, eq)
end
```

It takes an arbitrary string and returns it formatted as a long string. The call to `gmatch` creates an iterator to traverse all occurrences of the pattern `']=*%f[%]]'` (that is, a closing square bracket followed by a sequence of zero or more equals signs followed by a frontier with a closing square bracket) in the string `s`. For each occurrence, the loop updates `n` with the maximum number of equals signs so far. After the loop, we use `string.rep` to replicate an equals sign `n + 1` times, which is one more than the maximum occurring in the string. Finally, `string.format` encloses `s` with pairs of brackets with the correct number of equals signs in between and adds extra spaces around the quoted string plus a newline at the beginning of the enclosed string.

(We might be tempted to use the simpler pattern `']=*]'`, which does not use a frontier pattern for the second square bracket, but there is a subtlety here. Suppose the subject is `"]=]==]"`. The first match is `"]=]"`. After it, what is left in the string is `"==]"`, and so there is no other match; in the end of the loop, `n` would be one instead of two. The frontier pattern does not consume the bracket, so that it remains in the subject for the following matches.)

Saving tables without cycles

Our next (and harder) task is to save tables. There are several ways to save them, according to what assumptions we make about the table structure. No single algorithm seems appropriate for all cases. Simple tables not only can use simpler algorithms, but also the output can be shorter and clearer.

Our first attempt is in Figure 15.2, “Serializing tables without cycles”.

Figure 15.2. Serializing tables without cycles

```
function serialize (o)
  local t = type(o)
  if t == "number" or t == "string" or t == "boolean" or
     t == "nil" then
    io.write(string.format("%q", o))
  elseif t == "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
```

Despite its simplicity, that function does a reasonable job. It even handles nested tables (that is, tables within other tables), as long as the table structure is a tree —that is, there are no shared subtables and no cycles. (A small aesthetic improvement would be to indent nested tables; see Exercise 15.1.)

The previous function assumes that all keys in a table are valid identifiers. If a table has numeric keys, or string keys that are not syntactic valid Lua identifiers, we are in trouble. A simple way to solve this difficulty is to use the following code to write each key:

```
io.write(string.format(" [%s] = ", serialize(k)))
```

With this change, we improve the robustness of our function, at the cost of the aesthetics of the resulting file. Consider the next call:

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

The result of this call using the first version of `serialize` is this:

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

Compare it with the result using the second version:

```
{
  ["a"] = 12,
  ["b"] = "Lua",
  ["key"] = "another \"one\"",
}
```

We can have both robustness and aesthetics by testing for each case whether it needs the square brackets; again, we will leave this improvement as an exercise.

Saving tables with cycles

To handle tables with generic topology (i.e., with cycles and shared subtables) we need a different approach. Constructors cannot create such tables, so we will not use them. To represent cycles we need names, so our next function will get as arguments the value to be saved plus its name. Moreover, we must keep track of the names of the tables already saved, to reuse them when we detect a cycle. We will use an extra table for this tracking. This table will have previously saved tables as indices and their names as the associated values.

The resulting code is in Figure 15.3, “Saving tables with cycles”.

Figure 15.3. Saving tables with cycles

```
function basicSerialize (o)
  -- assume 'o' is a number or a string
  return string.format("%q", o)
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             -- value already saved?
      io.write(saved[value], "\n")   -- use its previous name
    else
      saved[value] = name            -- save 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
```

We keep the restriction that the tables we want to save have only strings and numbers as keys. The function `basicSerialize` serializes these basic types, returning the result. The next function, `save`, does the hard work. The `saved` parameter is the table that keeps track of tables already saved. As an example, suppose we build a table like this:

```
a = {x=1, y=2; {3,4,5}}
a[2] = a    -- cycle
a.z = a[1]  -- shared subtable
```

The call `save("a", a)` will save it as follows:

```
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 actual order of these assignments may vary, as it depends on a table traversal. Nevertheless, the algorithm ensures that any node needed in a new definition is already defined.

If we want to save several values with shared parts, we can make the calls to `save` them using the same `saved` table. For instance, assume the following two tables:

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

If we save them independently, the result will not have common parts. However, if we use the same `saved` table for both calls to `save`, then the result will share 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 is usual in Lua, there are several other alternatives. Among them, we can save a value without giving it a global name (instead, the chunk builds a local value and returns it), we can use the list syntax when possible (see the exercises for this chapter), and so on. Lua gives you the tools; you build the mechanisms.

Exercises

Exercise 15.1: Modify the code in Figure 15.2, “Serializing tables without cycles” so that it indents nested tables. (Hint: add an extra parameter to `serialize` with the indentation string.)

Exercise 15.2: Modify the code of the previous exercise so that it uses the syntax `["`*`key`*`"]=`*`value`*, as suggested in the section called “Saving tables without cycles”.

Exercise 15.3: Modify the code of the previous exercise so that it uses the syntax `["`*`key`*`"]=`*`value` only when necessary (that is, when the key is a string but not a valid identifier).

Exercise 15.4: Modify the code of the previous exercise so that it uses the constructor syntax for lists whenever possible. For instance, it should serialize the table `{14, 15, 19}` as `{14, 15, 19}`, not as `{[1] = 14, [2] = 15, [3] = 19}`. (Hint: start by saving the values of the keys 1, 2, ..., as long as they are not nil. Take care not to save them again when traversing the rest of the table.)

Exercise 15.5: The approach of avoiding constructors when saving tables with cycles is too radical. It is possible to save the table in a more pleasant format using constructors for the simple case, and to use assignments later only to fix sharing and loops. Reimplement the function `save` (Figure 15.3, “Saving tables with cycles”) using this approach. Add to it all the goodies that you have implemented in the previous exercises (indentation, record syntax, and list syntax).

Chapter 16. Compilation, Execution, and Errors

Although we refer to Lua as an interpreted language, Lua always precompiles source code to an intermediate form before running it. (This is not a big deal: many interpreted languages do the same.) The presence of a compilation phase may sound out of place in an interpreted language. However, the distinguishing feature of interpreted languages is not that they are not compiled, but that it is possible (and easy) to execute code generated on the fly. We may say that the presence of a function like `dofile` is what entitles us to call Lua an interpreted language.

In this chapter, we will discuss in more details the process that Lua uses for running its chunks, what compilation means (and does), how Lua runs that compiled code, and how it handles errors in that process.

Compilation

Previously, we introduced `dofile` as a kind of primitive operation to run chunks of Lua code, but `dofile` is actually an auxiliary function: the function `loadfile` does the hard work. Like `dofile`, `loadfile` loads a Lua chunk from a file, but it does not run the chunk. Instead, it only compiles the chunk and returns the compiled chunk as a function. Moreover, unlike `dofile`, `loadfile` does not raise errors, but instead returns error codes. We could define `dofile` as follows:

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

Note the use of `assert` to raise an error if `loadfile` fails.

For simple tasks, `dofile` is handy, because it does the complete job in one call. However, `loadfile` is more flexible. In case of error, `loadfile` returns nil plus the error message, which allows us to handle the error in customized ways. Moreover, if we need to run a file several times, we can call `loadfile` once and call its result several times. This approach is much cheaper than several calls to `dofile`, because it compiles the file only once. (Compilation is a somewhat expensive operation when compared to other tasks in the language.)

The function `load` is similar to `loadfile`, except that it reads its chunk from a string or from a function, not from a file.[1] For instance, consider the next line:

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

After this code, `f` will be a function that executes `i = i + 1` when invoked:

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

The function `load` is powerful; we should use it with care. It is also an expensive function (when compared to some alternatives) and can result in incomprehensible code. Before you use it, make sure that there is no simpler way to solve the problem at hand.

[1] In Lua 5.1, function `loadstring` did the role of `load` for strings.

If we want to do a quick-and-dirty `dostring` (i.e., to load and run a chunk), we can call the result from `load` directly:

```
load(s)()
```

However, if there is any syntax error, `load` will return nil and the final error message will be something like *"attempt to call a nil value"*. For clearer error messages, it is better to use `assert`:

```
assert(load(s))()
```

Usually, it does not make sense to use `load` on a literal string. For instance, the next two lines are roughly equivalent:

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

f = function () i = i + 1 end
```

However, the second line is much faster, because Lua compiles the function together with its enclosing chunk. In the first line, the call to `load` involves a separate compilation.

Because `load` does not compile with lexical scoping, the two lines in the previous example may not be truly equivalent. To see the difference, let us change the example a little:

```
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 function `g` manipulates the local `i`, as expected, but `f` manipulates a global `i`, because `load` always compiles its chunks in the global environment.

The most typical use of `load` is to run external code (that is, pieces of code that come from outside our program) or dynamically-generated code. For instance, we may want to plot a function defined by the user; the user enters the function code and then we use `load` to evaluate it. Note that `load` expects a chunk, that is, statements. If we want to evaluate an expression, we can prefix the expression with **return**, so that we get a statement that returns the value of the given expression. See the example:

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

Because the function returned by `load` is a regular function, we can call it several times:

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

We can call `load` also with a *reader function* as its first argument. A reader function can return the chunk in parts; `load` calls the reader successively until it returns nil, which signals the chunk's end. As an example, the next call is equivalent to `loadfile`:

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

As we saw in Chapter 7, *The External World*, the call `io.lines(filename, "*L")` returns a function that, at each call, returns a new line from the given file. So, `load` will read the chunk from the file line by line. The following version is similar, but slightly more efficient:

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

Here, the iterator returned by `io.lines` reads the file in blocks of 1024 bytes.

Lua treats any independent chunk as the body of an anonymous variadic function. For instance, `load("a = 1")` returns the equivalent of the following expression:

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

Like any other function, chunks can declare local variables:

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

Using these features, we can rewrite our plot example to avoid the use of a global variable `x`:

```
print "enter function to be plotted (with variable 'x'):"
local line = io.read()
local f = assert(load("local x = ...; return " .. line))
for i = 1, 20 do
  print(string.rep("*", f(i)))
end
```

In this code, we append the declaration `"local x = ..."` at the beginning of the chunk to declare `x` as a local variable. We then call `f` with an argument `i` that becomes the value of the vararg expression (`...`).

The functions `load` and `loadfile` never raise errors. In case of any kind of error, they return nil plus an error message:

```
print(load("i i"))
  --> nil     [string "i i"]:1: '=' expected near 'i'
```

Moreover, these functions never have any kind of side effect, that is, they do not change or create variables, do not write to files, etc. They only compile the chunk to an internal representation and return the result as an anonymous function. A common mistake is to assume that loading a chunk defines functions. In Lua, function definitions are assignments; as such, they happen at runtime, not at compile time. For instance, suppose we have a file `foo.lua` like this:

```
-- file 'foo.lua'
function foo (x)
  print(x)
end
```

We then run the command

```
f = loadfile("foo.lua")
```

This command compiles `foo` but does not define it. To define it, we must run the chunk:

```
f = loadfile("foo.lua")
print(foo)    --> nil
f()           -- run the chunk
foo("ok")     --> ok
```

This behavior may sound strange, but it becomes clear if we rewrite the file without the syntax sugar:

```
-- file 'foo.lua'
foo = function (x)
  print(x)
end
```

In a production-quality program that needs to run external code, we should handle any errors reported when loading a chunk. Moreover, we may want to run the new chunk in a protected environment, to avoid unpleasant side effects. We will discuss environments in detail in Chapter 22, *The Environment*.

Precompiled Code

As I mentioned in the beginning of this chapter, Lua precompiles source code before running it. Lua also allows us to distribute code in precompiled form.

The simplest way to produce a precompiled file —also called a *binary chunk* in Lua jargon— is with the `luac` program that comes in the standard distribution. For instance, the next call creates a new file `prog.lc` with a precompiled version of a file `prog.lua`:

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

The Lua interpreter can execute this new file just like normal Lua code, performing exactly as it would with the original source:

```
$ lua prog.lc
```

Lua accepts precompiled code mostly anywhere it accepts source code. In particular, both `loadfile` and `load` accept precompiled code.

We can write a minimal `luac` directly in Lua:

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

The key function here is `string.dump`: it receives a Lua function and returns its precompiled code as a string, properly formatted to be loaded back by Lua.

The `luac` program offers some other interesting options. In particular, option `-l` lists the opcodes that the compiler generates for a given chunk. As an example, Figure 16.1, “Example of output from `luac -l`” shows the output of `luac` with option `-l` on the following one-line file:

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

Figure 16.1. Example of output from `luac -l`

```
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
```

(We will not discuss the internals of Lua in this book; if you are interested in more details about those opcodes, a Web search for `"lua opcode"` should give you relevant material.)

Code in precompiled form is not always smaller than the original, but it loads faster. Another benefit is that it gives a protection against accidental changes in sources. Unlike source code, however, maliciously corrupted binary code can crash the Lua interpreter or even execute user-provided machine code. When running usual code, there is nothing to worry about. However, you should avoid running untrusted code in precompiled form. The function `load` has an option exactly for this task.

Besides its required first argument, `load` has three more arguments, all of them optional. The second is a name for the chunk, used only in error messages. The fourth argument is an environment, which we will discuss in Chapter 22, *The Environment*. The third argument is the one we are interested here; it controls what kinds of chunks can be loaded. If present, this argument must be a string: the string `"t"` allows only textual (normal) chunks; `"b"` allows only binary (precompiled) chunks; `"bt"`, the default, allows both formats.

Errors

Errare humanum est. Therefore, we must handle errors the best way we can. Because Lua is an extension language, frequently embedded in an application, it cannot simply crash or exit when an error happens. Instead, whenever an error occurs, Lua must offer ways to handle it.

Any unexpected condition that Lua encounters raises an error. Errors occur when a program tries to add values that are not numbers, call values that are not functions, index values that are not tables, and so on. (We can modify this behavior using *metatables*, as we will see later.) We can also explicitly raise an error calling the function `error`, with an error message as an argument. Usually, this function is the appropriate way to signal errors in our code:

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

This construction of calling `error` subject to some condition is so common that Lua has a built-in function just for this job, called `assert`:

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

The function `assert` checks whether its first argument is not false and simply returns this argument; if the argument is false, `assert` raises an error. Its second argument, the message, is optional. Beware, however, that `assert` is a regular function. As such, Lua always evaluates its arguments before calling the function. If we write something like

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

Lua will always do the concatenation, even when `n` is a number. It may be wiser to use an explicit test in such cases.

When a function finds an unexpected situation (an *exception*), it can assume two basic behaviors: it can return an error code (typically nil or **false**) or it can raise an error, calling `error`. There are no fixed rules for choosing between these two options, but I use the following guideline: an exception that is easily avoided should raise an error; otherwise, it should return an error code.

For instance, let us consider `math.sin`. How should it behave when called on a table? Suppose it returns an error code. If we need to check for errors, we would have to write something like this:

```
local res = math.sin(x)
if not res then     -- error?
  error-handling code
```

However, we could as easily check this exception *before* calling the function:

```
if not tonumber(x) then     -- x is not a number?
  error-handling code
```

Frequently we check neither the argument nor the result of a call to `sin`; if the argument is not a number, it means that probably there is something wrong in our program. In such situations, the simplest and most practical way to handle the exception is to stop the computation and issue an error message.

On the other hand, let us consider `io.open`, which opens a file. How should it behave when asked to open a file that does not exist? In this case, there is no simple way to check for the exception before calling the function. In many systems, the only way of knowing whether a file exists is by trying to open it. Therefore, if `io.open` cannot open a file because of an external reason (such as “file does not exist” or “permission denied”), it returns false, plus a string with the error message. In this way, we have a chance to handle the situation in an appropriate way, for instance by asking the user for another file name:

```
local file, msg
repeat
  print "enter a file name:"
  local name = io.read()
  if not name then return end   -- no input
  file, msg = io.open(name, "r")
  if not file then print(msg) end
until file
```

If we do not want to handle such situations, but still want to play safe, we simply use `assert` to guard the operation:

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

This is a typical Lua idiom: if `io.open` fails, `assert` will raise an error. Notice how the error message, which is the second result from `io.open`, goes as the second argument to `assert`.

Error Handling and Exceptions

For many applications, we do not need to do any error handling in Lua; the application program does this handling. All Lua activities start from a call by the application, usually asking Lua to run a chunk. If there is any error, this call returns an error code, so that the application can take appropriate actions. In the case of the stand-alone interpreter, its main loop just prints the error message and continues showing the prompt and running the given commands.

However, if we want to handle errors inside the Lua code, we should use the function `pcall` (*protected call*) to encapsulate our code.

Suppose we want to run a piece of Lua code and to catch any error raised while running that code. Our first step is to encapsulate that piece of code in a function; more often than not, we use an anonymous function for that. Then, we call that function through `pcall`:

```
local ok, msg = pcall(function ()
     some code
     if unexpected_condition then error() end
     some code
     print(a[i])    -- potential error: 'a' may not be a table
     some code
   end)

if ok then    -- no errors while running protected code
  regular code
else   -- protected code raised an error: take appropriate action
  error-handling code
end
```

The function `pcall` calls its first argument in *protected mode*, so that it catches any errors while the function is running. The function `pcall` never raises any error, no matter what. If there are no errors, `pcall` returns **true**, plus any values returned by the call. Otherwise, it returns **false**, plus the error message.

Despite its name, the error message does not have to be a string; a better name is *error object*, because `pcall` will return any Lua value that we pass to `error`:

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

These mechanisms provide all we need to do exception handling in Lua. We throw an exception with `error` and catch it with `pcall`. The error message identifies the kind of error.

Error Messages and Tracebacks

Although we can use a value of any type as an error object, usually error objects are strings describing what went wrong. When there is an internal error (such as an attempt to index a non-table value), Lua generates the error object, which in that case is always a string; otherwise, the error object is the value passed to the function `error`. Whenever the object is a string, Lua tries to add some information about the location where the error happened:

```
local status, err = pcall(function () error("my error") end)
print(err)          --> stdin:1: my error
```

The location information gives the chunk's name (`stdin`, in the example) plus the line number (1, in the example).

The function `error` has an additional second parameter, which gives the *level* where it should report the error. We use this parameter to blame someone else for the error. For instance, suppose we write a function whose first task is to check whether it was called correctly:

```
function foo (str)
  if type(str) ~= "string" then
    error("string expected")
  end
  regular code
end
```

Then, someone calls this function with a wrong argument:

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

As it is, Lua points its finger to `foo` —after all, it was it who called `error`— and not to the real culprit, the caller. To correct this problem, we inform `error` that the error it is reporting occurred on level two in the calling hierarchy (level one is our own function):

```
function foo (str)
  if type(str) ~= "string" then
    error("string expected", 2)
  end
  regular code
end
```

Frequently, when an error happens, we want more debug information than only the location where the error occurred. At least, we want a traceback, showing the complete stack of calls leading to the error. When `pcall` returns its error message, it destroys part of the stack (the part that goes from it to the error point). Consequently, if we want a traceback, we must build it before `pcall` returns. To do this, Lua provides the function `xpcall`. It works like `pcall`, but its second argument is a *message handler function*. In case of error, Lua calls this message handler before the stack unwinds, so that it can use the debug library to gather any extra information it wants about the error. Two common message handlers are `debug.debug`, which gives us a Lua prompt so that we can inspect by ourselves what was going on when the error happened; and `debug.traceback`, which builds an extended error message with a traceback. The latter is the function that the stand-alone interpreter uses to build its error messages.

Exercises

Exercise 16.1: Frequently, it is useful to add some prefix to a chunk of code when loading it. (We saw an example previously in this chapter, where we prefixed a **return** to an expression being loaded.) Write a function `loadwithprefix` that works like `load`, except that it adds its extra first argument (a string) as a prefix to the chunk being loaded.

Like the original `load`, `loadwithprefix` should accept chunks represented both as strings and as reader functions. Even in the case that the original chunk is a string, `loadwithprefix` should not actually concatenate the prefix with the chunk. Instead, it should call `load` with a proper reader function that first returns the prefix and then returns the original chunk.

Exercise 16.2: Write a function `multiload` that generalizes `loadwithprefix` by receiving a list of readers, as in the following example:

```
f = multiload("local x = 10;",
              io.lines("temp", "*L"),
              " print(x)")
```

In the above example, `multiload` should load a chunk equivalent to the concatenation of the string `"local..."`, the contents of the `temp` file, and the string `"print(x)"`. Like `loadwithprefix`, from the previous exercise, `multiload` should not actually concatenate anything.

Exercise 16.3: The function `stringrep`, in Figure 16.2, “String repetition”, uses a binary multiplication algorithm to concatenate `n` copies of a given string `s`.

Figure 16.2. String repetition

```
function stringrep (s, n)
  local r = ""
  if n > 0 then
    while n > 1 do
      if n % 2 ~= 0 then  r = r .. s  end
      s = s .. s
      n = math.floor(n / 2)
    end
    r = r .. s
  end
  return r
end
```

For any fixed `n`, we can create a specialized version of `stringrep` by unrolling the loop into a sequence of instructions `r = r .. s` and `s = s .. s`. As an example, for `n = 5` the unrolling gives us 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, given `n`, returns a specialized function `stringrep_`*n*. Instead of using a closure, your function should build the text of a Lua function with the proper sequence of instructions (a mix of `r = r .. s` and `s = s .. s`) and then use `load` to produce the final function. Compare the performance of the generic function `stringrep` (or of a closure using it) with your tailor-made functions.

Exercise 16.4: Can you find any value for `f` such that the call `pcall(pcall, f)` returns **false** as its first result? Why is this relevant?

Chapter 17. Modules and Packages

Usually, Lua does not set policies. Instead, Lua provides mechanisms that are powerful enough for groups of developers to implement the policies that best suit them. However, this approach does not work well for modules. One of the main goals of a module system is to allow different groups to share code. The lack of a common policy impedes this sharing.

Starting in version 5.1, Lua has defined a set of policies for modules and packages (a package being a collection of modules). These policies do not demand any extra facility from the language; programmers can implement them using what we have seen so far. Programmers are free to use different policies. Of course, alternative implementations may lead to programs that cannot use foreign modules and modules that cannot be used by foreign programs.

From the point of view of the user, a *module* is some code (either in Lua or in C) that can be loaded through the function `require` and that creates and returns a table. Everything that the module exports, such as functions and constants, it defines inside this table, which works as a kind of namespace.

As an example, all standard libraries are modules. We can use the mathematical library like this:

```
local m = require "math"
print(m.sin(3.14))          --> 0.0015926529164868
```

However, the stand-alone interpreter preloads all standard libraries with code equivalent to this:

```
math = require "math"
string = require "string"
...
```

This preloading allows us to write the usual notation `math.sin`, without bothering to require the module `math`.

An obvious benefit of using tables to implement modules is that we can manipulate modules like any other table and use the whole power of Lua to create extra facilities. In most languages, modules are not first-class values (that is, they cannot be stored in variables, passed as arguments to functions, etc.); those languages need special mechanisms for each extra facility they want to offer for modules. In Lua, we get extra facilities for free.

For instance, there are several ways for a user to call a function from a module. The usual way is this:

```
local mod = require "mod"
mod.foo()
```

The user can set any local name for the module:

```
local m = require "mod"
m.foo()
```

She can also provide alternative names for individual functions:

```
local m = require "mod"
local f = m.foo
f()
```

She can also import only a specific function:

```
local f = require "mod".foo        -- (require("mod")).foo
```

```
f()
```

The nice thing about these facilities is that they involve no special support from Lua. They use what the language already offers.

The Function `require`

Despite its central role in the implementation of modules in Lua, `require` is a regular function, with no special privileges. To load a module, we simply call it with a single argument, the module name. Remember that, when the single argument to a function is a literal string, the parentheses are optional, and it is customary to omit them in regular uses of `require`. Nevertheless, the following uses are all correct, too:

```
local m = require('math')

local modname = 'math'
local m = require(modname)
```

The function `require` tries to keep to a minimum its assumptions about what a module is. For it, a module is just any code that defines some values, such as functions or tables containing functions. Typically, that code returns a table comprising the module functions. However, because this action is done by the module code, not by `require`, some modules may choose to return other values or even to have side effects (e.g., by creating global variables).

The first step of `require` is to check in the table `package.loaded` whether the module is already loaded. If so, `require` returns its corresponding value. Therefore, once a module is loaded, other calls requiring the same module simply return the same value, without running any code again.

If the module is not loaded yet, `require` searches for a Lua file with the module name. (This search is guided by the variable `package.path`, which we will discuss later.) If it finds such a file, it loads it with `loadfile`. The result is a function that we call a *loader*. (The loader is a function that, when called, loads the module.)

If `require` cannot find a Lua file with the module name, it searches for a C library with that name.[1] (In that case, the search is guided by the variable `package.cpath`.) If it finds a C library, it loads it with the low-level function `package.loadlib`, looking for a function called `luaopen_`*`modname`*. The loader in this case is the result of `loadlib`, which is the C function `luaopen_`*`modname`* represented as a Lua function.

No matter whether the module was found in a Lua file or a C library, `require` now has a loader for it. To finally load the module, `require` calls the loader with two arguments: the module name and the name of the file where it got the loader. (Most modules just ignore these arguments.) If the loader returns any value, `require` returns this value and stores it in the `package.loaded` table, to return the same value in future calls for this same module. If the loader returns no value, and the table entry `package.loaded[@rep{modname}]` is still empty, `require` behaves as if the module returned **true**. Without this correction, a subsequent call to `require` would run the module again.

To force `require` into loading the same module twice, we can erase the library entry from `package.loaded`:

```
package.loaded.modname = nil
```

The next time the module is required, `require` will do all its work again.

[1] In the section called “C Modules”, we will discuss how to write C libraries.

A common complaint against `require` is that it cannot pass arguments to the module being loaded. For instance, the mathematical module might have an option for choosing between degrees and radians:

```
-- bad code
local math = require("math", "degree")
```

The problem here is that one of the main goals of `require` is to avoid loading a module multiple times. Once a module is loaded, it will be reused by whatever part of the program that requires it again. There would be a conflict if the same module were required with different parameters. In case you really want your module to have parameters, it is better to create an explicit function to set them, like here:

```
local mod = require "mod"
mod.init(0, 0)
```

If the initialization function returns the module itself, we can write that code like this:

```
local mod = require "mod".init(0, 0)
```

In any case, remember that the module itself is loaded only once; it is up to it to handle conflicting initializations.

Renaming a module

Usually, we use modules with their original names, but sometimes we must rename a module to avoid name clashes. A typical situation is when we need to load different versions of the same module, for instance for testing. Lua modules do not have their names fixed internally, so usually it is enough to rename the `.lua` file. However, we cannot edit the object code of a C library to correct the name of its `luaopen_*` function. To allow for such renamings, `require` uses a small trick: if the module name contains a hyphen, `require` strips from the name its suffix after the hyphen when creating the `luaopen_*` function name. For instance, if a module is named `mod-v3.4`, `require` expects its open function to be named `luaopen_mod`, instead of `luaopen_mod-v3.4` (which would not be a valid C name anyway). So, if we need to use two modules (or two versions of the same module) named `mod`, we can rename one of them to `mod-v1`, for instance. When we call `m1 = require "mod-v1"`, `require` will find the renamed file `mod-v1` and, inside this file, the function with the original name `luaopen_mod`.

Path searching

When searching for a Lua file, the path that guides `require` is a little different from typical paths. A typical path is a list of directories wherein to search for a given file. However, ISO C (the abstract platform where Lua runs) does not have the concept of directories. Therefore, the path used by `require` is a list of *templates*, each of them specifying an alternative way to transform a module name (the argument to `require`) into a file name. More specifically, each template in the path is a file name containing optional question marks. For each template, `require` substitutes the module name for each question mark and checks whether there is a file with the resulting name; if not, it goes to the next template. The templates in a path are separated by semicolons, a character seldom used for file names in most operating systems. For instance, consider the following path:

```
?;?.lua;c:\windows\?;/usr/local/lua/?/?.lua
```

With this path, the call `require "sql"` will try to open the following Lua files:

```
sql
sql.lua
c:\windows\sql
/usr/local/lua/sql/sql.lua
```

The function `require` assumes only the semicolon (as the component separator) and the question mark; everything else, including directory separators and file extensions, is defined by the path itself.

The path that `require` uses to search for Lua files is always the current value of the variable `package.path`. When the module `package` is initialized, it sets this variable with the value of the environment variable `LUA_PATH_5_3`; if this environment variable is undefined, Lua tries the environment variable `LUA_PATH`. If both are unefined, Lua uses a compiled-defined default path.[2] When using the value of an environment variable, Lua substitutes the default path for any substring `";;"`. For instance, if we set `LUA_PATH_5_3` to `"mydir/?.lua;;"`, the final path will be the template `"mydir/?.lua"` followed by the default path.

The path used to search for a C library works exactly in the same way, but its value comes from the variable `package.cpath`, instead of `package.path`. Similarly, this variable gets its initial value from the environment variables `LUA_CPATH_5_3` or `LUA_CPATH`. A typical value for this path in POSIX is like this:

```
./?.so;/usr/local/lib/lua/5.2/?.so
```

Note that the path defines the file extension. The previous example uses `.so` for all templates; in Windows, a typical path would be more like this one:

```
.\?.dll;C:\Program Files\Lua502\dll\?.dll
```

The function `package.searchpath` encodes all those rules for searching libraries. It takes a module name and a path, and looks for a file following the rules described here. It returns either the name of the first file that exists or nil plus an error message describing all files it unsuccessfully tried to open, as in the next 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'
```

As an interesting exercise, in Figure 17.1, “A homemade `package.searchpath`” we implement a function similar to `package.searchpath`.

Figure 17.1. A homemade `package.searchpath`

```
function search (modname, path)
  modname = string.gsub(modname, "%.", "/")
  local msg = {}
  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
    else
      msg[#msg + 1] = string.format("\n\tno file '%s'", fname);
    end
  end
  return nil, table.concat(msg)       -- not found
end
```

[2] Since Lua 5.2, the stand-alone interpreter accepts the command-line option `-E` to prevent the use of those environment variables and force the default.

The first step is to substitute the directory separator, assumed to be a slash in this example, for any dots. (As we will see later, a dot has a special meaning in a module name.) Then the function loops over all components of the path, wherein each component is a maximum expansion of non-semicolon characters. For each component, the function substitutes the module name for the question marks to get the final file name, and then it checks whether there is such a file. If so, the function closes the file and returns its name. Otherwise, it stores the failed name for a possible error message. (Note the use of a string buffer to avoid creating useless long strings.) If no file is found, then it returns nil plus the final error message.

Searchers

In reality, `require` is a little more complex than we have described. The search for a Lua file and the search for a C library are just two instances of a more general concept of *searchers*. A searcher is simply a function that takes the module name and returns either a loader for that module or nil if it cannot find one.

The array `package.searchers` lists the searchers that `require` uses. When looking for a module, `require` calls each searcher in the list passing the module name, until one of them finds a loader for the module. If the list ends without a positive response, `require` raises an error.

The use of a list to drive the search for a module allows great flexibility to `require`. For instance, if we want to store modules compressed in zip files, we only need to provide a proper searcher function for that and add it to the list. In its default configuration, the searcher for Lua files and the searcher for C libraries that we described earlier are respectively the second and the third elements in the list. Before them, there is the preload searcher.

The *preload* searcher allows the definition of an arbitrary function to load a module. It uses a table, called `package.preload`, to map module names to loader functions. When searching for a module name, this searcher simply looks for the given name in the table. If it finds a function there, it returns this function as the module loader. Otherwise, it returns nil. This searcher provides a generic method to handle some non-conventional situations. For instance, a C library statically linked to Lua can register its `luaopen_` function into the `preload` table, so that it will be called only when (and if) the user requires that module. In this way, the program does not waste resources opening the module if it is not used.

The default content of `package.searchers` includes a fourth function that is relevant only for sub-modules. We will discuss it at the section called “Submodules and Packages”.

The Basic Approach for Writing Modules in Lua

The simplest way to create a module in Lua is really simple: we create a table, put all functions we want to export inside it, and return this table. Figure 17.2, “A simple module for complex numbers” illustrates this approach.

Figure 17.2. A simple module for complex numbers

```
local M = {}         -- the module

-- creates a new complex number
local function new (r, i)
  return {r=r, i=i}
end

M.new = new        -- add 'new' to the module

-- constant 'i'
M.i = new(0, 1)

function M.add (c1, c2)
  return new(c1.r + c2.r, c1.i + c2.i)
end

function M.sub (c1, c2)
  return new(c1.r - c2.r, c1.i - c2.i)
end

function M.mul (c1, c2)
  return new(c1.r*c2.r - c1.i*c2.i, c1.r*c2.i + c1.i*c2.r)
end

local function inv (c)
  local n = c.r^2 + c.i^2
  return new(c.r/n, -c.i/n)
end

function M.div (c1, c2)
  return M.mul(c1, inv(c2))
end

function M.tostring (c)
  return string.format("(%g,%g)", c.r, c.i)
end

return M
```

Note how we define `new` and `inv` as private functions simply by declaring them local to the chunk.

Some people do not like the final return statement. One way of eliminating it is to assign the module table directly into `package.loaded`:

```
local M = {}
package.loaded[...] = M
  as before, without the return statement
```

Remember that `require` calls the loader passing the module name as the first argument. So, the vararg expression `...` in the table index results in that name. After this assignment, we do not need to return `M` at the end of the module: if a module does not return a value, `require` will return the current value of `package.loaded[modname]` (if it is not nil). Anyway, I find it clearer to write the final return. If we forget it, any trivial test with the module will detect the error.

Another approach to write a module is to define all functions as locals and build the returning table at the end, as in Figure 17.3, “Module with export list”.

Figure 17.3. Module with export list

```
local function new (r, i) return {r=r, i=i} end

-- defines 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,
}
```

What are the advantages of this approach? We do not need to prefix each name with `M.` or something similar; there is an explicit export list; and we define and use exported and internal functions in the same way inside the module. What are the disadvantages? The export list is at the end of the module instead of at the beginning, where it would be more useful as a quick documentation; and the export list is somewhat redundant, as we must write each name twice. (This last disadvantage may become an advantage, as it allows functions to have different names inside and outside the module, but I think programmers seldom do this.)

Anyway, remember that no matter how we define a module, 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)
```

Later, we will see how we can use some advanced Lua features, such as metatables and environments, for writing modules. However, except for a nice technique to detect global variables created by mistake, I use only the basic approach in my modules.

Submodules and Packages

Lua allows module names to be hierarchical, using a dot to separate name levels. For instance, a module named `mod.sub` is a *submodule* of `mod`. A *package* is a complete tree of modules; it is the unit of distribution in Lua.

When we require a module called `mod.sub`, the function `require` will query first the table `package.loaded` and then the table `package.preload`, using the original module name `"mod.sub"` as the key. Here, the dot is just a character like any other in the module name.

However, when searching for a file that defines that submodule, `require` translates the dot into another character, usually the system's directory separator (e.g., a slash for POSIX or a backslash for Windows). After the translation, `require` searches for the resulting name like any other name. For instance, assume the slash as the directory separator and the following path:

```
./?.lua;/usr/local/lua/?.lua;/usr/local/lua/?/init.lua
```

The call `require "a.b"` 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 of a package to live in a single directory. For instance, if a package has modules `p`, `p.a`, and `p.b`, their respective files can be `p/init.lua`, `p/a.lua`, and `p/b.lua`, with the directory `p` within some appropriate directory.

The directory separator used by Lua is configured at compile time and can be any string (remember, Lua knows nothing about directories). For instance, systems without hierarchical directories can use an underscore as the "directory separator", so that `require "a.b"` will search for a file `a_b.lua`.

Names in C cannot contain dots, so a C library for submodule `a.b` cannot export a function `luaopen_a.b`. Here, `require` translates the dot into another character, an underscore. So, a C library named `a.b` should name its initialization function `luaopen_a_b`.

As an extra facility, `require` has one more searcher for loading C submodules. When it cannot find either a Lua file or a C file for a submodule, this last searcher searches again the C path, but this time looking for the package name. For example, if the program requires a submodule `a.b.c` this searcher will look for `a`. If it finds a C library for this name, then `require` looks into this library for an appropriate open function, `luaopen_a_b_c` in this example. This facility allows a distribution to put several submodules together, each with its own open function, into a single C library.

From the point of view of Lua, submodules in the same package have no explicit relationship. Requiring a module does not automatically load any of its submodules; similarly, requiring a submodule does not automatically load its parent. Of course, the package implementer is free to create these links if she wants. For instance, a particular module may start by explicitly requiring one or all of its submodules.

Exercises

Exercise 17.1: Rewrite the implementation of double-ended queues (Figure 14.2, "A double-ended queue") as a proper module.

Exercise 17.2: Rewrite the implementation of the geometric-region system (the section called "A Taste of Functional Programming") as a proper module.

Exercise 17.3: What happens in the search for a library if the path has some fixed component (that is, a component without a question mark)? Can this behavior be useful?

Exercise 17.4: Write a searcher that searches for Lua files and C libraries at the same time. For instance, the path used for this searcher could be something like this:

```
./?.lua;./?.so;/usr/lib/lua5.2/?.so;/usr/share/lua5.2/?.lua
```

(Hint: use `package.searchpath` to find a proper file and then try to load it, first with `loadfile` and next with `package.loadlib`.)

Part III. Lua-isms

Table of Contents

Chapter 18. Iterators and the Generic for

We have been using the generic **for** for several tasks through the book, such as reading the lines of a file or traversing the matches of a pattern over a subject. However, we still do not know how to create our own iterators. In this chapter, we will fill this gap. Starting with simple iterators, we will learn how to use all the power of the generic **for** to write all kinds of iterators.

Iterators and Closures

An *iterator* is any construction that allows us to iterate over the elements of a collection. In Lua, we typically represent iterators by functions: each time we call the function, it returns the "next" element from the collection. A typical example is `io.read`: each time we call it, it returns the next line of the standard input file, returning nil when there are no more lines to be read.

Any iterator needs to keep some state between successive calls, so that it knows where it is and how to proceed from there. For `io.read`, C keeps that state in its stream structure. For our own iterators, closures provide an excellent mechanism for keeping state. Remember that a closure is a function that accesses one or more local variables from its enclosing environment. These variables keep their values across successive calls to the closure, allowing the closure to remember where it is along a traversal. Of course, to create a new closure we must also create its non-local variables. Therefore, a closure construction typically involves two functions: the closure itself and a *factory*, the function that creates the closure plus its enclosing variables.

As an example, let us write a simple iterator for a list. Unlike `ipairs`, this iterator does not return the index of each element, only its value:

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

In this example, `values` is the factory. Each time we call this factory, it creates a new closure (the iterator itself). This closure keeps its state in its external variables `t` and `i`, which are also created by `values`. Each time we call the iterator, it returns a next value from the list `t`. After the last element the iterator returns nil, which signals the end of the iteration.

We can use this iterator in a **while** loop:

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

However, it is easier to use the generic **for**. After all, it was designed for this kind of iteration:

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

```
  print(element)
end
```

The generic **for** does all the bookkeeping for an iteration loop: it keeps the iterator function internally, so that we do not need the `iter` variable; it calls the iterator for each new iteration; and it stops the loop when the iterator returns nil. (In the next section, we will see that the generic **for** does even more than that.)

As a more advanced example, Figure 18.1, “Iterator to traverse all words from the standard input” shows an iterator to traverse all the words from the standard input.

Figure 18.1. Iterator to traverse all words from the standard input

```
function allwords ()
  local line = io.read()  -- current line
  local pos = 1           -- current position in the line
  return function ()      -- iterator function
    while line do         -- repeat while there are lines
      local w, e = string.match(line, "(%w+)()", pos)
      if w then           -- found a word?
        pos = e           -- next position is after this word
        return w          -- return the word
      else
        line = io.read()  -- word not found; try next line
        pos = 1           -- restart from first position
      end
    end
    return nil            -- no more lines: end of traversal
  end
end
```

To do this traversal, we keep two values: the contents of the current line (variable `line`), and where we are on this line (variable `pos`). With this data, we can always generate the next word. The main part of the iterator function is the call to `string.match`, which searches for a word in the current line starting at the current position. It describes a “word” using the pattern `'%w+'`, which matches one or more alphanumeric characters. If it finds a word, it captures and returns the word and the position of the first character after it (with an empty capture). The function then updates the current position and returns this word. Otherwise, the iterator reads a new line and repeats the search. If there are no more lines, it returns nil to signal the end of the iteration.

Despite its complexity, the use of `allwords` is straightforward:

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

This is a common situation with iterators: they may not be easy to write, but they are easy to use. This is not a big problem; more often than not, end users programming in Lua do not define iterators, but just use those provided by the application.

The Semantics of the Generic for

One drawback of those previous iterators is that we need to create a new closure to initialize each new loop. For many situations, this is not a real problem. For instance, in the `allwords` iterator, the cost of creating one single closure is negligible compared to the cost of reading a whole file. However, in some

situations this overhead can be inconvenient. In such cases, we can use the generic **for** itself to keep the iteration state. In this section, we will see the facilities that the generic **for** offers to hold state.

We saw that the generic **for** keeps the iterator function internally, during the loop. Actually, it keeps three values: the iterator function, an *invariant state*, and a *control variable*. Let us see the details now.

The syntax for the generic **for** is as follows:

```
for var-list in exp-list do
  body
end
```

Here, *`var-list`* is a list of one or more variable names, separated by commas, and *`exp-list`* is a list of one or more expressions, also separated by commas. Usually the expression list has only one element, a call to an iterator factory. In the next code, for instance, the list of variables is `k,v` and the list of expressions has the single element `pairs(t)`:

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

We call the first (or only) variable in the list the *control variable*. Its value is never nil during the loop, because when it becomes nil the loop ends.

The first thing the **for** does is to evaluate the expressions after the **in**. These expressions should result in the three values kept by the **for**: the iterator function, the invariant state, and the initial value for the control variable. Like in a multiple assignment, only the last (or the only) element of the list can result in more than one value; and the number of values is adjusted to three, extra values being discarded or nils added as needed. For instance, when we use simple iterators, the factory returns only the iterator function, so the invariant state and the control variable get nil.

After this initialization step, the **for** calls the iterator function with two arguments: the invariant state and the control variable. From the standpoint of the **for** construct, the invariant state has no meaning at all. The **for** only passes the state value from the initialization step to all calls to the iterator function. Then the **for** assigns the values returned by the iterator function to the variables declared by its variable list. If the first returned value (the one assigned to the control variable) is nil, the loop terminates. Otherwise, the **for** executes its body and calls the iteration function again, repeating the process.

More precisely, a construction like

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

is 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 iterator function is *f*, the invariant state is *s*, and the initial value for the control variable is a_0, the control variable will loop over the values $a_1 = f(s, a_0)$, $a_2 = f(s, a_1)$, and so on, until a_i is nil. If the **for** has other variables, they simply get the extra values returned by each call to *f*.

Stateless Iterators

As the name implies, a stateless iterator is an iterator that does not keep any state by itself. Therefore, we can use the same stateless iterator in multiple loops, avoiding the cost of creating new closures.

As we just saw, the **for** loop calls its iterator function with two arguments: the invariant state and the control variable. A stateless iterator generates the next element for the iteration using only these two values. A typical example of this kind of iterator is `ipairs`, which iterates over all elements of a sequence:

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

The whole state of the iteration comprises the table being traversed (the invariant state, which does not change during the loop), plus the current index (the control variable). Both `ipairs` (the factory) and the iterator are quite simple; we could write them in Lua as follows:

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

function ipairs (t)
  return iter, t, 0
end
```

When Lua calls `ipairs(t)` in a **for** loop, it gets three values: the function `iter` as the iterator, the table `t` as the invariant state, and zero as the initial value for the control variable. Then, Lua calls `iter(t, 0)`, which results in `1,t[1]` (unless `t[1]` is already nil). In the second iteration, Lua calls `iter(t, 1)`, which results in `2,t[2]`, and so on, until the first nil element.

The function `pairs`, which iterates over all elements of a table, is similar, except that its iterator function is `next`, which is a primitive function in Lua:

```
function pairs (t)
  return next, t, nil
end
```

The call `next(t, k)`, where `k` is a key of the table `t`, returns a next key in the table, in an arbitrary order, plus the value associated with this key as a second return value. The call `next(t, nil)` returns a first pair. When there are no more pairs, `next` returns nil.

We might use `next` directly, without calling `pairs`:

```
for k, v in next, t do
  loop body
end
```

Remember that the **for** loop adjusts its expression list to three results, so that it gets `next`, `t`, and nil; this is exactly what it gets when it calls `pairs(t)`.

Another interesting example of a stateless iterator is one to traverse a linked list. (Linked lists are not frequent in Lua, but sometimes we need them.) A first thought could be to use only the current node as the control variable, so that the iterator function could return its next node:

```
local function getnext (node)
    return node.next
end

function traverse (list)
  return getnext, nil, list
end
```

However, this implementation would skip the first node. Instead, we can use the following code:

```
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
```

The trick here is to use the first node as the invariant state (the second value returned by `traverse`), besides the current node as the control variable. The first time the iterator function `getnext` is called, `node` will be nil, and so the function will return `list` as the first node. In subsequent calls, `node` will not be nil, and so the iterator will return `node.next`, as expected.

Traversing Tables in Order

A common confusion happens when programmers try to order the entries of a table. In a table, the entries form a set, and have no order whatsoever. If we want to order them, we have to copy the keys to an array and then sort the array.

We saw an example of this technique in the “Most Frequent Words” program, in Chapter 11, *Interlude: Most Frequent Words*. Let us see here another example. Suppose that we read a source file and build a table that gives, for each function name, the line where this function is defined; something like this:

```
lines = {
  ["luaH_set"] = 10,
  ["luaH_get"] = 24,
  ["luaH_present"] = 48,
}
```

Now we want to print these function names in alphabetical order. If we traverse this table with `pairs`, the names appear in an arbitrary order. We cannot sort them directly, because these names are keys of the table. However, when we put them into an array, then we can sort them. First, we must create an array with these names, then sort it, and finally print the 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 people get confused here. After all, for Lua, arrays also have no order (they are tables, after all). But we know how to count! So, we impose the order, when we access the array with ordered indices. That is why we should always traverse arrays with `ipairs`, rather than `pairs`. The first function imposes the key order 1, 2, etc., whereas the latter uses the natural arbitrary order of the table (which may not be what we need, even though usually it is).

Now we are ready to write an iterator that traverses a table following the order of its keys:

```
function pairsByKeys (t, f)
  local a = {}
  for n in pairs(t) do   -- create a list with all keys
    a[#a + 1] = n
  end
  table.sort(a, f)       -- sort the list
  local i = 0            -- iterator variable
  return function ()     -- iterator function
    i = i + 1
    return a[i], t[a[i]]   -- return key, value
  end
end
```

The factory function `pairsByKeys` first collects the keys into an array, then it sorts the array, and finally it returns the iterator function. At each step, the iterator returns the next key and value from the original table, following the order in the array `a`. An optional parameter `f` allows the specification of an alternative order.

With this function, it is easy to solve our initial problem of traversing a table in order:

```
for name, line in pairsByKeys(lines) do
  print(name, line)
end
```

As usual, all the complexity is hidden inside the iterator.

True Iterators

The name “iterator” is a little misleading, because our iterators do not iterate: what iterates is the **for** loop. Iterators only provide the successive values for the iteration. Maybe a better name would be “generator” — which *generates* elements for the iteration— but “iterator” is already well established in other languages, such as Java.

However, there is another way to build iterators wherein iterators actually do the iteration. When we use such iterators, we do not write a loop; instead, we simply call the iterator with an argument that describes what the iterator must do at each iteration. More specifically, the iterator receives as argument a function that it calls inside its loop.

As a concrete example, let us rewrite once more the `allwords` iterator using this style:

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

```
  end
end
```

To use this iterator, we must supply the loop body as a function. If we want only to print each word, we simply use `print`:

```
allwords(print)
```

Often, we use an anonymous function as the body. For instance, the next code fragment counts how many times the word “hello” appears in the input file:

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

The same task, written with the previous iterator style, is not very different:

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

True iterators were popular in older versions of Lua, when the language did not have the **for** statement. How do they compare with generator-style iterators? Both styles have approximately the same overhead: one function call per iteration. On the one hand, it is easier to write the iterator with true iterators (although we can recover this easiness with coroutines, as we will see in the section called “Coroutines as Iterators”). On the other hand, the generator style is more flexible. First, it allows two or more parallel iterations. (For instance, consider the problem of iterating over two files comparing them word by word.) Second, it allows the use of **break** and **return** inside the iterator body. With a true iterator, a **return** returns from the anonymous function, not from the function doing the iteration. For these reasons, overall I usually prefer generators.

Exercises

Exercise 18.1: Write an iterator `fromto` such that the next loop becomes equivalent to a numeric **for**:

```
for i in fromto(n, m) do
  body
end
```

Can you implement it as a stateless iterator?

Exercise 18.2: Add a step parameter to the iterator from the previous exercise. Can you still implement it as a stateless iterator?

Exercise 18.3: Write an iterator `uniquewords` that returns all words from a given file without repetitions. (Hint: start with the `allwords` code in Figure 18.1, “Iterator to traverse all words from the standard input”; use a table to keep all words already reported.)

Exercise 18.4: Write an iterator that returns all non-empty substrings of a given string.

Exercise 18.5: Write a true iterator that traverses all subsets of a given set. (Instead of creating a new table for each subset, it can use the same table for all its results, only changing its contents between iterations.)

Chapter 19. Interlude: Markov Chain Algorithm

Our next complete program is an implementation of the Markov chain algorithm, described by Kernighan & Pike in their book *The Practice of Programming* (Addison-Wesley, 1999).

The program generates pseudo-random text based on what words can follow a sequence of *n* previous words in a base text. For this implementation, we will assume that *n* is two.

The first part of the program reads the base text and builds a table that, for each prefix of two words, gives a list of the words that follow that prefix in the text. After building the table, the program uses it to generate random text, wherein each word follows two previous words with the same probability as in the base text. As a result, we have text that is very, but not quite, random. For instance, when applied to this book, the output of the program has pieces like this: "*Constructors can also traverse a table constructor, 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 argument. 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.*"

To use a two-word prefix as a key in tables, we will represent it by the two words concatenated with a space in between:

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

We use the string `NOWORD` (a newline) to initialize the prefix words and to mark the end of the text. For instance, for the text `"the more we try the more we do"` the table of following words would be like this:

```
{ ["\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 keeps its table in the variable `statetab`. To insert a new word in a list in this table, we use the following function:

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

It first checks whether that prefix already has a list; if not, it creates a new one with the new value. Otherwise, it inserts the new value at the end of the existing list.

To build the `statetab` table, we keep two variables, `w1` and `w2`, with the last two words read. We read the words using the iterator `allwords`, from the section called “Iterators and Closures”, but we adapted the definition of “word” to include optional punctuation marks, such as commas and periods (see Figure 19.1, “Auxiliary definitions for the Markov program”). For each new word read, we add it to the list associated with `w1`–`w2` and then update `w1` and `w2`.

After building the table, the program starts to generate a text with `MAXGEN` words. First, it re-initializes variables `w1` and `w2`. Then, for each prefix, it chooses a next word randomly from the list of valid next words, prints this word, and updates `w1` and `w2`. Figure 19.1, “Auxiliary definitions for the Markov program” and Figure 19.2, “The Markov program” show the complete program.

Figure 19.1. Auxiliary definitions for the Markov program

```
function allwords ()
  local line = io.read()    -- current line
  local pos = 1             -- current position in the line
  return function ()        -- iterator function
    while line do           -- repeat while there are lines
      local w, e = string.match(line, "(%w+[,;.:]?)()", pos)
      if w then                          -- found a word?
        pos = e                          -- update next position
        return w                         -- return the word
      else
        line = io.read()         -- word not found; try next line
        pos = 1                  -- restart from first position
      end
    end
    return nil              -- no more lines: end of traversal
  end
end

function prefix (w1, w2)
  return w1 .. " " .. w2
end

local statetab = {}

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

Figure 19.2. The Markov program

```
local MAXGEN = 200
local NOWORD = "\n"

-- build table
local w1, w2 = NOWORD, NOWORD
for nextword in allwords() do
  insert(prefix(w1, w2), nextword)
  w1 = w2; w2 = nextword;
end
insert(prefix(w1, w2), NOWORD)

-- generate text
w1 = NOWORD; w2 = NOWORD     -- reinitialize
for i = 1, MAXGEN do
  local list = statetab[prefix(w1, w2)]
  -- choose a random item from 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 19.1: Generalize the Markov-chain algorithm so that it can use any size for the sequence of previous words used in the choice of the next word.

Chapter 20. Metatables and Metamethods

Usually, each value in Lua has a quite predictable set of operations. We can add numbers, we can concatenate strings, we can insert key–value pairs into tables, and so on. However, we cannot add tables, we cannot compare functions, and we cannot call a string. Unless we use metatables.

Metatables allow us to change the behavior of a value when confronted with an unknown operation. For instance, using metatables, we can define how Lua computes the expression `a + b`, where `a` and `b` are tables. Whenever Lua tries to add two tables, it checks whether either of them has a *metatable* and whether this metatable has an `__add` field. If Lua finds this field, it calls the corresponding value —the so-called *metamethod*, which should be a function— to compute the sum.

We can think about metatables as a restricted kind of classes, in object-oriented terminology. Like classes, metatables define the behavior of its instances. However, metatables are more restricted than classes, because they can only give behavior to a predefined set of operations; also, metatables do not have inheritance. Nevertheless, we will see in Chapter 21, *Object-Oriented Programming* how to build a quite complete class system on top of metatables.

Each value in Lua can have a metatable. Tables and userdata have individual metatables; values of other types share one single metatable for all values of that type. Lua always creates new tables without metatables:

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

We can use `setmetatable` to set or change the metatable of a table:

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

From Lua, we can set the metatables only of tables; to manipulate the metatables of values of other types we must use C code or the debug library. (The main reason for this restriction is to curb excessive use of type-wide metatables. Experience with older versions of Lua has shown that those global settings frequently lead to non-reusable code.) The string library sets a metatable for strings; all other types by default have no metatable:

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

Any table can be the metatable of any value; a group of related tables can share a common metatable, which describes their common behavior; a table can be its own metatable, so that it describes its own individual behavior. Any configuration is valid.

Arithmetic Metamethods

In this section, we will introduce a running example to explain the basics of metatables. Suppose we have a module that uses tables to represent sets, with functions to compute set union, intersection, and the like, as shown in Figure 20.1, “A simple module for sets”.

Figure 20.1. A simple module for sets

```
local Set = {}

-- create a new set with the values of 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
    res[k] = b[k]
  end
  return res
end

-- presents a set as a string
function Set.tostring (set)
  local l = {}     -- list to put all elements from the set
  for e in pairs(set) do
    l[#l + 1] = tostring(e)
  end
  return "{" .. table.concat(l, ", ") .. "}"
end

return Set
```

Now, we want to use the addition operator to compute the union of two sets. For that, we will arrange for all tables representing sets to share a metatable. This metatable will define how they react to the addition operator. Our first step is to create a regular table, which we will use as the metatable for sets:

```
local mt = {}    -- metatable for sets
```

The next step is to modify `Set.new`, which creates sets. The new version has only one extra line, which sets `mt` as the metatable for the tables that it creates:

```
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, every set we create with `Set.new` will have that same table as its metatable:

```
s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
print(getmetatable(s1))          --> table: 0x00672B60
print(getmetatable(s2))          --> table: 0x00672B60
```

Finally, we add to the metatable the *metamethod* `__add`, a field that describes how to perform the addition:

```
mt.__add = Set.union
```

After that, whenever Lua tries to add two sets, it will call `Set.union`, with the two operands as arguments.

With the metamethod in place, we can use the addition operator to do set unions:

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

Similarly, we may set the multiplication operator to perform set intersection:

```
mt.__mul = Set.intersection

print(Set.tostring((s1 + s2)*s1))     --> {10, 20, 30, 50}
```

For each arithmetic operator there is a corresponding metamethod name. Besides addition and multiplication, there are metamethods for subtraction (`__sub`), float division (`__div`), floor division (`__idiv`), negation (`__unm`), modulo (`__mod`), and exponentiation (`__pow`). Similarly, there are metamethods for all bitwise operations: bitwise AND (`__band`), OR (`__bor`), exclusive OR (`__bxor`), NOT (`__bnot`), left shift (`__shl`), and right shift (`__shr`). We may define also a behavior for the concatenation operator, with the field `__concat`.

When we add two sets, there is no question about what metatable to use. However, we may write an expression that mixes two values with different metatables, for instance like this:

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

When looking for a metamethod, Lua performs the following steps: if the first value has a metatable with the required metamethod, Lua uses this metamethod, independently of the second value; otherwise, if the second value has a metatable with the required metamethod, Lua uses it; otherwise, Lua raises an error. Therefore, the last example will call `Set.union`, as will the expressions `10 + s` and `"hello" + s` (because both numbers and strings do not have a metamethod `__add`).

Lua does not care about these mixed types, but our implementation does. If we run the `s = s + 8` example, we will get an error inside the function `Set.union`:

```
bad argument #1 to 'pairs' (table expected, got number)
```

If we want more lucid error messages, we must check the type of the operands explicitly before attempting to perform the operation, for instance with code like this:

```
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 `error` (2, in this example) sets the source location in the error message to the code that called the operation.

Relational Metamethods

Metatables also allow us to give meaning to the relational operators, through the metamethods `__eq` (*equal to*), `__lt` (*less than*), and `__le` (*less than or equal to*). There are no separate metamethods for the other three relational operators: Lua translates `a ~= b` to `not (a == b)`, `a > b` to `b < a`, and `a >= b` to `b <= a`.

In older versions, Lua translated all order operators to a single one, by translating `a <= b` to `not (b < a)`. However, this translation is incorrect when we have a *partial order*, that is, when not all elements in our type are properly ordered. For instance, most machines do not have a total order for floating-point numbers, because of the value *Not a Number* (*NaN*). According to the IEEE 754 standard, NaN represents undefined values, such as the result of *0/0*. The standard specifies that any comparison that involves NaN should result in false. This means that `NaN <= x` is always false, but `x < NaN` is also false. It also means that the translation from `a <= b` to `not (b < a)` is not valid in this case.

In our example with sets, we have a similar problem. An obvious (and useful) meaning for `<=` in sets is set containment: `a <= b` means that `a` is a subset of `b`. With this meaning, it is possible that `a <= b` and `b < a` are both false. Therefore, we must implement both `__le` (*less or equal*, the subset relation) and `__lt` (*less than*, the proper subset relation):

```
mt.__le = function (a, b)    -- subset
  for k in pairs(a) do
    if not b[k] then return false end
  end
  return true
end

mt.__lt = function (a, b)    -- proper subset
  return a <= b and not (b <= a)
end
```

Finally, we can define set equality through set containment:

```
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
```

The equality comparison has some restrictions. If two objects have different basic types, the equality operation results in **false**, without even calling any metamethod. So, a set will always be different from a number, no matter what its metamethod says.

Library-Defined Metamethods

So far, all the metamethods we have seen are for the Lua core. It is the virtual machine who detects that the values involved in an operation have metatables with metamethods for that particular operation. However,

because metatables are regular tables, anyone can use them. So, it is a common practice for libraries to define and use their own fields in metatables.

The function `tostring` provides a typical example. As we saw earlier, `tostring` represents tables in a rather simple format:

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

The function `print` always calls `tostring` to format its output. However, when formatting any value, `tostring` first checks whether the value has a `__tostring` metamethod. In this case, `tostring` calls the metamethod to do its job, passing the object as an argument. Whatever this metamethod returns is the result of `tostring`.

In our example with sets, we have already defined a function to present a set as a string. So, we need only to set the `__tostring` field in the metatable:

```
mt.__tostring = Set.tostring
```

After that, whenever we call `print` with a set as its argument, `print` calls `tostring` that calls `Set.tostring`:

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

The functions `setmetatable` and `getmetatable` also use a metafield, in this case to protect metatables. Suppose we want to protect our sets, so that users can neither see nor change their metatables. If we set a `__metatable` field in the metatable, `getmetatable` will return the value of this field, whereas `setmetatable` will raise an error:

```
mt.__metatable = "not your business"

s1 = Set.new{}
print(getmetatable(s1))     --> not your business
setmetatable(s1, {})
  stdin:1: cannot change protected metatable
```

Since Lua 5.2, `pairs` also have a metamethod, so that we can modify the way a table is traversed and add a traversal behavior to non-table objects. When an object has a `__pairs` metamethod, `pairs` will call it to do all its work.

Table-Access Metamethods

The metamethods for arithmetic, bitwise, and relational operators all define behavior for otherwise erroneous situations; they do not change the normal behavior of the language. Lua also offers a way to change the behavior of tables for two normal situations, the access and modification of absent fields in a table.

The `__index` metamethod

We saw earlier that, when we access an absent field in a table, the result is nil. This is true, but it is not the whole truth. Actually, such accesses trigger the interpreter to look for an `__index` metamethod: if there is no such method, as usually happens, then the access results in nil; otherwise, the metamethod will provide the result.

The archetypal example here is inheritance. Suppose we want to create several tables describing windows. Each table must describe several window parameters, such as position, size, color scheme, and the like. All these parameters have default values and so we want to build window objects giving only the non-

default parameters. A first alternative is to provide a constructor that fills in the absent fields. A second alternative is to arrange for the new windows to *inherit* any absent field from a prototype window. First, we declare the prototype:

```
-- create the prototype with default values
prototype = {x = 0, y = 0, width = 100, height = 100}
```

Then, we define a constructor function, which creates new windows sharing a metatable:

```
local mt = {}    -- create a metatable
-- declare the constructor function
function new (o)
  setmetatable(o, mt)
  return o
end
```

Now, we define the `__index` metamethod:

```
mt.__index = function (_, key)
  return prototype[key]
end
```

After this code, we create a new window and query it for an absent field:

```
w = new{x=10, y=20}
print(w.width)    --> 100
```

Lua detects that `w` does not have the requested field, but has a metatable with an `__index` field. So, Lua calls this `__index` metamethod, with arguments `w` (the table) and `"width"` (the absent key). The metamethod then indexes the prototype with the given key and returns the result.

The use of the `__index` metamethod for inheritance is so common that Lua provides a shortcut. Despite being called a *method*, the `__index` metamethod does not need to be a function: it can be a table, instead. When it is a function, Lua calls it with the table and the absent key as its arguments, as we have just seen. When it is a table, Lua redoes the access in this table. Therefore, in our previous example, we could declare `__index` simply like this:

```
mt.__index = prototype
```

Now, when Lua looks for the metatable's `__index` field, it finds the value of `prototype`, which is a table. Consequently, Lua repeats the access in this table, that is, it executes the equivalent of `prototype["width"]`. This access then gives the desired result.

The use of a table as an `__index` metamethod provides a fast and simple way of implementing single inheritance. A function, although more expensive, provides more flexibility: we can implement multiple inheritance, caching, and several other variations. We will discuss these forms of inheritance in Chapter 21, *Object-Oriented Programming*, when we will cover object-oriented programming.

When we want to access a table without invoking its `__index` metamethod, we use the function `rawget`. The call `rawget(t, i)` does a *raw* access to table `t`, that is, a primitive access without considering metatables. Doing a raw access will not speed up our code (the overhead of a function call kills any gain we could have), but sometimes we need it, as we will see later.

The `__newindex` metamethod

The `__newindex` metamethod does for table updates what `__index` does for table accesses. When we assign a value to an absent index in a table, the interpreter looks for a `__newindex` metamethod: if there

is one, the interpreter calls it instead of making the assignment. Like `__index`, if the metamethod is a table, the interpreter does the assignment in this table, instead of in the original one. Moreover, there is a raw function that allows us to bypass the metamethod: the call `rawset(t, k, v)` does the equivalent to `t[k] = v` without invoking any metamethod.

The combined use of the `__index` and `__newindex` metamethods allows several powerful constructs in Lua, such as read-only tables, tables with default values, and inheritance for object-oriented programming. In this chapter, we will see some of these uses. Object-oriented programming has its own chapter, later.

Tables with default values

The default value of any field in a regular table is nil. It is easy to change this default value 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 the call to `setDefault`, any access to an absent field in `tab` calls its `__index` metamethod, which returns zero (the value of `d` for this metamethod).

The function `setDefault` creates a new closure plus a new metatable for each table that needs a default value. This can be expensive if we have many tables that need default values. However, the metatable has the default value `d` wired into its metamethod, so we cannot use a single metatable for tables with different default values. To allow the use of a single metatable for all tables, we can store the default value of each table in the table itself, using an exclusive field. If we are not worried about name clashes, we can use a key like `"___"` for our exclusive field:

```
local mt = {__index = function (t) return t.___ end}
function setDefault (t, d)
  t.___ = d
  setmetatable(t, mt)
end
```

Note that now we create the metatable `mt` and its corresponding metamethod only once, outside `SetDefault`.

If we are worried about name clashes, it is easy to ensure the uniqueness of the special key. All we need is a new exclusive table to use as the 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
```

An alternative approach for associating each table with its default value is a technique I call *dual representation*, which uses a separate table where the indices are the tables and the values are their default values. However, for the correct implementation of this approach, we need a special breed of table called *weak tables*, and so we will not use it here; we will return to the subject in Chapter 23, *Garbage*.

Another alternative is to *memorize* metatables in order to reuse the same metatable for tables with the same default. However, that needs weak tables too, so that again we will have to wait until Chapter 23, *Garbage*.

Tracking table accesses

Suppose we want to track every single access to a certain table. Both `__index` and `__newindex` are relevant only when the index does not exist in the table. So, the only way to catch all accesses to a table is to keep it empty. If we want to monitor all accesses to a table, we should create a *proxy* for the real table. This proxy is an empty table, with proper metamethods that track all accesses and redirect them to the original table. The code in Figure 20.2, “Tracking table accesses” implements this idea.

Figure 20.2. Tracking table accesses

```
function track (t)
  local proxy = {}        -- proxy table for 't'

  -- create metatable for the proxy
  local mt = {
    __index = function (_, k)
      print("*access to element " .. tostring(k))
      return t[k]   -- access the original table
    end,

    __newindex = function (_, k, v)
      print("*update of element " .. tostring(k) ..
            " to " .. tostring(v))
      t[k] = v   -- update original table
    end,

    __pairs = function ()
      return function (_, k)   -- iteration function
        local nextkey, nextvalue = next(t, k)
        if nextkey ~= nil then   -- avoid last value
          print("*traversing element " .. tostring(nextkey))
        end
        return nextkey, nextvalue
      end
    end,

    __len = function () return #t end
  }

  setmetatable(proxy, mt)

  return proxy
end
```

The following example illustrates its use:

```
> t = {}               -- an arbitrary table
> t = track(t)
> t[2] = "hello"
  --> *update of element 2 to hello
> print(t[2])
```

```
  --> *access to element 2
  --> hello
```

The metamethods `__index` and `__newindex` follow the guidelines that we set, tracking each access and then redirecting it to the original table. The `__pairs` metamethod allows us to traverse the proxy as if it were the original table, tracking the accesses along the way. Finally, the `__len` metamethod gives the length operator through the proxy:

```
t = track({10, 20})
print(#t)              --> 2
for k, v in pairs(t) do print(k, v) end
  --> *traversing element 1
  --> 1 10
  --> *traversing element 2
  --> 2 20
```

If we want to monitor several tables, we do not need a different metatable for each one. Instead, we can somehow map each proxy to its original table and share a common metatable for all proxies. This problem is similar to the problem of associating tables to their default values, which we discussed in the previous section, and allows the same solutions. For instance, we can keep the original table in a proxy's field, using an exclusive key, or we can use a dual representation to map each proxy to its corresponding table.

Read-only tables

It is easy to adapt the concept of proxies to implement read-only tables. All we have to do is to raise an error whenever we track any attempt to update the table. For the `__index` metamethod, we can use a table —the original table itself— instead of a function, as we do not need to track queries; it is simpler and rather more efficient to redirect all queries to the original table. This use demands a new metatable for each read-only proxy, with `__index` 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 use, we can create a read-only table for weekdays:

```
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 20.1: Define a metamethod `__sub` for sets that returns the difference of two sets. (The set *a* - *b* is the set of elements from *a* that are not in *b*.)

Exercise 20.2: Define a metamethod `__len` for sets so that `#s` returns the number of elements in the set `s`.

Exercise 20.3: An alternative way to implement read-only tables might use a function as the `__index` metamethod. This alternative makes accesses more expensive, but the creation of read-only tables is cheaper, as all read-only tables can share a single metatable. Rewrite the function `readOnly` using this approach.

Exercise 20.4: Proxy tables can represent other kinds of objects besides tables. file as array Write a function `fileAsArray` that takes the name of a file and returns a proxy to that file, so that after a call `t = fileAsArray("myFile")`, an access to `t[i]` returns the `i`-th byte of that file, and an assignment to `t[i]` updates its `i`-th byte.

Exercise 20.5: Extend the previous example to allow us to traverse the bytes in the file with `pairs(t)` and get the file length with `#t`.

Chapter 21. Object-Oriented Programming

A table in Lua is an object in more than one sense. Like objects, tables have a state. Like objects, tables have an identity (a *self*) that is independent of their values; specifically, two objects (tables) with the same value are different objects, whereas an object can have different values at different times. Like objects, tables have a life cycle that is independent of who created them or where they were created.

Objects have their own operations. Tables also can have operations, as in the following fragment:

```
Account = {balance = 0}
function Account.withdraw (v)
  Account.balance = Account.balance - v
end
```

This definition creates a new function and stores it in field `withdraw` of the object `Account`. Then, we can call it like here:

```
Account.withdraw(100.00)
```

This kind of function is almost what we call a *method*. However, the use of the global name `Account` inside the function is a horrible programming practice. First, this function will work only for this particular object. Second, even for this particular object, the function will work only as long as the object is stored in that particular global variable. If we change the object's name, `withdraw` does not work any more:

```
a, Account = Account, nil
a.withdraw(100.00)      -- ERROR!
```

Such behavior violates the principle that objects have independent life cycles.

A more principled approach is to operate on the *receiver* of the operation. For that, our method would need an extra parameter with the value of the receiver. This parameter usually has the name *self* or *this*:

```
function Account.withdraw (self, v)
  self.balance = self.balance - v
end
```

Now, when we call the method we have to specify the object that it has to operate on:

```
a1 = Account; Account = nil
...
a1.withdraw(a1, 100.00)   -- OK
```

With the use of a *self* parameter, we can use the same method for many objects:

```
a2 = {balance=0, withdraw = Account.withdraw}
...
a2.withdraw(a2, 260.00)
```

This use of a *self* parameter is a central point in any object-oriented language. Most OO languages have this mechanism hidden from the programmer, so that she does not have to declare this parameter (although she still can use the name *self* or *this* inside a method). Lua also can hide this parameter, with the *colon*

operator. Using it, we can rewrite the previous method call as `a2:withdraw(260.00)` and the previous definition as here:

```
function Account:withdraw (v)
  self.balance = self.balance - v
end
```

The effect of the colon is to add an extra argument in a method call and to add an extra hidden parameter in a method definition. The colon is only a syntactic facility, although a convenient one; there is nothing really new here. We can define a function with the dot syntax and call it with the colon syntax, or 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)
```

Classes

So far, our objects have an identity, state, and operations on this state. They still lack a class system, inheritance, and privacy. Let us tackle the first problem: how can we create several objects with similar behavior? Specifically, how can we create several accounts?

Most object-oriented languages offer the concept of class, which works as a mold for the creation of objects. In such languages, each object is an instance of a specific class. Lua does not have the concept of class; the concept of metatables is somewhat similar, but using it as a class would not lead us too far. Instead, we can emulate classes in Lua following the lead from prototype-based languages like Self. (Javascript also followed that path.) In these languages, objects have no classes. Instead, each object may have a prototype, which is a regular object where the first object looks up any operation that it does not know about. To represent a class in such languages, we simply create an object to be used exclusively as a prototype for other objects (its instances). Both classes and prototypes work as a place to put behavior to be shared by several objects.

In Lua, we can implement prototypes using the idea of inheritance that we saw in the section called “The `__index` metamethod”. More specifically, if we have two objects `A` and `B`, all we have to do to make `B` a prototype for `A` is this:

```
setmetatable(A, {__index = B})
```

After that, `A` looks up in `B` for any operation that it does not have. To see `B` as the class of the object `A` is not much more than a change in terminology.

Let us go back to our example of a bank account. To create other accounts with behavior similar to `Account`, we arrange for these new objects to inherit their operations from `Account`, using the `__index` metamethod.

```
local mt = {__index = Account}

function Account.new (o)
  o = o or {}     -- create table if user does not provide one
  setmetatable(o, mt)
  return o
end
```

After this code, what happens when we create a new account and call a method on it, like this?

```
a = Account.new{balance = 0}
a:deposit(100.00)
```

When we create the new account, `a`, it will have `mt` as its metatable. When we call `a:deposit(100.00)`, we are actually calling `a.deposit(a, 100.00)`; the colon is only syntactic sugar. However, Lua cannot find a `"deposit"` entry in the table `a`; hence, Lua looks into the `__index` entry of the metatable. The situation now is more or less like this:

```
getmetatable(a).__index.deposit(a, 100.00)
```

The metatable of `a` is `mt`, and `mt.__index` is `Account`. Therefore, the previous expression evaluates to this one:

```
Account.deposit(a, 100.00)
```

That is, Lua calls the original `deposit` function, but passing `a` as the *self* parameter. So, the new account `a` inherited the function `deposit` from `Account`. By the same mechanism, it inherits all fields from `Account`.

We can make two small improvements on this scheme. The first one is that we do not need to create a new table for the metatable role; instead, we can use the `Account` table itself for that purpose. The second one is that we can use the colon syntax for the `new` method, too. With these two changes, method `new` becomes like this:

```
function Account:new (o)
  o = o or {}
  self.__index = self
  setmetatable(o, self)
  return o
end
```

Now, when we call `Account:new()`, the hidden parameter `self` gets `Account` as its value, we make `Account.__index` also equal to `Account`, and set `Account` as the metatable for the new object. It may seem that we do not gained much with the second change (the colon syntax); the advantage of using `self` will become apparent when we introduce class inheritance, in the next section.

The inheritance works not only for methods, but also for other fields that are absent in the new account. Therefore, a class can provide not only methods, but also constants and default values for its instance fields. Remember that, in our first definition of `Account`, we provided a field `balance` with value 0. So, if we create a new account without an initial balance, it will inherit this default value:

```
b = Account:new()
print(b.balance)    --> 0
```

When we call the `deposit` method on `b`, it runs the equivalent of the following code, because `self` is `b`:

```
b.balance = b.balance + v
```

The expression `b.balance` evaluates to zero and the method assigns an initial deposit to `b.balance`. Subsequent accesses to `b.balance` will not invoke the index metamethod, because now `b` has its own `balance` field.

Inheritance

Because classes are objects, they can get methods from other classes, too. This behavior makes inheritance (in the usual object-oriented meaning) quite easy to implement in Lua.

Let us assume we have a base class like `Account`, in Figure 21.1, “the `Account` class”.

Figure 21.1. the `Account` class

```
Account = {balance = 0}

function Account:new (o)
  o = o or {}
  self.__index = self
  setmetatable(o, 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 want to derive a subclass `SpecialAccount` that allows the customer to withdraw more than his balance. We start with an empty class that simply inherits all its operations from its base class:

```
SpecialAccount = Account:new()
```

Up to now, `SpecialAccount` is just an instance of `Account`. The magic happens now:

```
s = SpecialAccount:new{limit=1000.00}
```

`SpecialAccount` inherits `new` from `Account`, like any other method. This time, however, when `new` executes, its `self` parameter will refer to `SpecialAccount`. Therefore, the metatable of `s` will be `SpecialAccount`, whose value at field `__index` is also `SpecialAccount`. So, `s` inherits from `SpecialAccount`, which inherits from `Account`. Later, when we evaluate `s:deposit(100.00)`, Lua cannot find a `deposit` field in `s`, so it looks into `SpecialAccount`; it cannot find a `deposit` field there, too, so it looks into `Account`; there it finds the original implementation for a deposit.

What makes a `SpecialAccount` special is that we can redefine any method inherited from its superclass. All we have to do is to write the 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 does not go to `Account`, because it finds the new `withdraw` method in `SpecialAccount` first. As `s.limit` is 1000.00 (remember that we set this field when we created `s`), the program does the withdrawal, leaving `s` with a negative balance.

An interesting aspect of objects in Lua is that we do not need to create a new class to specify a new behavior. If only a single object needs a specific behavior, we can implement that behavior directly in the object. For instance, if the account `s` represents some special client whose limit is always 10% of her balance, we can modify only this single account:

```
function s:getLimit ()
  return self.balance * 0.10
end
```

After this declaration, the call `s:withdraw(200.00)` runs the `withdraw` method from `SpecialAccount`, but when `withdraw` calls `self:getLimit`, it is this last definition that it invokes.

Multiple Inheritance

Because objects are not primitive in Lua, there are several ways to do object-oriented programming in Lua. The approach that we have seen, using the index metamethod, is probably the best combination of simplicity, performance, and flexibility. Nevertheless, there are other implementations, which may be more appropriate for some particular cases. Here we will see an alternative implementation that allows multiple inheritance in Lua.

The key to this implementation is the use of a function for the metafield `__index`. Remember that, when a table's metatable has a function in the `__index` field, Lua will call this function whenever it cannot find a key in the original table. Then, `__index` can look up for the missing key in how many parents it wants.

Multiple inheritance means that a class does not have a unique superclass. Therefore, we should not use a (super)class method to create subclasses. Instead, we will define an independent function for this purpose, `createClass`, which has as arguments all superclasses of the new class; see Figure 21.2, “An implementation of multiple inheritance”. This function creates a table to represent the new class and sets its metatable with an `__index` metamethod that does the multiple inheritance. Despite the multiple inheritance, each object instance still belongs to one single class, where it looks for all its methods. Therefore, the relationship between classes and superclasses is different from the relationship between instances and classes. Particularly, a class cannot be the metatable for its instances and for its subclasses at the same time. In Figure 21.2, “An implementation of multiple inheritance”, we keep the class as the metatable for its instances, and create another table to be the metatable of the class.

Figure 21.2. An implementation of multiple inheritance

```
-- look up for 'k' in list of tables 'plist'
local function search (k, plist)
  for i = 1, #plist do
    local v = plist[i][k]     -- try 'i'-th superclass
    if v then return v end
  end
end

function createClass (...)
  local c = {}             -- new class
  local parents = {...}    -- list of parents

  -- class searches for absent methods in its list of parents
  setmetatable(c, {__index = function (t, k)
    return search(k, parents)
  end})

  -- prepare 'c' to be the 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 new class
end
```

Let us illustrate the use of `createClass` with a small example. Assume our previous class `Account` and another class, `Named`, with only two 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` that is a subclass of both `Account` and `Named`, we simply call `createClass`:

```
NamedAccount = createClass(Account, Named)
```

To create and to use instances, we do as usual:

```
account = NamedAccount:new{name = "Paul"}
print(account:getname())     --> Paul
```

Now let us follow how Lua evaluates the expression `account:getname()`; more specifically, let us follow the evaluation of `account["getname"]`. Lua cannot find the field `"getname"` in `account`;

so, it looks for the field `__index` on the `account`'s metatable, which is `NamedAccount` in our example. But `NamedAccount` also cannot provide a `"getname"` field, so Lua looks for the field `__index` of `NamedAccount`'s metatable. Because this field contains a function, Lua calls it. This function then looks for `"getname"` first in `Account`, without success, and then in `Named`, where it finds a non-nil value, which is the final result of the search.

Of course, due to the underlying complexity of this search, the performance of multiple inheritance is not the same as single inheritance. A simple way to improve this performance is to copy inherited methods into the subclasses. Using this technique, the index metamethod for classes would be like this:

```
setmetatable(c, {__index = function (t, k)
  local v = search(k, parents)
  t[k] = v       -- save for next access
  return v
end})
```

With this trick, accesses to inherited methods are as fast as to local methods, except for the first access. The drawback is that it is difficult to change method definitions after the program has started, because these changes do not propagate down the hierarchy chain.

Privacy

Many people consider privacy (also called *information hiding*) to be an integral part of an object-oriented language: the state of each object should be its own internal affair. In some object-oriented languages, such as C++ and Java, we can control whether a field (also called an *instance variable*) or a method is visible outside the object. Smalltalk, which popularized object-oriented languages, makes all variables private and all methods public. Simula, the first ever object-oriented language, did not offer any kind of protection.

The standard implementation of objects in Lua, which we have shown previously, does not offer privacy mechanisms. Partly, this is a consequence of our use of a general structure (tables) to represent objects. Moreover, Lua avoids redundancy and artificial restrictions. If you do not want to access something that lives inside an object, just *do not do it*. A common practice is to mark all private names with an underscore at the end. You immediately feel the smell when you see a marked name being used in public.

Nevertheless, another aim of Lua is to be flexible, offering the programmer meta-mechanisms that enable her to emulate many different mechanisms. Although the basic design for objects in Lua does not offer privacy mechanisms, we can implement objects in a different way, to have access control. Although programmers do not use this implementation frequently, it is instructive to know about it, both because it explores some interesting aspects of Lua and because it is a good solution for more specific problems.

The basic idea of this alternative design is to represent each object through two tables: one for its state and another for its operations, or its interface. We access the object itself through the second table, that is, through the operations that compose its interface. To avoid unauthorized access, the table representing the state of an object is not kept in a field of the other table; instead, it is kept only in the closure of the methods. For instance, to represent our bank account with this design, we could create new objects running the following factory function:

```
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
```

First, the function creates a table to keep the internal object state and stores it in the local variable `self`. Then, the function creates the methods of the object. Finally, the function creates and returns the external object, which maps method names to the actual method implementations. The key point here is that these methods do not get `self` as an extra parameter; instead, they access `self` directly. Because there is no extra argument, we do not use the colon syntax to manipulate such objects. We call their methods just like regular functions:

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance())     --> 60
```

This design gives full privacy to anything stored in the `self` table. After the call to `newAccount` returns, there is no way to gain direct access to this table. We can access it only through the functions created inside `newAccount`. Although our example puts only one instance variable into the private table, we can store all private parts of an object in this table. We can also define private methods: they are like public methods, but we do not put them in the interface. For instance, our accounts may give an extra credit of 10% for users with balances above a certain limit, but we do not want the users to have access to the details of that computation. 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 for any user to access the function `extra` directly.

The Single-Method Approach

A particular case of the previous approach for object-oriented programming occurs when an object has a single method. In such cases, we do not need to create an interface table; instead, we can return this single method as the object representation. If this sounds a little weird, it is worth remembering iterators like `io.lines` or `string.gmatch`. An iterator that keeps state internally is nothing more than a single-method object.

Another interesting case of single-method objects occurs when this single-method is actually a dispatch method that performs different tasks based on a distinguished argument. A prototypical implementation for such an object is as follows:

```
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 use is straightforward:

```
d = newObject(0)
print(d("get"))    --> 0
d("set", 10)
print(d("get"))    --> 10
```

This unconventional implementation for objects is quite effective. The syntax `d("set", 10)`, although peculiar, is only two characters longer than the more conventional `d:set(10)`. Each object uses one single closure, which is usually cheaper than one table. There is no inheritance, but we have full privacy: the only way to access an object state is through its sole method.

Tcl/Tk uses a similar approach for its widgets. The name of a widget in Tk denotes a function (a *widget command*) that can perform all kinds of operations over the widget, according to its first argument.

Dual Representation

Another interesting approach for privacy uses a *dual representation*. Let us start seeing what a dual representation is.

Usually, we associate attributes to tables using keys, like this:

```
table[key] = value
```

However, we can use a dual representation: we can use a table to represent a key, and use the object itself as a key in that table:

```
key = {}
...
key[table] = value
```

A key ingredient here is the fact that we can index tables in Lua not only with numbers and strings, but with any value —in particular with other tables.

As an example, in our Account implementation, we could keep the balances of all accounts in a table `balance`, instead of keeping them in the accounts themselves. Our `withdraw` method would become like this:

```
function Account.withdraw (self, v)
  balance[self] = balance[self] - v
end
```

What we gain here? Privacy. Even if a function has access to an account, it cannot directly access its balance unless it also has access to the table `balance`. If the table `balance` is kept in a local inside the module `Account`, only functions inside the module can access it and, therefore, only those functions can manipulate account balances.

Before we go on, I must discuss a big naivety of this implementation. Once we use an account as a key in the `balance` table, that account will never become garbage for the garbage collector. It will be anchored there until some code explicitly removes it from that table. That may not be a problem for bank accounts (as an account usually has to be formally closed before going away), but for other scenarios that could be a big drawback. In the section called “Object Attributes”, we will see how to solve this problem. For now, we will ignore it.

Figure 21.3, “Accounts using a dual representation” shows again an implementation for accounts, this time using a dual representation.

Figure 21.3. Accounts using a dual representation

```
local balance = {}

Account = {}

function Account:withdraw (v)
  balance[self] = balance[self] - v
end

function Account:deposit (v)
  balance[self] = balance[self] + v
end

function Account:balance ()
  return balance[self]
end

function Account:new (o)
  o = o or {}     -- create table if user does not provide one
  setmetatable(o, self)
  self.__index = self
  balance[o] = 0     -- initial balance
  return o
end
```

We use this class just like any other one:

```
a = Account:new{}
a:deposit(100.00)
print(a:balance())
```

However, we cannot tamper with an account balance. By keeping the table `balance` private to the module, this implementation ensures its safety.

Inheritance works without modifications. This approach has a cost quite similar to the standard one, both in terms of time and of memory. New objects need one new table and one new entry in each private table being used. The access `balance[self]` can be slightly slower than `self.balance`, because the latter uses a local variable while the first uses an external variable. Usually this difference is negligible. As we will see later, it also demands some extra work from the garbage collector.

Exercises

Exercise 21.1: Implement a class `Stack`, with methods `push`, `pop`, `top`, and `isempty`.

Exercise 21.2: Implement a class `StackQueue` as a subclass of `Stack`. Besides the inherited methods, add to this class a method `insertbottom`, which inserts an element at the bottom of the stack. (This method allows us to use objects of this class as queues.)

Exercise 21.3: Reimplement your `Stack` class using a dual representation.

Exercise 21.4: A variation of the dual representation is to implement objects using proxies (the section called “Tracking table accesses”). Each object is represented by an empty proxy table. An internal table maps proxies to tables that carry the object state. This internal table is not accessible from the outside, but methods use it to translate their `self` parameters to the real tables where they operate. Implement the Account example using this approach and discuss its pros and cons.

Chapter 22. The Environment

Global variables are a necessary evil of most programming languages. On one hand, the use of global variables can easily lead to complex code, entangling apparently unrelated parts of a program. On the other hand, the judicious use of global variables can better express truly global aspects of a program; moreover, global constants are innocuous, but dynamic languages like Lua have no way to distinguish constants from variables. An embedded language like Lua adds another ingredient to this mix: a global variable is a variable that is visible in the whole program, but Lua has no clear concept of a program, having instead pieces of code (chunks) called by the host application.

Lua solves this conundrum by not having global variables, but going to great lengths to pretend it has. In a first approximation, we can think that Lua keeps all its global variables in a regular table, called the *global environment*. Later in this chapter, we will see that Lua can keep its "global" variables in several environments. For now, we will stick to that first approximation.

The use of a table to store global variables simplifies the internal implementation of Lua, because there is no need for a different data structure for global variables. Another advantage is that we can manipulate this table like any other table. To help such manipulations, Lua stores the global environment itself in the global variable `_G`. (As a result, `_G._G` is equal to `_G`.) For instance, the following code prints the names of all the variables defined in the global environment:

```
for n in pairs(_G) do print(n) end
```

Global Variables with Dynamic Names

Usually, assignment is enough for accessing and setting global variables. However, sometimes we need some form of meta-programming, such as when we need to manipulate a global variable whose name is stored in another variable or is somehow computed at run time. To get the value of such a variable, some programmers are tempted to write something like this:

```
value = load("return " .. varname)()
```

If `varname` is `x`, for example, the concatenation will result in `"return x"`, which when run achieves the desired result. However, this code involves the creation and compilation of a new chunk, which is somewhat expensive. We can accomplish the same effect with the following code, which is more than an order of magnitude more efficient than the previous one:

```
value = _G[varname]
```

Because the environment is a regular table, we can simply index it with the desired key (the variable name).

In a similar way, we can assign a value to a global variable whose name is computed dynamically by writing `_G[varname] = value`. Beware, however: some programmers get a little excited with these facilities and end up writing code like `_G["a"] = _G["b"]`, which is just a complicated way to write `a = b`.

A generalization of the previous problem is to allow fields in the dynamic name, such as `"io.read"` or `"a.b.c.d"`. If we write `_G["io.read"]`, clearly we will not get the field `read` from the table `io`. But we can write a function `getfield` such that `getfield("io.read")` returns the expected result. This function is mainly a loop, which starts at `_G` and evolves field by field:

```
function getfield (f)
  local v = _G    -- start with the table of globals
```

```
  for w in string.gmatch(f, "[%a_][%w_]*") do
    v = v[w]
  end
  return v
end
```

We rely on `gmatch` to iterate over all identifiers in `f`.

The corresponding function to set fields is a little more complex. An assignment like `a.b.c.d = v` is equivalent to the following code:

```
local temp = a.b.c
temp.d = v
```

That is, we must retrieve up to the last name and then handle this last name separately. The function `setfield`, in Figure 22.1, "The function `setfield`", does the task and also creates intermediate tables in a path when they do not exist.

Figure 22.1. The function `setfield`

```
function setfield (f, v)
  local t = _G            -- start with the table of globals
  for w, d in string.gmatch(f, "([%a_][%w_]*)(%.?)") do
    if d == "." then      -- not last name?
      t[w] = t[w] or {}   -- create table if absent
      t = t[w]            -- get the table
    else                  -- last name
      t[w] = v            -- do the assignment
    end
  end
end
```

The pattern there captures the field name in the variable `w` and an optional following dot in the variable `d`. If a field name is not followed by a dot, then it is the last name.

With the previous functions in place, the next call creates a global table `t`, another table `t.x`, and assigns 10 to `t.x.y`:

```
setfield("t.x.y", 10)

print(t.x.y)               --> 10
print(getfield("t.x.y"))   --> 10
```

Global-Variable Declarations

Global variables in Lua do not need declarations. Although this behavior is handy for small programs, in larger programs a simple typo can cause bugs that are difficult to find. However, we can change this behavior if we like. Because Lua keeps its global variables in a regular table, we can use metatables to detect when Lua accesses non-existent variables.

A first approach simply detects any access to absent 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 this code, any attempt to access a non-existent global variable will trigger an error:

```
> print(a)
stdin:1: attempt to read undeclared variable a
```

But how do we declare new variables? One option is to use `rawset`, which bypasses the metamethod:

```
function declare (name, initval)
  rawset(_G, name, initval or false)
end
```

(The **or** with **false** ensures that the new global always gets a value different from nil.)

A simpler option is to restrict assignments to new global variables only inside functions, allowing free assignments in the outer level of a chunk.

To check whether an assignment is in the main chunk, we must use the debug library. The call `debug.getinfo(2, "S")` returns a table whose field `what` tells whether the function that called the metamethod is a main chunk, a regular Lua function, or a C function. (We will see `debug.getinfo` in more detail in the section called “Introspective Facilities”.) Using this function, we can rewrite the `__newindex` metamethod like this:

```
  __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 accepts assignments from C code, as this kind of code usually knows what it is doing.

If we need to test whether a variable exists, we cannot simply compare it to nil because, if it is nil, the access will raise an error. Instead, we use `rawget`, which avoids the metamethod:

```
if rawget(_G, var) == nil then
  -- 'var' is undeclared
  ...
end
```

As it is, our scheme does not allow global variables with nil values, as they would be automatically considered undeclared. But it is not difficult to correct this problem. All we need is an auxiliary table that keeps the names of declared variables. Whenever a metamethod is called, it checks in this table whether the variable is undeclared. The code can be like the one in Figure 22.2, “Checking global-variable declaration”.

Figure 22.2. Checking global-variable declaration

```
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,
})
```

Now, even an assignment like `x = nil` is enough to declare a global variable.

The overhead for both solutions is negligible. With the first solution, the metamethods are never called during normal operation. In the second, they can be called, but only when the program accesses a variable holding a nil.

The Lua distribution comes with a module `strict.lua` that implements a global-variable check that uses essentially the code in Figure 22.2, “Checking global-variable declaration”. It is a good habit to use it when developing Lua code.

Non-Global Environments

In Lua, global variables do not need to be truly global. As I already hinted, Lua does not even have global variables. That may sound strange at first, as we have been using global variables all along this text. As I said, Lua goes to great lengths to give the programmer an illusion of global variables. Now we will see how Lua builds this illusion.[1]

First, let us forget about global variables. Instead, we will start with the concept of free names. A *free name* is a name that is not bound to an explicit declaration, that is, it does not occur inside the scope of a corresponding local variable. For instance, both `x` and `y` are free names in the following chunk, but `z` is not:

```
local z = 10
x = y + z
```

Now comes the important part: The Lua compiler translates any free name `x` in the chunk to `_ENV.x`. So, the previous chunk is fully equivalent to this one:

```
local z = 10
_ENV.x = _ENV.y + z
```

[1]This mechanism was one of the parts of Lua that changed most from version 5.1 to 5.2. Very little of the following discussion applies to Lua 5.1.

But what is this new `_ENV` variable?

`_ENV` cannot be a global variable; we just said that Lua has no global variables. Again, the compiler does the trick. I already mentioned that Lua treats any chunk as an anonymous function. Actually, Lua compiles our original chunk as the following code:

```
local _ENV = some value
return function (...)
  local z = 10
  _ENV.x = _ENV.y + z
end
```

That is, Lua compiles any chunk in the presence of a predefined upvalue (an external local variable) called `_ENV`. So, any variable is either local, if it is a bounded name, or a field in `_ENV`, which itself is a local variable (an upvalue).

The initial value for `_ENV` can be any table. (Actually, it does not need to be a table; more about that later.) Any such table is called an environment. To preserve the illusion of global variables, Lua keeps internally a table that it uses as a *global environment*. Usually, when we load a chunk, the function `load` initializes this predefined upvalue with that global environment. So, our original chunk becomes equivalent to this one:

```
local _ENV = the global environment
return function (...)
  local z = 10
  _ENV.x = _ENV.y + z
end
```

The result of all these arrangements is that the `x` field of the global environment gets the value of the `y` field plus 10.

At first sight, this may seem a rather convoluted way to manipulate global variables. I will not argue that it is the simplest way, but it offers a flexibility that is difficult to achieve with a simpler implementation.

Before we go on, let us summarize the handling of global variables in Lua:

- The compiler creates a local variable `_ENV` outside any chunk that it compiles.
- The compiler translates any free name `var` to `_ENV.var`.
- The function `load` (or `loadfile`) initializes the first upvalue of a chunk with the global environment, which is a regular table kept internally by Lua.

After all, it is not that complicated.

Some people get confused because they try to infer extra magic from these rules. There is no extra magic. In particular, the first two rules are done entirely by the compiler. Except for being predefined by the compiler, `_ENV` is a plain regular variable. Outside the compiler, the name `_ENV` has no special meaning at all to Lua.[2] Similarly, the translation from `x` to `_ENV.x` is a plain syntactic translation, with no hidden meanings. In particular, after the translation, `_ENV` will refer to whatever `_ENV` variable is visible at that point in the code, following the standard visibility rules.

Using _ENV

In this section, we will see some ways to explore the flexibility brought by `_ENV`. Keep in mind that we must run most examples in this section as a single chunk. If we enter code line by line in interactive mode,

[2] To be completely honest, Lua uses that name for error messages, so that it reports an error involving a variable `_ENV.x` as being about `global x`.

each line becomes a different chunk and therefore each will have a distinct `_ENV` variable. To run a piece of code as a single chunk, we can either run it from a file or enclose it in a **do**--**end** block.

Because `_ENV` is a regular variable, we can assign to and access it as any other variable. The assignment `_ENV = nil` will invalidate any direct access to global variables in the rest of the chunk. This can be useful to control what variables our 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 (a "global variable") will raise a similar error.

We can write the `_ENV` explicitly to bypass a local declaration:

```
a = 13          -- global
local a = 12
print(a)        --> 12  (local)
print(_ENV.a)   --> 13  (global)
```

We can do the same with `_G`:

```
a = 13          -- global
local a = 12
print(a)        --> 12  (local)
print(_G.a)   --> 13  (global)
```

Usually, `_G` and `_ENV` refer to the same table but, despite that, they are quite different entities. `_ENV` is a local variable, and all accesses to "global variables" in reality are accesses to it. `_G` is a global variable with no special status whatsoever. By definition, `_ENV` always refers to the current environment; `_G` usually refers to the global environment, provided it is visible and no one changed its value.

The main use for `_ENV` is to change the environment used by a piece of code. Once we change the environment, all global accesses will use the new table:

```
-- change current environment to a new 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, we have lost all our global variables, including `print`. So, we should first populate it with some useful values, for instance with the global environment:

```
a = 15                      -- create a global variable
_ENV = {g = _G}             -- change current environment
a = 1                       -- create a field in _ENV
g.print(_ENV.a, g.a)        --> 1    15
```

Now, when we access the "global" `g` (which lives in `_ENV`, not in the global environment) we get the global environment, wherein Lua will find the function `print`.

We can rewrite the previous example using the name `_G` instead of `g`:

```
a = 15                      -- create a global variable
```

```
_ENV = {_G = _G}            -- change current environment
a = 1                       -- create a field in _ENV
_G.print(_ENV.a, _G.a)     --> 1    15
```

The only special status of `_G` happens when Lua creates the initial global table and makes its field `_G` points to itself. Lua does not care about the current value of this variable. Nevertheless, it is customary to use this same name whenever we have a reference to the global environment, as we did in the rewritten example.

Another way to populate our new environment is with inheritance:

```
a = 1
local newgt = {}        -- create new environment
setmetatable(newgt, {__index = _G})
_ENV = newgt            -- set it
print(a)                --> 1
```

In this code, the new environment inherits both `print` and `a` from the global one. However, any assignment goes to the new table. There is no danger of changing a variable in the global environment by mistake, although we still can change them through `_G`:

```
-- continuing the previous chunk
a = 10
print(a, _G.a)          --> 10    1
_G.a = 20
print(_G.a)             --> 20
```

Being a regular variable, `_ENV` follows the usual scoping rules. In particular, functions defined inside a chunk access `_ENV` as they access any other external variable:

```
_ENV = {_G = _G}
local function foo ()
  _G.print(a)     -- compiled as '_ENV._G.print(_ENV.a)'
end
a = 10
foo()             --> 10
_ENV = {_G = _G, a = 20}
foo()             --> 20
```

If we define a new local variable called `_ENV`, references to free names will bind to that new variable:

```
a = 2
do
  local _ENV = {print = print, a = 14}
  print(a)     --> 14
end
print(a)       --> 2   (back to the original _ENV)
```

Therefore, it is not difficult to define a function with a private environment:

```
function factory (_ENV)
  return function () return 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 their "global" `a`. When the closure is created, its visible `_ENV` variable is the parameter `_ENV` of the enclosing `factory` function; therefore, each closure will use its own external variable (as an upvalue) to access its free names.

Using the usual scoping rules, we can manipulate environments in several other ways. For instance, we may have several functions sharing a common environment, or a function that changes the environment that it shares with other functions.

Environments and Modules

In the section called "The Basic Approach for Writing Modules in Lua", when we discussed how to write modules, I mentioned that one drawback of those methods was that it was all too easy to pollute the global space, for instance by forgetting a **local** in a private declaration. Environments offer an interesting technique for solving that problem. Once the module main chunk has an exclusive environment, not only all its functions share this table, but also all its global variables go to this table. We can declare all public functions as global variables and they will go to a separate table automatically. All the module has to do is to assign this table to the `_ENV` variable. After that, when we declare a function `add`, it goes to `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 the same module without any prefix. In the previous code, `add` gets `new` from its environment, that is, it calls `M.new`.

This method offers a good support for modules, with little extra work for the programmer. It needs no prefixes at all. There is no difference between calling an exported function and a private one. If the programmer forgets a **local**, he does not pollute the global namespace; instead, a private function simply becomes public.

Nevertheless, currently I still prefer the original basic method. It may need more work, but the resulting code states clearly what it does. To avoid creating a global by mistake, I use the simple method of assigning nil to `_ENV`. After that, any assignment to a global name will raise an error. This approach has the extra advantage that it works without changes in older versions of Lua. (In Lua 5.1, the assignment to `_ENV` will not prevent errors, but it will not cause any harm, either.)

To access other modules, we can use one of the methods we discussed in the previous section. For instance, we can declare a local variable that holds the global environment:

```
local M = {}
local _G = _G
_ENV = nil
```

We then prefix global names with `_G` and module names with `M`.

A more disciplined approach is to declare as locals only the functions we need or, at most, the modules we need:

```
-- module setup
local M = {}
```

```
-- Import Section:
-- declare everything this module needs from outside
local sqrt = math.sqrt
local io = io

-- no more external access after this point
_ENV = nil
```

This technique demands more work, but it documents the module dependencies better.

_ENV and load

As I mentioned earlier, `load` usually initializes the `_ENV` upvalue of a loaded chunk with the global environment. However, `load` has an optional fourth parameter that allows us to give a different initial value for `_ENV`. (The function `loadfile` has a similar parameter.)

For an initial example, consider that we have a typical configuration file, defining several constants and functions to be used by a program; it can be something like this:

```
-- file 'config.lua'
width = 200
height = 300
...
```

We can load it with the following code:

```
env = {}
loadfile("config.lua", "t", env)()
```

The whole code in the configuration file will run in the empty environment `env`, which works as a kind of sandbox. In particular, all definitions will go into this environment. The configuration file has no way to affect anything else, even by mistake. Even malicious code cannot do much damage. It can do a denial of service (DoS) attack, by wasting CPU time and memory, but nothing else.

Sometimes, we may want to run a chunk several times, each time with a different environment table. In that case, the extra argument to `load` is not useful. Instead, we have two other options.

The first option is to use the function `debug.setupvalue`, from the debug library. As its name implies, `setupvalue` allows us to change any upvalue of a given function. The next fragment illustrates its use:

```
f = load("b = 10; return a")
env = {a = 20}
debug.setupvalue(f, 1, env)
print(f())              --> 20
print(env.b)            --> 10
```

The first argument in the call to `setupvalue` is the function, the second is the upvalue index, and the third is the new value for the upvalue. For this kind of use, the second argument is always one: when a function represents a chunk, Lua assures that it has only one upvalue and that this upvalue is `_ENV`.

A small drawback of this option is its dependence on the debug library. This library breaks some usual assumptions about programs. For instance, `debug.setupvalue` breaks Lua's visibility rules, which ensures that we cannot access a local variable from outside its lexical scope.

Another option to run a chunk with several different environments is to twist the chunk a little when loading it. Imagine that we add the following line just before the chunk:

```
_ENV = ...;
```

Remember that Lua compiles any chunk as a variadic function. So, that extra line of code will assign to the `_ENV` variable the first argument passed to the chunk, thereby setting that argument as the environment. The following code snippet illustrates the idea, using the function `loadwithprefix` that you implemented in Exercise 16.1:

```
prefix = "_ENV = ...;"
f = loadwithprefix(prefix, io.lines(filename, "*L"))
...
env1 = {}
f(env1)
env2 = {}
f(env2)
```

Exercises

Exercise 22.1: The function `getfield` that we defined in the beginning of this chapter is too forgiving, as it accepts "fields" like `math?sin` or `string!!!gsub`. Rewrite it so that it accepts only single dots as name separators.

Exercise 22.2: Explain in detail what happens in the following program and what it will print.

```
local foo
do
  local _ENV = _ENV
  function foo ()  print(X) end
end
X = 13
_ENV = nil
foo()
X = 0
```

Exercise 22.3: Explain in detail what happens in the following program and what it will print.

```
local print = print
function foo (_ENV, a)
  print(a + b)
end

foo({b = 14}, 12)
foo({b = 10}, 1)
```

Chapter 23. Garbage

Lua does automatic memory management. Programs can create objects (tables, closures, etc.), but there is no function to delete objects. Lua automatically deletes objects that become garbage, using *garbage collection*. This frees us from most of the burden of memory management and, more importantly, frees us from most of the bugs related to this activity, such as dangling pointers and memory leaks.

In an ideal world, the garbage collector would be invisible to the programmer, like a good cleaner that does not interfere with other workers. However, sometimes even the smarter collector needs our help. We may need to stop it at some performance-critical periods or allow it to work only in some specific times. Moreover, a garbage collector can collect only what it can be sure is garbage; it cannot guess what we consider garbage. No garbage collector allows us to forget all worries about resource management, such as hoarding memory and external resources.

Weak tables, finalizers, and the function `collectgarbage` are the main mechanisms that we can use in Lua to help the garbage collector. Weak tables allow the collection of Lua objects that are still accessible to the program; finalizers allow the collection of external objects that are not directly under control of the garbage collector. The function `collectgarbage` allows us to control the pace of the collector. In this chapter, we will discuss these mechanisms.

Weak Tables

As we said, a garbage collector cannot guess what we consider garbage. A typical example is a stack, implemented with an array and an index to the top. We know that the valid part of the array goes only up to the top, but Lua does not. If we pop an element by simply decrementing the top, the object left in the array is not garbage to Lua. Similarly, any object stored in a global variable is not garbage to Lua, even if our program will never use it again. In both cases, it is up to us (i.e., our program) to assign nil to these positions so that they do not lock an otherwise disposable object.

However, simply cleaning our references is not always enough. Some constructions need extra collaboration between the program and the collector. A typical example happens when we want to keep a list of all live objects of some kind (e.g., files) in our program. This task seems simple: all we have to do is to insert each new object into the list. However, once the object is part of the list, it will never be collected! Even if nothing else points to it, the list does. Lua cannot know that this reference should not prevent the reclamation of the object, unless we tell Lua about this fact.

Weak tables are the mechanism that we use to tell Lua that a reference should not prevent the reclamation of an object. A *weak reference* is a reference to an object that is not considered by the garbage collector. If all references pointing to an object are weak, the collector will collect the object and delete these weak references. Lua implements weak references through *weak tables*: a weak table is a table whose entries are weak. This means that, if an object is held only in weak tables, Lua will eventually collect the object.

Tables have keys and values, and both can contain any kind of object. Under normal circumstances, the garbage collector does not collect objects that appear as keys or as values of an accessible table. That is, both keys and values are *strong* references, as they prevent the reclamation of objects they refer to. In a weak table, both keys and values can be weak. This means that there are three kinds of weak tables: tables with weak keys, tables with weak values, and tables where both keys and values are weak. Irrespective of the kind of table, when a key or a value is collected the whole entry disappears from the table.

The weakness of a table is given by the field `__mode` of its metatable. The value of this field, when present, should be a string: if this string is `"k"`, the keys in the table are weak; if this string is `"v"`, the values in the table are weak; if this string is `"kv"`, both keys and values are weak. The following example, although artificial, illustrates the basic behavior of weak tables:

```
a = {}
mt = {__mode = "k"}
setmetatable(a, mt)    -- now 'a' has weak keys
key = {}               -- creates first key
a[key] = 1
key = {}               -- creates second key
a[key] = 2
collectgarbage()       -- forces a garbage collection cycle
for k, v in pairs(a) do print(v) end
  --> 2
```

In this example, the second assignment `key = {}` overwrites the reference to the first key. The call to `collectgarbage` forces the garbage collector to do a full collection. As there is no other reference to the first key, Lua collects this key and removes the corresponding entry in the table. The second key, however, is still anchored in the variable `key`, so Lua does not collect it.

Notice that only objects can be removed from a weak table. Values, such as numbers and Booleans, are not collectible. For instance, if we insert a numeric key in the table `a` (from our previous example), the collector will never remove it. Of course, if the value corresponding to a numeric key is collected in a table with weak values, then the whole entry is removed from the table.

Strings present a subtlety here: although strings are collectible from an implementation point of view, they are not like other collectible objects. Other objects, such as tables and closures, are created explicitly. For instance, whenever Lua evaluates the expression `{}`, it creates a new table. However, does Lua create a new string when it evaluates `"a".."b"`? What if there is already a string `"ab"` in the system? Does Lua create a new one? Can the compiler create this string before running the program? It does not matter: these are implementation details. From the programmer's point of view, strings are values, not objects. Therefore, like a number or a Boolean, a string key is not removed from a weak table unless its associated value is collected.

Memorize Functions

A common programming technique is to trade space for time. We can speed up a function by *memorizing* its results so that, later, when we call the function with the same argument, the function can reuse that result.[1]

Imagine a generic server that takes requests in the form of strings with Lua code. Each time it gets a request, it runs `load` on the string, and then calls the resulting function. However, `load` is an expensive function, and some commands to the server may be quite frequent. Instead of calling `load` repeatedly each time it receives a common command like `"closeconnection()"`, the server can *memorize* the results from `load` using an auxiliary table. Before calling `load`, the server checks in the table whether the given string already has a translation. If it cannot find a match, then (and only then) the server calls `load` and stores the result into the table. We can pack this behavior in a new function:

```
local results = {}
function mem_loadstring (s)
  local res = results[s]
  if res == nil then                  -- result not available?
    res = assert(load(s))             -- compute new result
    results[s] = res                  -- save for later reuse
  end
  return res
end
```

[1] Although the established English word “memorize” describes precisely what we want to do, the programming community created a new word, *memoize*, to describe this technique. I will stick to the original word.

The savings with this scheme can be huge. However, it may also cause unsuspected waste. Although some commands repeat over and over, many other commands happen only once. Gradually, the table `results` accumulates all commands the server has ever received plus their respective codes; after enough time, this behavior will exhaust the server's memory.

A weak table provides a simple solution to this problem. If the `results` table has weak values, each garbage-collection cycle will remove all translations not in use at that moment (which means virtually all of them):

```
local results = {}
setmetatable(results, {__mode = "v"})  -- make values weak
function mem_loadstring (s)
   as before
```

Actually, because the indices are always strings, we can make this table fully weak, if we want:

```
setmetatable(results, {__mode = "kv"})
```

The net result is the same.

The memorization technique is useful also to ensure the uniqueness of some kind of object. For instance, assume a system that represents colors as tables, with fields `red`, `green`, and `blue` in some range. A naive color factory generates a new color for each new request:

```
function createRGB (r, g, b)
  return {red = r, green = g, blue = b}
end
```

Using memorization, we can reuse the same table for the same color. To create a unique key for each color, we simply concatenate the color indices with a separator in between:

```
local results = {}
setmetatable(results, {__mode = "v"})  -- make values weak
function createRGB (r, g, b)
  local key = string.format("%d-%d-%d", 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 user can compare colors using the primitive equality operator, because two coexistent equal colors are always represented by the same table. Any given color can be represented by different tables at different times, because from time to time the garbage collector clears the `results` table. However, as long as a given color is in use, it is not removed from `results`. So, whenever a color survives long enough to be compared with a new one, its representation also has survived long enough to be reused by the new color.

Object Attributes

Another important use of weak tables is to associate attributes with objects. There are endless situations where we need to attach some attribute to an object: names to functions, default values to tables, sizes to arrays, and so on.

When the object is a table, we can store the attribute in the table itself, with an appropriate unique key. (As we saw before, a simple and error-proof way to create a unique key is to create a new table and use it as the key.) However, if the object is not a table, it cannot keep its own attributes. Even for tables, sometimes we may not want to store the attribute in the original object. For instance, we may want to keep the attribute private, or we do not want the attribute to disturb a table traversal. In all these cases, we need an alternative way to map attributes to objects.

Of course, an external table provides an ideal way to map attributes to objects. It is what we called a *dual representation* in the section called “Dual Representation”. We use the objects as keys, and their attributes as values. An external table can keep attributes of any type of object, as Lua allows us to use any type of object as a key. Moreover, attributes kept in an external table do not interfere with other objects, and can be as private as the table itself.

However, this seemingly perfect solution has a huge drawback: once we use an object as a key in a table, we lock the object into existence. Lua cannot collect an object that is being used as a key. For instance, if we use a regular table to map functions to its names, none of these functions will ever be collected. As you might expect, we can avoid this drawback by using a weak table. This time, however, we need weak keys. The use of weak keys does not prevent any key from being collected, once there are no other references to it. On the other hand, the table cannot have weak values; otherwise, attributes of live objects could be collected.

Revisiting Tables with Default Values

In the section called “Tables with default values”, we discussed how to implement tables with non-nil default values. We saw one particular technique and commented that two other techniques needed weak tables, so we postponed them. Now it is time to revisit the subject. As we will see, these two techniques for default values are actually particular applications of the two general techniques that we have just discussed: dual representation and memorization.

In the first solution, we use a weak table to map each table to its default value:

```
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
```

This is a typical use of a dual representation, where we use `defaults[t]` to represent `t.default`. If the table `defaults` did not have weak keys, it would anchor all tables with default values into permanent existence.

In the second solution, we use distinct metatables for distinct default values, but we reuse the same metatable whenever we repeat a default value. This is a typical use of memorization:

```
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     -- memorize
  end
  setmetatable(t, mt)
```

```
end
```

In this case, we use weak values to allow the collection of metatables that are not being used anymore.

Given these two implementations for default values, which is best? As usual, it depends. Both have similar complexity and similar performance. The first implementation needs a few memory words for each table with a default value (an entry in `defaults`). The second implementation needs a few dozen memory words for each distinct default value (a new table, a new closure, plus an entry in the table `metas`). So, if your application has thousands of tables with a few distinct default values, the second implementation is clearly superior. On the other hand, if few tables share common defaults, then you should favor the first implementation.

Ephemeron Tables

A tricky situation occurs when, in a table with weak keys, a value refers to its own key.

This scenario is more common than it may seem. As a typical example, consider a constant-function factory. Such a factory takes an object and returns a function that, whenever called, returns that object:

```
function factory (o)
  return (function () return o end)
end
```

This factory is a good candidate for memorization, to avoid the creation of a new closure when there is one already available. Figure 23.1, “Constant-function factory with memorization” shows this improvement.

Figure 23.1. Constant-function factory with memorization

```
do
  local mem = {}     -- memorization table
  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
```

There is a catch, however. Note that the value (the constant function) associated with an object in `mem` refers back to its own key (the object itself). Although the keys in that table are weak, the values are not. From a standard interpretation of weak tables, nothing would ever be removed from that memorizing table. Because values are not weak, there is always a strong reference to each function. Each function refers to its corresponding object, so there is always a strong reference to each key. Therefore, these objects would not be collected, despite the weak keys.

This strict interpretation, however, is not very useful. Most people expect that a value in a table is only accessible through its respective key. We can think of the above scenario as a kind of cycle, where the closure refers to the object that refers back (through the memorizing table) to the closure.

Lua solves the above problem with the concept of ephemeron tables.[2] In Lua, a table with weak keys and strong values is an *ephemeron table*. In an ephemeron table, the accessibility of a key controls the

[2]Ephemeron tables were introduced in Lua 5.2. Lua 5.1 still has the problem we described.

accessibility of its corresponding value. More specifically, consider an entry *(k,v)* in an ephemeron table. The reference to *v* is only strong if there is some other external reference to *k*. Otherwise, the collector will eventually collect *k* and remove the entry from the table, even if *v* refers (directly or indirectly) to *k*.

Finalizers

Although the goal of the garbage collector is to collect Lua objects, it can also help programs to release external resources. For that purpose, several programming languages offer finalizers. A *finalizer* is a function associated with an object that is called when that object is about to be collected.

Lua implements finalizers through the metamethod `__gc`, as the following example illustrates:

```
o = {x = "hi"}
setmetatable(o, {__gc = function (o) print(o.x) end})
o = nil
collectgarbage()   --> hi
```

In this example, we first create a table and give it a metatable that has a `__gc` metamethod. Then we erase the only link to the table (the global variable `o`) and force a complete garbage collection. During the collection, Lua detects that the table is no longer accessible, and therefore calls its finalizer —the `__gc` metamethod.

A subtlety of finalizers in Lua is the concept of marking an object for finalization. We mark an object for finalization by setting a metatable for it with a non-null `__gc` metamethod. If we do not mark the object, it will not be finalized. Most code we write works naturally, but some strange cases can occur, like here:

```
o = {x = "hi"}
mt = {}
setmetatable(o, mt)
mt.__gc = function (o) print(o.x) end
o = nil
collectgarbage()   --> (prints nothing)
```

Here, the metatable we set for `o` does not have a `__gc` metamethod, so the object is not marked for finalization. Even if we later add a `__gc` field to the metatable, Lua does not detect that assignment as something special, so it will not mark the object.

As we said, this is seldom a problem; it is not usual to change metamethods after setting a metatable. If you really need to set the metamethod later, you can provide any value for the `__gc` field, as a placeholder:

```
o = {x = "hi"}
mt = {__gc = true}
setmetatable(o, mt)
mt.__gc = function (o) print(o.x) end
o = nil
collectgarbage()   --> hi
```

Now, because the metatable has a `__gc` field, `o` is properly marked for finalization. There is no problem if you do not set a metamethod later; Lua only calls the finalizer if it is a proper function.

When the collector finalizes several objects in the same cycle, it calls their finalizers in the reverse order that the objects were marked for finalization. Consider the next example, which 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 is object 3, which was the last to be marked.

A common misconception is to think that links among objects being collected can affect the order that they are finalized. For instance, one can think that object 2 in the previous example must be finalized before object 1 because there is a link from 2 to 1. However, links can form cycles. Therefore, they do not impose any order to finalizers.

Another tricky point about finalizers is *resurrection*. When a finalizer is called, it gets the object being finalized as a parameter. So, the object becomes alive again, at least during the finalization. I call this a *transient resurrection*. While the finalizer runs, nothing stops it from storing the object in a global variable, for instance, so that it remains accessible after the finalizer returns. I call this a *permanent resurrection*.

Resurrection must be transitive. Consider the following piece of code:

```
A = {x = "this is A"}
B = {f = A}
setmetatable(B, {__gc = function (o) print(o.f.x) end})
A, B = nil
collectgarbage()   --> this is A
```

The finalizer for `B` accesses `A`, so `A` cannot be collected before the finalization of `B`. Lua must resurrect both `B` and `A` before running that finalizer.

Because of resurrection, Lua collects objects with finalizers in two phases. The first time the collector detects that an object with a finalizer is not reachable, the collector resurrects the object and queues it to be finalized. Once its finalizer runs, Lua marks the object as finalized. The next time the collector detects that the object is not reachable, it deletes the object. If we want to ensure that all garbage in our program has been actually released, we must call `collectgarbage` twice; the second call will delete the objects that were finalized during the first call.

The finalizer for each object runs exactly once, due to the mark that Lua puts on finalized objects. If an object is not collected until the end of a program, Lua will call its finalizer when the entire Lua state is closed. This last feature allows a form of `atexit` functions in Lua, that is, functions that will run immediately before the program terminates. All we have to do is to create a table with a finalizer and anchor it somewhere, for instance in a global variable:

```
local t = {__gc = function ()
  -- your 'atexit' code comes here
  print("finishing Lua program")
end}
setmetatable(t, t)
_G["*AA*"] = t
```

Another interesting technique allows a program to call a given function every time Lua completes a collection cycle. As a finalizer runs only once, the trick here is to make each finalization create a new object to run the next finalizer, as in Figure 23.2, “Running a function at every GC cycle”.

Figure 23.2. Running a function at every GC cycle

```
do
  local mt = {__gc = function (o)
    -- whatever you want to do
    print("new cycle")
    -- creates new object for next cycle
    setmetatable({}, getmetatable(o))
  end}
  -- creates first object
  setmetatable({}, mt)
end

collectgarbage()   --> new cycle
collectgarbage()   --> new cycle
collectgarbage()   --> new cycle
```

The interaction of objects with finalizers and weak tables also has a subtlety. At each cycle, the collector clears the values in weak tables before calling the finalizers, but it clears the keys after it. The rationale for this behavior is that frequently we use tables with weak keys to hold properties of an object (as we discussed in the section called “Object Attributes”), and therefore finalizers may need to access those attributes. However, we use tables with weak values to reuse live objects; in this case, objects being finalized are not useful anymore.

The Garbage Collector

Up to version 5.0, Lua used a simple mark-and-sweep garbage collector (GC). This kind of collector is sometimes called a “stop-the-world” collector. This means that, from time to time, Lua would stop running the main program to perform a whole garbage-collection cycle. Each cycle comprises four phases: *mark*, *cleaning*, *sweep*, and *finalization*.

The collector starts the mark phase by marking as alive its *root set*, which comprises the objects that Lua has direct access to. In Lua, this set is only the C registry. (As we will see in the section called “The registry”, both the main thread and the global environment are predefined entries in this registry.)

Any object stored in a live object is reachable by the program, and therefore is marked as alive too. (Of course, entries in weak tables do not follow this rule.) The mark phase ends when all reachable objects are marked as alive.

Before starting the sweep phase, Lua performs the cleaning phase, where it handles finalizers and weak tables. First, it traverses all objects marked for finalization looking for non-marked objects. Those objects are marked as alive (resurrected) and put in a separate list, to be used in the finalization phase. Then, Lua traverses its weak tables and removes from them all entries wherein either the key or the value is not marked.

The sweep phase traverses all Lua objects. (To allow this traversal, Lua keeps all objects it creates in a linked list.) If an object is not marked as alive, Lua collects it. Otherwise, Lua clears its mark, in preparation for the next cycle.

Finally, in the finalization phase, Lua calls the finalizers of the objects that were separated in the cleaning phase.

The use of a real garbage collector means that Lua has no problems with cycles among object references. We do not need to take any special action when using cyclic data structures; they are collected like any other data.

In version 5.1, Lua got an *incremental collector*. This collector performs the same steps as the old one, but it does not need to stop the world while it runs. Instead, it runs interleaved with the interpreter. Every time the interpreter allocates some amount of memory, the collector runs a small step. (This means that, while the collector is working, the interpreter may change an object's reachability. To ensure the correctness of the collector, some operations in the interpreter have barriers that detect dangerous changes and correct the marks of the objects involved.)

Lua 5.2 introduced *emergency collection*. When a memory allocation fails, Lua forces a full collection cycle and tries again the allocation. These emergencies can occur any time Lua allocates memory, including points where Lua is not in a consistent state to run code; so, these collections are unable to run finalizers.

Controlling the Pace of Collection

The function `collectgarbage` allows us to exert some control over the garbage collector. It is actually several functions in one: its optional first argument, a string, specifies what to do. Some options have an integer as a second argument, which we call `data`.

The options for the first argument are:

`"stop"`: stops the collector until another call to `collectgarbage` with the option `"restart"`.

`"restart"`: restarts the collector.

`"collect"`: performs a complete garbage-collection cycle, so that all unreachable objects are collected and finalized. This is the default option.

`"step"`: performs some garbage-collection work. The second argument, `data`, specifies the amount of work, which is equivalent to what the collector would do after allocating `data` bytes.

`"count"`: returns the number of kilobytes of memory currently in use by Lua. This result is a floating-point number that multiplied by 1024 gives the exact total number of bytes. The count includes dead objects that have not yet been collected.

`"setpause"`: sets the collector's `pause` parameter. The `data` parameter gives the new value in percentage points: when `data` is 100, the parameter is set to 1 (100%).

`"setstepmul"`: sets the collector's step multiplier (`stepmul`) parameter. The new value is given by `data`, also in percentage points.

The two parameters `pause` and `stepmul` control the collector's character. Any garbage collector trades memory for CPU time. At one extreme, the collector might not run at all. It would spend zero CPU time, at the price of a huge memory consumption. At the other extreme, the collector might run a complete cycle after every single assignment. The program would use the minimum memory necessary, at the price of a huge CPU consumption. The default values for `pause` and `stepmul` try to find a balance between those two extremes and are good enough for most applications. In some scenarios, however, it is worth trying to optimize them.

The `pause` parameter controls how long the collector waits between finishing a collection and starting a new one. A pause of zero makes Lua start a new collection as soon as the previous one ends. A pause of 200% waits for memory usage to double before restarting the collector. We can set a lower pause if we want to trade more CPU time for lower memory usage. Typically, we should keep this value between 0 and 200%.

The step-multiplier parameter (`stepmul`) controls how much work the collector does for each kilobyte of memory allocated. The higher this value the less incremental the collector. A huge value like 100000000% makes the collector work like a non-incremental collector. The default value is 200%. Values lower than 100% make the collector so slow that it may never finish a collection.

The other options of `collectgarbage` give us control over when the collector runs. Again, the default control is good enough for most programs, but some specific applications may benefit from a manual control. Games often need this kind of control. For instance, if we do not want any garbage-collection work during some periods, we can stop it with a call `collectgarbage("stop")` and then restart it with `collectgarbage("restart")`. In systems where we have periodic idle phases, we can keep the collector stopped and call `collectgarbage("step", n)` during the idle time. To set how much work to do at each idle period, we can either choose experimentally an appropriate value for `n` or call `collectgarbage` in a loop, with `n` set to zero (meaning minimal steps), until the idle period expires.

Exercises

Exercise 23.1: Write an experiment to determine whether Lua actually implements ephemeron tables. (Remember to call `collectgarbage` to force a garbage collection cycle.) If possible, try your code both in Lua 5.1 and in Lua 5.2/5.3 to see the difference.

Exercise 23.2: Consider the first example of the section called “Finalizers”, which creates a table with a finalizer that only prints a message when activated. What happens if the program ends without a collection cycle? What happens if the program calls `os.exit`? What happens if the program ends with an error?

Exercise 23.3: Imagine you have to implement a memorizing table for a function from strings to strings. Making the table weak will not do the removal of entries, because weak tables do not consider strings as collectable objects. How can you implement memorization in that case?

Exercise 23.4: Explain the output of the program in Figure 23.3, “Finalizers and memory”.

Figure 23.3. Finalizers and memory

```
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)
```

Exercise 23.5: For this exercise, you need at least one Lua script that uses lots of memory. If you do not have one, write it. (It can be as simple as a loop creating tables.)

- Run your script with different values for `pause` and `stepmul`. How they affect the performance and memory usage of the script? What happens if you set the pause to zero? What happens if you set the

pause to 1000? What happens if you set the step multiplier to zero? What happens if you set the step multiplier to 1000000?

- Adapt your script so that it keeps full control over the garbage collector. It should keep the collector stopped and call it from time to time to do some work. Can you improve the performance of your script with this approach?

Chapter 24. Coroutines

We do not need coroutines very often, but when we do, it is an unparalleled feature. Coroutines can literally turn upside-down the relationship between callers and callees, and this flexibility solves what I call the "who-is-the-boss" (or "who-has-the-main-loop") problem in software architecture. This is a generalization of several seemingly unrelated problems, such as entanglement in event-driven programs, building iterators through generators, and cooperative multithreading. Coroutines solve all these problems in simple and efficient ways.

A *coroutine* is similar to a thread (in the sense of multithreading): it is a line of execution, with its own stack, its own local variables, and its own instruction pointer; it shares global variables and mostly anything else with other coroutines. The main difference between threads and coroutines is that a multithreaded program runs several threads in parallel, while coroutines are collaborative: at any given time, a program with coroutines is running only one of its coroutines, and this running coroutine suspends its execution only when it explicitly requests to be suspended.

In this chapter we will cover how coroutines work in Lua and how we can use them to solve a diverse set of problems.

Coroutine Basics

Lua packs all its coroutine-related functions in the table `coroutine`. The function `create` creates new coroutines. It has a single argument, a function with the code that the coroutine will run (the coroutine *body*). It returns a value of type `"thread"`, which represents the new coroutine. Often, the argument to `create` is an anonymous function, like here:

```
co = coroutine.create(function () print("hi") end)
print(type(co))           --> thread
```

A coroutine can be in one of four states: suspended, running, normal, and dead. We can check the state of a coroutine with the function `coroutine.status`:

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

When we create a coroutine, it starts in the suspended state; a coroutine does not run its body automatically when we create it. The function `coroutine.resume` (re)starts the execution of a coroutine, changing its state from suspended to running:

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

(If you run this code in interactive mode, you may want to finish the previous line with a semicolon, to suppress the display of the result from `resume`.) In this first example, the coroutine body simply prints `"hi"` and terminates, leaving the coroutine in the dead state:

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

Until now, coroutines look like nothing more than a complicated way to call functions. The real power of coroutines stems from the function `yield`, which allows a running coroutine to suspend its own execution so that it can be resumed later. Let us see a simple example:

```
co = coroutine.create(function ()
       for i = 1, 10 do
```

```
      print("co", i)
      coroutine.yield()
    end
  end)
```

Now, the coroutine body does a loop, printing numbers and yielding after each print. When we resume this coroutine, it starts its execution and runs until the first `yield`:

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

If we check its status, we can see that the coroutine is suspended and, therefore, can be resumed:

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

From the coroutine's point of view, all activity that happens while it is suspended is happening inside its call to `yield`. When we resume the coroutine, this call to `yield` finally returns and the coroutine continues its execution until the next yield or until its end:

```
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 coroutine body finishes the loop and then returns, without printing anything. If we try to resume it again, `resume` returns **false** plus an error message:

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

Note that `resume` runs in protected mode, like `pcall`. Therefore, if there is any error inside a coroutine, Lua will not show the error message, but instead will return it to the `resume` call.

When a coroutine resumes another, it is not suspended; after all, we cannot resume it. However, it is not running either, because the running coroutine is the other one. So, its own status is what we call the *normal* state.

A useful facility in Lua is that a pair resume–yield can exchange data. The first `resume`, which has no corresponding `yield` waiting for it, passes its extra arguments to the coroutine main 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
```

A call to `coroutine.resume` returns, after the **true** that signals no errors, any arguments passed to the corresponding `yield`:

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

Symmetrically, `coroutine.yield` returns any extra arguments passed to the corresponding `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 a coroutine ends, any values returned by its main function go to the corresponding `resume`:

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

We seldom use all these facilities in the same coroutine, but all of them have their uses.

Although the general concept of coroutines is well understood, the details vary considerably. So, for those that already know something about coroutines, it is important to clarify these details before we go on. Lua offers what we call *asymmetric coroutines*. This means that it has a function to suspend the execution of a coroutine and a different function to resume a suspended coroutine. Some other languages offer *symmetric coroutines*, where there is only one function to transfer control from one coroutine to another.

Some people call asymmetric coroutines *semi-coroutines*. However, other people use the same term *semi-coroutine* to denote a restricted implementation of coroutines, where a coroutine can suspend its execution only when it is not calling any function, that is, when it has no pending calls in its control stack. In other words, only the main body of such semi-coroutines can yield. (A *generator* in Python is an example of this meaning of semi-coroutines.)

Unlike the difference between symmetric and asymmetric coroutines, the difference between coroutines and generators (as presented in Python) is a deep one; generators are simply not powerful enough to implement some of the most interesting constructions that we can write with full coroutines. Lua offers full, asymmetric coroutines. Those that prefer symmetric coroutines can implement them on top of the asymmetric facilities of Lua (see Exercise 24.6).

Who Is the Boss?

One of the most paradigmatic examples of coroutines is the producer–consumer problem. Let us suppose that we have a function that continually produces values (e.g., reading them from a file) and another function that continually consumes these values (e.g., writing them to another file). These two functions could look like this:

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

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

```
end
```

(To simplify this example, both the producer and the consumer run forever. It is not hard to change them to stop when there is no more data to handle.) The problem here is how to match `send` with `receive`. It is a typical instance of the “who-has-the-main-loop” problem. Both the producer and the consumer are active, both have their own main loops, and both assume that the other is a callable service. For this particular example, it is easy to change the structure of one of the functions, unrolling its loop and making it a passive agent. However, this change of structure may be far from easy in other real scenarios.

Coroutines provide an ideal tool to match producers and consumers without changing their structure, because a resume–yield pair turns upside-down the typical relationship between the caller and its callee. When a coroutine calls `yield`, it does not enter into a new function; instead, it returns a pending call (to `resume`). Similarly, a call to `resume` does not start a new function, but returns a call to `yield`. This property is exactly what we need to match a `send` with a `receive` in such a way that each one acts as if it were the master and the other the slave. (That is why I called this the "who-is-the-boss" problem.) So, `receive` resumes the producer, so that it can produce a new value; and `send` yields the new 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 now run inside a coroutine:

```
producer = coroutine.create(producer)
```

In this design, the program starts by calling the consumer. When the consumer needs an item, it resumes the producer, which runs until it has an item to give to the consumer, and then stops until the consumer resumes it again. Therefore, we have what we call a *consumer-driven* design. Another way to write the program is to use a *producer-driven* design, where the consumer is the coroutine. Although the details seem reversed, the overall idea of both designs is the same.

We can extend this design with filters, which are tasks that sit between the producer and the consumer doing some kind of transformation in the data. A *filter* is a consumer and a producer at the same time, so it resumes a producer to get new values and yields the transformed values to a consumer. As a trivial example, we can add to our previous code a filter that inserts a line number at the beginning of each line. The code is in Figure 24.1, “Producer–consumer with filters”.

Figure 24.1. Producer–consumer 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 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

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

Its last line simply creates the components it needs, connects them, and starts the final consumer.

If you thought about POSIX pipes after reading the previous example, you are not alone. After all, coroutines are a kind of (non-preemptive) multithreading. With pipes, each task runs in a separate process; with coroutines, each task runs in a separate coroutine. Pipes provide a buffer between the writer (producer) and the reader (consumer) so there is some freedom in their relative speeds. This is important in the context of pipes, because the cost of switching between processes is high. With coroutines, the cost of switching between tasks is much smaller (roughly equivalent to a function call), so the writer and the reader can run hand in hand.

Coroutines as Iterators

We can see loop iterators as a particular example of the producer–consumer pattern: an iterator produces items to be consumed by the loop body. Therefore, it seems appropriate to use coroutines to write iterators. Indeed, coroutines provide a powerful tool for this task. Again, the key feature is their ability to turn inside

out the relationship between caller and callee. With this feature, we can write iterators without worrying about how to keep state between successive calls.

To illustrate this kind of use, let us write an iterator to traverse all permutations of a given array. It is not an easy task to write directly such an iterator, but it is not so difficult to write a recursive function that generates all these permutations. The idea is simple: put each array element in the last position, in turn, and recursively generate all permutations of the remaining elements. The code is in Figure 24.2, “A function to generate permutations”.

Figure 24.2. A function to generate permutations

```
function permgen (a, n)
  n = n or #a          -- default for 'n' is size of 'a'
  if n <= 1 then       -- nothing to change?
    printResult(a)
  else
    for i = 1, n do

      -- put i-th element as the last one
      a[n], a[i] = a[i], a[n]

      -- generate all permutations of the other elements
      permgen(a, n - 1)

      -- restore i-th element
      a[n], a[i] = a[i], a[n]
    end
  end
end
```

To put it to work, we must define an appropriate `printResult` function 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
```

After we have the generator ready, it is an automatic task to convert it to an iterator. First, we change `printResult` to `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 arranges for the generator to run inside a coroutine and creates the iterator function. The iterator simply resumes the coroutine to produce the next permutation:

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

With this machinery in place, it is trivial to iterate over all permutations of an array with a **for** statement:

```
for p in permutations{"a", "b", "c"} do
  printResult(p)
end
  --> b c a
  --> c b a
  --> c a b
  --> a c b
  --> b a c
  --> a b c
```

The function `permutations` uses a common pattern in Lua, which packs a call to resume with its corresponding coroutine inside a function. This pattern is so common that Lua provides a special function for it: `coroutine.wrap`. Like `create`, `wrap` creates a new coroutine. Unlike `create`, `wrap` does not return the coroutine itself; instead, it returns a function that, when called, resumes the coroutine. Unlike the original `resume`, that function does not return an error code as its first result; instead, it raises the error in case of error. Using `wrap`, we can write `permutations` as follows:

```
function permutations (a)
  return coroutine.wrap(function () permgen(a) end)
end
```

Usually, `coroutine.wrap` is simpler to use than `coroutine.create`. It gives us exactly what we need from a coroutine: a function to resume it. However, it is also less flexible. There is no way to check the status of a coroutine created with `wrap`. Moreover, we cannot check for runtime errors.

Event-Driven Programming

It may not be obvious at first sight, but the typical entanglement created by conventional event-driven programming is another consequence of the who-is-the-boss problem.

In a typical event-driven platform, an external entity generates events to our program in a so-called *event loop* (or *run loop*). It is clear who is the boss there, and it is not our code. Our program becomes a slave of the event loop, and that makes it a collection of individual event handlers without any apparent connection.

To make things a little more concrete, let us assume that we have an asynchronous I/O library similar to `libuv`. The library has four functions that concern our small example:

```
  lib.runloop();
  lib.readline(stream, callback);
  lib.writeline(stream, line, callback);
  lib.stop();
```

The first function runs the event loop, which will process the incoming events and call the associated callbacks. A typical event-driven program initializes some stuff and then calls this function, which becomes the main loop of the application. The second function instructs the library to read a line of the given stream and, when it is done, to call the given callback function with the result. The third function is similar to the second, but for writing a line. The last function breaks the event loop, usually to finish the program.

Figure 24.3, “An ugly implementation of the asynchronous I/O library” presents an implementation for such a library.

Figure 24.3. An ugly implementation of the asynchronous I/O library

```
local cmdQueue = {}     -- queue of pending operations

local lib = {}

function lib.readline (stream, callback)
  local nextCmd = function ()
    callback(stream:read())
  end
  table.insert(cmdQueue, nextCmd)
end

function lib.writeline (stream, line, callback)
  local nextCmd = function ()
    callback(stream:write(line))
  end
  table.insert(cmdQueue, nextCmd)
end

function lib.stop ()
  table.insert(cmdQueue, "stop")
end

function lib.runloop ()
  while true do
    local nextCmd = table.remove(cmdQueue, 1)
    if nextCmd == "stop" then
      break
    else
      nextCmd()       -- perform next operation
    end
  end
end

return lib
```

It is a very ugly implementation. Its “event queue” is in fact a list of pending operations that, when performed (synchronously!), will generate the events. Despite its uglyness, it fulfills the previous specification and, therefore, allows us to test the following examples without the need for a real asynchronous library.

Let us now write a trivial program with that library, which reads all lines from its input stream into a table and writes them to the output stream in reverse order. With traditional I/O, the program would be like this:

```
local t = {}
local inp = io.input()                -- input stream
```

```
local out = io.output()                -- output stream

for line in inp:lines() do
  t[#t + 1] = line
end

for i = #t, 1, -1 do
  out:write(t[i], "\n")
end
```

Now we rewrite that program in an event-driven style on top of the asynchronous I/O library; the result is in Figure 24.4, “Reversing a file in event-driven fashion”.

Figure 24.4. Reversing a file in event-driven fashion

```
local lib = require "async-lib"

local t = {}
local inp = io.input()
local out = io.output()
local i

-- write-line handler
local function putline ()
  i = i - 1
  if i == 0 then       -- no more lines?
    lib.stop()         -- finish the main loop
  else                 -- write line and prepare next one
    lib.writeline(out, t[i] .. "\n", putline)
  end
end

-- read-line handler
local function getline (line)
  if line then                      -- not EOF?
    t[#t + 1] = line                -- save line
    lib.readline(inp, getline)      -- read next one
  else                              -- end of file
    i = #t + 1                      -- prepare write loop
    putline()                       -- enter write loop
  end
end

lib.readline(inp, getline)          -- ask to read first line
lib.runloop()                       -- run the main loop
```

As is typical in an event-driven scenario, all our loops are gone, because the main loop is in the library. They got replaced by recursive calls disguised as events. We could improve things by using closures in a continuation-passing style, but we still could not write our own loops; we would have to rewrite them through recursion.

Coroutines allow us to reconcile our loops with the event loop. The key idea is to run our main code as a coroutine that, at each request to the library, sets the callback as a function to resume itself and then yields. Figure 24.5, “Running synchronous code on top of the asynchronous library” uses this idea to implement a library that runs conventional, synchronous code on top of the asynchronous I/O library.

Figure 24.5. Running synchronous code on top of the asynchronous library

```
local lib = require "async-lib"

function run (code)
  local co = coroutine.wrap(function ()
    code()
    lib.stop()     -- finish event loop when done
  end)
  co()             -- start coroutine
  lib.runloop()    -- start event loop
end

function putline (stream, line)
  local co = coroutine.running()     -- calling coroutine
  local callback = (function () coroutine.resume(co) end)
  lib.writeline(stream, line, callback)
  coroutine.yield()
end

function getline (stream, line)
  local co = coroutine.running()     -- calling coroutine
  local callback = (function (l) coroutine.resume(co, l) end)
  lib.readline(stream, callback)
  local line = coroutine.yield()
  return line
end
```

As its name implies, the `run` function runs the synchronous code, which it takes as a parameter. It first creates a coroutine to run the given code and finish the event loop when it is done. Then, it resumes this coroutine (which will yield at its first I/O call) and then enters the event loop.

The functions `getline` and `putline` simulate synchronous I/O. As outlined, both call an appropriate asynchronous function passing as the callback a function that resumes the calling coroutine. (Note the use of the `coroutine.running` function to access the calling coroutine.) After that, they yield, and the control goes back to the event loop. Once the operation completes, the event loop calls the callback, resuming the coroutine that triggered the operation.

With that library in place, we are ready to run synchronous code on the top of the asynchronous library. As an example, the following fragment implements once more our reverse-lines example:

```
run(function ()
  local t = {}
  local inp = io.input()
  local out = io.output()

  while true do
    local line = getline(inp)
    if not line then break end
    t[#t + 1] = line
  end

  for i = #t, 1, -1 do
    putline(out, t[i] .. "\n")
  end
```

```
end)
```

The code is equal to the original synchronous one, except that it uses `get`/`putline` for I/O and runs inside a call to `run`. Underneath its synchronous structure, it actually runs in an event-driven fashion, and it is fully compatible with other parts of the program written in a more typical event-driven style.

Exercises

Exercise 24.1: Rewrite the producer–consumer example in the section called “Who Is the Boss?” using a *producer-driven* design, where the consumer is the coroutine and the producer is the main thread.

Exercise 24.2: Exercise 6.5 asked you to write a function that prints all combinations of the elements in a given array. Use coroutines to transform this function into a generator for combinations, to be used like here:

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

Exercise 24.3: In Figure 24.5, “Running synchronous code on top of the asynchronous library”, both the functions `getline` and `putline` create a new closure each time they are called. Use memorization to avoid this waste.

Exercise 24.4: Write a line iterator for the coroutine-based library (Figure 24.5, “Running synchronous code on top of the asynchronous library”), so that you can read the file with a **for** loop.

Exercise 24.5: Can you use the coroutine-based library (Figure 24.5, “Running synchronous code on top of the asynchronous library”) to run multiple threads concurrently? What would you have to change?

Exercise 24.6: Implement a `transfer` function in Lua. If you think about resume–yield as similar to call–return, a transfer would be like a goto: it suspends the running coroutine and resumes any other coroutine, given as an argument. (Hint: use a kind of dispatch to control your coroutines. Then, a transfer would yield to the dispatch signaling the next coroutine to run, and the dispatch would resume that next coroutine.)

Chapter 25. Reflection

Reflection is the ability of a program to inspect and modify some aspects of its own execution. Dynamic languages like Lua naturally support several reflective features: environments allow run-time inspection of global variables; functions like `type` and `pairs` allow run-time inspection and traversal of unknown data structures; functions like `load` and `require` allow a program to add code to itself or update its own code. However, many things are still missing: programs cannot introspect on their local variables, programs cannot trace their execution, functions cannot know their callers, etc. The debug library fills many of these gaps.

The debug library comprises two kinds of functions: *introspective functions* and *hooks*. Introspective functions allow us to inspect several aspects of the running program, such as its stack of active functions, current line of execution, and values and names of local variables. Hooks allow us to trace the execution of a program.

Despite its name, the debug library does not give us a debugger for Lua. Nevertheless, it provides all the primitives that we need to write our own debuggers, with varying levels of sophistication.

Unlike the other libraries, we should use the debug library with parsimony. First, some of its functionality is not exactly famous for performance. Second, it breaks some sacred truths of the language, such as that we cannot access a local variable from outside its lexical scope. Although the library is readily available as a standard library, I prefer to require it explicitly in any chunk that uses it.

Introspective Facilities

The main introspective function in the debug library is `getinfo`. Its first parameter can be a function or a stack level. When we call `debug.getinfo(foo)` for a function `foo`, it returns a table with some data about this function. The table can have the following fields:

`source`: This field tells where the function was defined. If the function was defined in a string (through a call to `load`), `source` is that string. If the function was defined in a file, `source` is the file name prefixed with an at-sign.

`short_src`: This field gives a short version of `source` (up to 60 characters). It is useful for error messages.

`linedefined`: This field gives the number of the first line in the source where the function was defined.

`lastlinedefined`: This field gives the number of the last line in the source where the function was defined.

`what`: This field tells what this function is. Options are `"Lua"` if `foo` is a regular Lua function, `"C"` if it is a C function, or `"main"` if it is the main part of a Lua chunk.

`name`: This field gives a reasonable name for the function, such as the name of a global variable that stores this function.

`namewhat`: This field tells what the previous field means. This field can be `"global"`, `"local"`, `"method"`, `"field"`, or `""` (the empty string). The empty string means that Lua did not find a name for the function.

`nups`: This is the number of upvalues of that function.

`nparams`: This is the number of parameters of that function.

`isvararg`: This tells whether the function is variadic (a Boolean).

`activelines`: This field is a table representing the set of active lines of the function. An *active line* is a line with some code, as opposed to empty lines or lines containing only comments. (A typical use of this information is for setting breakpoints. Most debuggers do not allow us to set a breakpoint outside an active line, as it would be unreachable.)

`func`: This field has the function itself.

When `foo` is a C function, Lua does not have much data about it. For such functions, only the fields `what`, `name`, `namewhat`, `nups`, and `func` are meaningful.

When we call `debug.getinfo(n)` for some number *n*, we get data about the function active at that stack level. A *stack level* is a number that refers to a particular function that is active at that moment. The function calling `getinfo` has level one, the function that called it has level two, and so on. (At level zero, we get data about `getinfo` itself, a C function.) If *n* is larger than the number of active functions on the stack, `debug.getinfo` returns nil. When we query an active function, by calling `debug.getinfo` with a stack level, the resulting table has two extra fields: `currentline`, the line where the function is at that moment; and `istailcall` (a Boolean), true if this function was called by a tail call. (In this case, the real caller of this function is not on the stack anymore.)

The field `name` is tricky. Remember that, because functions are first-class values in Lua, a function may not have a name, or may have several names. Lua tries to find a name for a function by looking into the code that called the function, to see how it was called. This method works only when we call `getinfo` with a number, that is, when we ask information about a particular invocation.

The function `getinfo` is not efficient. Lua keeps debug information in a form that does not impair program execution; efficient retrieval is a secondary goal here. To achieve better performance, `getinfo` has an optional second parameter that selects what information to get. In this way, the function does not waste time collecting data that the user does not need. The format of this parameter is a string, where each letter selects a group of fields, according to the following table:

`n`	selects `name` and `namewhat`
`f`	selects `func`
`S`	selects `source`, `short_src`, `what`, `linedefined`, and `lastlinedefined`
`l`	selects `currentline`
`L`	selects `activelines`
`u`	selects `nup`, `nparams`, and `isvararg`

The following function illustrates the use of `debug.getinfo` by printing a primitive traceback of the active stack:

```
function traceback ()
  for level = 1, math.huge do
    local info = debug.getinfo(level, "Sl")
    if not info then break end
    if info.what == "C" then   -- is a C function?
      print(string.format("%d\tC function", level))
    else   -- a Lua function
      print(string.format("%d\t[%s]:%d", level,
```

```
            info.short_src, info.currentline))
      end
    end
end
```

It is not difficult to improve this function, by including more data from `getinfo`. Actually, the debug library offers such an improved version, the function `traceback`. Unlike our version, `debug.traceback` does not print its result; instead, it returns a (potentially long) string containing the traceback:

```
> print(debug.traceback())
stack traceback:
        stdin:1: in main chunk
        [C]: in ?
```

Accessing local variables

We can inspect the local variables of any active function with `debug.getlocal`. This function has two parameters: the stack level of the function we are querying and a variable index. It returns two values: the name and the current value of the variable. If the variable index is larger than the number of active variables, `getlocal` returns nil. If the stack level is invalid, it raises an error. (We can use `debug.getinfo` to check the validity of the stack level.)

Lua numbers local variables in the order that they appear in a function, counting only the variables that are active in the current scope of the function. For instance, consider the following function:

```
function foo (a, b)
  local x
  do local c = a - b end
  local a = 1
  while true do
    local name, value = debug.getlocal(1, a)
    if not name then break end
    print(name, value)
    a = a + 1
  end
end
```

The call `foo(10, 20)` will print this:

```
a       10
b       20
x       nil
a       4
```

The variable with index 1 is `a` (the first parameter), 2 is `b`, 3 is `x`, and 4 is the inner `a`. At the point where `getlocal` is called, `c` is already out of scope, while `name` and `value` are not yet in scope. (Remember that local variables are only visible after their initialization code.)

Starting with Lua 5.2, negative indices get information about the extra arguments of a variadic function: index -1 refers to the first extra argument. The name of the variable in this case is always `"(*vararg)"`.

We can also change the values of local variables, with `debug.setlocal`. Its first two parameters are a stack level and a variable index, like in `getlocal`. Its third parameter is the new value for the variable. It returns the variable name or nil if the variable index is out of scope.

Accessing non-local variables

The debug library also allows us to access the non-local variables used by a Lua function, with `getupvalue`. Unlike local variables, the non-local variables referred by a function exist even when the function is not active (this is what closures are about, after all). Therefore, the first argument for `getupvalue` is not a stack level, but a function (a closure, more precisely). The second argument is the variable index. Lua numbers non-local variables in the order in which they are first referred in a function, but this order is not relevant, because a function cannot access two non-local variables with the same name.

We can also update non-local variables, with `debug.setupvalue`. As you might expect, it has three parameters: a closure, a variable index, and the new value. Like `setlocal`, it returns the name of the variable, or nil if the variable index is out of range.

Figure 25.1, “Getting the value of a variable” shows how we can access the value of a variable from a calling function, given the variable's name.

Figure 25.1. Getting the value of a variable

```
function getvarvalue (name, level, isenv)
  local value
  local found = false

  level = (level or 1) + 1

  -- try local variables
  for i = 1, math.huge do
    local n, v = debug.getlocal(level, i)
    if not n then break end
    if n == name then
      value = v
      found = true
    end
  end
  if found then return "local", value end

  -- try non-local variables
  local func = debug.getinfo(level, "f").func
  for i = 1, math.huge do
    local n, v = debug.getupvalue(func, i)
    if not n then break end
    if n == name then return "upvalue", v end
  end

  if isenv then return "noenv" end   -- avoid loop

  -- not found; get value from the environment
  local _, env = getvarvalue("_ENV", level, true)
  if env then
    return "global", env[name]
  else        -- no _ENV available
    return "noenv"
  end
end
```

It can be used like here:

```
> local a = 4; print(getvarvalue("a"))   --> local    4
> a = "xx"; print(getvarvalue("a"))      --> global   xx
```

The parameter `level` tells where on the stack the function should look; one (the default) means the immediate caller. The plus one in the code corrects the level to include the call to `getvarvalue` itself. I will explain the parameter `isenv` in a moment.

The function first looks for a local variable. If there is more than one local with the given name, it must get the one with the highest index; thus, it must always go through the whole loop. If it cannot find any local variable with that name, then it tries the non-local variables. For that, it gets the calling closure, with `debug.getinfo`, and then it traverses its non-local variables. Finally, if it cannot find a non-local variable with that name, then it goes for a global variable: it calls itself recursively to access the proper `_ENV` variable and then looks up the name in that environment.

The parameter `isenv` avoids a tricky problem. It tells when we are in a recursive call, looking for the variable `_ENV` to query a global name. A function that uses no global variables may not have an upvalue `_ENV`. In that case, if we tried to consult `_ENV` as a global, we would enter a recursive loop, because we would need `_ENV` to get its own value. So, when `isenv` is true and the function cannot find a local or an upvalue, it does not try the global variables.

Accessing other coroutines

All introspective functions from the debug library accept an optional coroutine as their first argument, so that we can inspect the coroutine from the outside. For instance, consider the next example:

```
co = coroutine.create(function ()
  local x = 10
  coroutine.yield()
  error("some error")
end)

coroutine.resume(co)
print(debug.traceback(co))
```

The call to `traceback` will work on the coroutine `co`, resulting in something like this:

```
stack traceback:
        [C]: in function 'yield'
        temp:3: in function <temp:1>
```

The trace does not go through the call to `resume`, because the coroutine and the main program run in different stacks.

When a coroutine raises an error, it does not unwind its stack. This means that we can inspect it after the error. Continuing our example, the coroutine hits the error if we resume it again:

```
print(coroutine.resume(co))      --> false   temp:4: some error
```

Now, if we print its traceback, we get something like this:

```
stack traceback:
        [C]: in function 'error'
```

```
temp:4: in function <temp:1>
```

We can also inspect local variables from a coroutine, even after an error:

```
print(debug.getlocal(co, 1, 1))     --> x       10
```

Hooks

The hook mechanism of the debug library allows us to register a function to be called at specific events as a program runs. There are four kinds of events that can trigger a hook:

- *call* events happen every time Lua calls a function;
- *return* events happen every time a function returns;
- *line* events happen when Lua starts executing a new line of code;
- *count* events happen after a given number of instructions. (Instructions here mean internal opcodes, which we visited briefly in the section called “Precompiled Code”.)

Lua calls all hooks with a string argument that describes the event that generated the call: `"call"` (or `"tail call"`), `"return"`, `"line"`, or `"count"`. For line events, it also passes a second argument, the new line number. To get more information inside a hook, we have to call `debug.getinfo`.

To register a hook, we call `debug.sethook` with two or three arguments: the first argument is the hook function; the second argument is a mask string, which describes the events we want to monitor; and the optional third argument is a number that describes at what frequency we want to get count events. To monitor the call, return, and line events, we add their first letters (`c`, `r`, or `l`) into the mask string. To monitor the count event, we simply supply a counter as the third argument. To turn off hooks, we call `sethook` with no arguments.

As a simple example, the following code installs a primitive tracer, which prints each line the interpreter executes:

```
debug.sethook(print, "l")
```

This call simply installs `print` as the hook function and instructs Lua to call it only at line events. A more elaborated tracer can use `getinfo` to add the current file name to the trace:

```
function trace (event, line)
  local s = debug.getinfo(2).short_src
  print(s .. ":" .. line)
end

debug.sethook(trace, "l")
```

A useful function to use with hooks is `debug.debug`. This simple function gives us a prompt that executes arbitrary Lua commands. It is roughly equivalent to 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 the “command” `cont`, the function returns. The standard implementation is very simple and runs the commands in the global environment, outside the scope of the code being debugged. Exercise 25.4 discusses a better implementation.

Profiles

Besides debugging, another common application for reflection is profiling, that is, an analysis of the behavior of a program regarding its use of resources. For a timing profile, it is better to use the C interface: the overhead of a Lua call for each hook is too high and may invalidate any measurement. However, for counting profiles, Lua code does a decent job. In this section, we will develop a rudimentary profiler that lists the number of times each function in a program is called during a run.

The main data structures of our program are two tables: one maps functions to their call counters, and the other maps functions to their names. The indices to both tables are the functions themselves.

```
local Counters = {}
local Names = {}
```

We could retrieve the function names after the profiling, but remember that we get better results if we get the name of a function while it is active, because then Lua can look at the code that is calling the function to find its name.

Now we define the hook function. Its job is to get the function being called, increment the corresponding counter, and collect the function name. The code is in Figure 25.2, “Hook for counting number of calls”.

Figure 25.2. Hook for counting number of calls

```
local function hook ()
  local f = debug.getinfo(2, "f").func
  local count = Counters[f]
  if count == nil then    -- first time 'f' is called?
    Counters[f] = 1
    Names[f] = debug.getinfo(2, "Sn")
  else     -- only increment the counter
    Counters[f] = count + 1
  end
end
```

The next step is to run the program with that hook. We will assume that the program we want to analyze is in a file and that the user gives this file name as an argument to the profiler, like this:

```
% lua profiler main-prog
```

With this scheme, the profiler can get the file name in `arg[1]`, turn on the hook, and run the file:

```
local f = assert(loadfile(arg[1]))
debug.sethook(hook, "c")  -- turn on the hook for calls
f()                       -- run the main program
debug.sethook()           -- turn off the hook
```

The last step is to show the results. The function `getname`, in Figure 25.3, “Getting the name of a function”, produces a name for a function.

Figure 25.3. Getting the name of a function

```
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
```

Because function names in Lua are so uncertain, we add to each function its location, given as a pair *file:line*. If a function has no name, then we use just its location. For a C function, we use only its name (as it has no location). After that definition, we print each function with its counter:

```
for func, count in pairs(Counters) do
  print(getname(func), count)
end
```

If we apply our profiler to the Markov example that we developed in Chapter 19, *Interlude: Markov Chain Algorithm*, we get a result like this:

```
[markov.lua]:4 884723
write   10000
[markov.lua]:0 1
read    31103
sub     884722
...
```

This result means that the anonymous function at line 4 (which is the iterator function defined inside `allwords`) was called 884723 times, `write` (`io.write`) was called 10000 times, and so on.

There are several improvements that we can make to this profiler, such as to sort the output, to print better function names, and to embellish the output format. Nevertheless, this basic profiler is already useful as it is.

Sandboxing

In the section called “`_ENV` and `load`”, we saw how easy it is to use `load` to run a Lua chunk in a restricted environment. Because Lua does all communication with the external world through library functions, once we remove these functions, we also remove the possibility of a script to have any effect on the external world. Nevertheless, we are still susceptible to denial of service (DoS) attacks, with a script wasting large amounts of CPU time or memory. Reflection, in the form of debug hooks, provides an interesting approach to curb such attacks.

A first step is to use a count hook to limit the number of instructions that a chunk can execute. Figure 25.4, “A naive sandbox with hooks” shows a program to run a given file in that kind of sandbox.

Figure 25.4. A naive sandbox with hooks

```
local debug = require "debug"

-- maximum "steps" that can be performed
local steplimit = 1000

local count = 0     -- counter for steps

local function step ()
  count = count + 1
  if count > steplimit then
    error("script uses too much CPU")
  end
end

-- load file
local f = assert(loadfile(arg[1], "t", {}))

debug.sethook(step, "", 100)    -- set hook

f()    -- run file
```

The program loads the given file, sets the hook, and runs the file. It sets the hook as a count hook, so that Lua calls the hook every 100 instructions. The hook (the function `step`) only increments a counter and checks it against a fixed limit. What can possibly go wrong?

Of course, we must restrict the size of the chunks that we load: a huge chunk can exhaust memory only by being loaded. Another problem is that a program can consume huge amounts of memory with surprisingly few instructions, as the next fragment shows:

```
local s = "123456789012345"
for i = 1, 36 do s = s .. s end
```

With less than 150 instructions, this tiny fragment will try to create a string with one terabyte. Clearly, restricting only steps and program size is not enough.

One improvement is to check and limit memory use in the `step` function, as we show in Figure 25.5, “Controlling memory use”.

Figure 25.5. Controlling memory use

```
-- maximum memory (in KB) that can be used
local memlimit = 1000

-- maximum "steps" that can be performed
local steplimit = 1000

local function checkmem ()
  if collectgarbage("count") > memlimit then
    error("script uses too much memory")
  end
end

local count = 0
local function step ()
  checkmem()
  count = count + 1
  if count > steplimit then
    error("script uses too much CPU")
  end
end

as before
```

Because memory can grow so fast with so few instructions, we should set a very low limit or call the hook in small steps. More concretely, a program can do a thousandfold increase in the size of a string in 40 instructions. So, either we call the hook with a higher frequency than every 40 steps or we set the memory limit to one thousandth of what we can really afford. I would probably choose both.

A subtler problem is the string library. We can call any function from this library as a method on a string. Therefore, we can call these functions even if they are not in the environment; literal strings smuggle them into our sandbox. No function in the string library affects the external world, but they bypass our step counter. (A call to a C function counts as one instruction in Lua.) Some functions in the string library can be quite dangerous for DoS attacks. For instance, the call `("x"):rep(2^30)` swallows 1 GB of memory in a single step. As another example, Lua 5.2 takes 13 minutes to run the following call in my new machine:

```
s = "01234567890123456789012345678901234567890123456789"
s:find(".*.*.*.*.*.*.*.*.*x")
```

An interesting way to restrict the access to the string library is to use call hooks. Every time a function is called, we check whether it is authorized. Figure 25.6, “Using hooks to bar calls to unauthorized functions” implements this idea.

Figure 25.6. Using hooks to bar calls to unauthorized functions

```
local debug = require "debug"

-- maximum "steps" that can be performed
local steplimit = 1000

local count = 0     -- counter for steps

-- set of authorized functions
local validfunc = {
  [string.upper] = true,
  [string.lower] = true,
  ...       -- other authorized functions
}

local function hook (event)
  if event == "call" then
    local info = debug.getinfo(2, "fn")
    if not validfunc[info.func] then
      error("calling bad function: " .. (info.name or "?"))
    end
  end
  count = count + 1
  if count > steplimit then
    error("script uses too much CPU")
  end
end

-- load chunk
local f = assert(loadfile(arg[1], "t", {}))

debug.sethook(hook, "", 100)    -- set hook

f()    -- run chunk
```

In that code, the table `validfunc` represents a set with the functions that the program can call. The function `hook` uses the debug library to access the function being called and then checks whether that function is in the `validfunc` set.

An important point in any sandbox implementation is what functions we allow inside the sandbox. Sandboxes for data description can restrict all or most functions. Other sandboxes must be more forgiving, maybe offering their own restricted implementations for some functions (e.g., `load` restricted to small text chunks, file access restricted to a fixed directory, or pattern matching restricted to small subjects).

We should never think in terms of what functions to remove, but what functions to add. For each candidate, we must carefully consider its possible weaknesses, which may be subtle. As a rule of thumb, all functions from the mathematical library are safe. Most functions from the string library are safe; just be careful with resource-consuming ones. The debug and package libraries are off-limits; almost everything there can be dangerous. The functions `setmetatable` and `getmetatable` are also tricky: first, they can allow access to otherwise inaccessible values; moreover, they allow the creation of tables with finalizers, where someone can install all sorts of “time bombs” (code that can be executed outside the sandbox, when the table is collected).

Exercises

Exercise 25.1: Adapt `getvarvalue` (Figure 25.1, “Getting the value of a variable”) to work with different coroutines (like the functions from the `debug` library).

Exercise 25.2: Write a function `setvarvalue` similar to `getvarvalue` (Figure 25.1, “Getting the value of a variable”).

Exercise 25.3: Write a version of `getvarvalue` (Figure 25.1, “Getting the value of a variable”) that returns a table with all variables that are visible at the calling function. (The returned table should not include environmental variables; instead, it should inherit them from the original environment.)

Exercise 25.4: Write an improved version of `debug.debug` that runs the given commands as if they were in the lexical scope of the calling function. (Hint: run the commands in an empty environment and use the `__index` metamethod attached to the function `getvarvalue` to do all accesses to variables.)

Exercise 25.5: Improve the previous exercise to handle updates, too.

Exercise 25.6: Implement some of the suggested improvements for the basic profiler that we developed in the section called “Profiles”.

Exercise 25.7: Write a library for breakpoints. It should offer at least two functions:

```
setbreakpoint(function, line)    --> returns handle
removebreakpoint(handle)
```

We specify a breakpoint by a function and a line inside that function. When the program hits a breakpoint, the library should call `debug.debug`. (Hint: for a basic implementation, use a line hook that checks whether it is in a breakpoint; to improve performance, use a call hook to trace program execution and only turn on the line hook when the program is running the target function.)

Exercise 25.8: One problem with the sandbox in Figure 25.6, “Using hooks to bar calls to unauthorized functions” is that sandboxed code cannot call its own functions. How can you correct this problem?

Chapter 26. Interlude: Multithreading with Coroutines

In this interlude, we will see an implementation of a multithreading system on top of coroutines.

As we saw earlier, coroutines allow a kind of collaborative multithreading. Each coroutine is equivalent to a thread. A pair yield–resume switches control from one thread to another. However, unlike regular multithreading, coroutines are non preemptive. While a coroutine is running, we cannot stop it from the outside. It suspends execution only when it explicitly requests so, through a call to `yield`. For several applications, this is not a problem, quite the opposite. Programming is much easier in the absence of preemption. We do not need to be paranoid about synchronization bugs, because all synchronization among threads is explicit in the program. We just need to ensure that a coroutine yields only when it is outside a critical region.

However, with non-preemptive multithreading, whenever any thread calls a blocking operation, the whole program blocks until the operation completes. For many applications, this behavior is unacceptable, which leads many programmers to disregard coroutines as a real alternative to conventional multithreading. As we will see here, this problem has an interesting (and obvious, with hindsight) solution.

Let us assume a typical multithreading situation: we want to download several remote files through HTTP. To download several remote files, first we must learn how to download one remote file. In this example, we will use the LuaSocket library. To download a file, we must open a connection to its site, send a request to the file, receive the file (in blocks), and close the connection. In Lua, we can write this task as follows. First, we load the LuaSocket library:

```
local socket = require "socket"
```

Then, we define the host and the file we want to download. In this example, we will download the Lua 5.3 manual from the Lua site:

```
host = "www.lua.org"
file = "/manual/5.3/manual.html"
```

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

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

This operation returns a connection object, which we use to send the file request:

```
local request = string.format(
    "GET %s HTTP/1.0\r\nhost: %s\r\n\r\n", file, host)
c:send(request)
```

Next, we read the file in blocks of 1 kB, writing each block to the standard output:

```
repeat
  local s, status, partial = c:receive(2^10)
  io.write(s or partial)
until status == "closed"
```

The method `receive` returns either a string with what it read or nil in case of error; in the latter case, it also returns an error code (`status`) and what it read until the error (`partial`). When the host closes the connection, we print that remaining input and break the receive loop.

After downloading the file, we close the connection:

```
c:close()
```

Now that we know how to download one file, let us return to the problem of downloading several files. The trivial approach is to download one at a time. However, this sequential approach, where we start reading a file only after finishing the previous one, is too slow. When reading a remote file, a program spends most of its time waiting for data to arrive. More specifically, it spends most of its time blocked in the call to `receive`. So, the program could run much faster if it downloaded all files concurrently. Then, while a connection has no data available, the program can read from another connection. Clearly, coroutines offer a convenient way to structure these simultaneous downloads. We create a new thread for each download task. When a thread has no data available, it yields control to a simple dispatcher, which invokes another thread.

To rewrite the program with coroutines, we first rewrite the previous download code as a function. The result is in Figure 26.1, “Function to download a Web page”.

Figure 26.1. Function to download a Web page

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

Because we are not interested in the remote file contents, this function counts and prints the file size, instead of writing the file to the standard output. (With several threads reading several files, the output would shuffle all files.)

In this new code, we use an auxiliary function (`receive`) to receive data from the connection. In the sequential approach, its code would be like this:

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

For the concurrent implementation, this function must receive data without blocking. Instead, if there is not enough data available, it yields. The new code is 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 call to `settimeout(0)` makes any operation over the connection a non-blocking operation. When the resulting status is `"timeout"`, it means that the operation returned without completion. In this case, the thread yields. The non-false argument passed to `yield` signals to the dispatcher that the thread is still performing its task. Note that, even in case of a timeout, the connection returns what it read until the timeout, which is in the variable `partial`.

Figure 26.2, “The dispatcher” shows the dispatcher plus some auxiliary code.

Figure 26.2. The dispatcher

```
tasks = {}    -- list of all live tasks

function get (host, file)
  -- create coroutine for a task
  local co = coroutine.wrap(function ()
    download(host, file)
  end)
  -- insert it in the list
  table.insert(tasks, co)
end

function dispatch ()
  local i = 1
  while true do
    if tasks[i] == nil then   -- no other tasks?
      if tasks[1] == nil then   -- list is empty?
        break   -- break the loop
      end
      i = 1                     -- else restart the loop
    end
    local res = tasks[i]()   -- run a task
    if not res then    -- task finished?
      table.remove(tasks, i)
    else
      i = i + 1   -- go to next task
    end
  end
end
```

The table `tasks` keeps a list of all live tasks for the dispatcher. The function `get` ensures that each download task runs in an individual thread. The dispatcher itself is mainly a loop that goes through all tasks, resuming them one by one. It must also remove from the list the tasks that have finished. It stops the loop when there are no more tasks to run.

Finally, the main program creates the tasks it needs and calls the dispatcher. To download some distributions from the Lua site, the main program could be like this:

```
get("www.lua.org", "/ftp/lua-5.3.2.tar.gz")
get("www.lua.org", "/ftp/lua-5.3.1.tar.gz")
get("www.lua.org", "/ftp/lua-5.3.0.tar.gz")
get("www.lua.org", "/ftp/lua-5.2.4.tar.gz")
get("www.lua.org", "/ftp/lua-5.2.3.tar.gz")
```

```
dispatch()   -- main loop
```

The sequential implementation takes fifteen seconds to download these files, in my machine. This implementation with coroutines runs more than three times faster.

Despite the speedup, this last implementation is far from optimal. Everything goes fine while at least one thread has something to read. However, when no thread has data to read, the dispatcher does a busy wait, going from thread to thread only to check that they still have no data. As a result, this coroutine implementation uses three orders of magnitude more CPU than the sequential solution.

To avoid this behavior, we can use the function `select` from LuaSocket: it allows a program to block while waiting for a status change in a group of sockets. The changes in our implementation are small: we have to change only the dispatcher, as shown in Figure 26.3, “Dispatcher using `select`”.

Figure 26.3. Dispatcher using `select`

```
function dispatch ()
  local i = 1
  local timedout = {}
  while true do
    if tasks[i] == nil then   -- no other tasks?
      if tasks[1] == nil then   -- list is empty?
        break     -- break the loop
      end
      i = 1                     -- else restart the loop
      timedout = {}
    end
    local res = tasks[i]()   -- run a task
    if not res then    -- task finished?
      table.remove(tasks, i)
    else               -- time out
      i = i + 1
      timedout[#timedout + 1] = res
      if #timedout == #tasks then    -- all tasks blocked?
        socket.select(timedout)   -- wait
      end
    end
  end
end
```

Along the loop, this new dispatcher collects the timed-out connections in the table `timedout`. (Remember that `receive` passes such connections to `yield`, thus `resume` returns them.) If all connections time out, the dispatcher calls `select` to wait for any of these connections to change status. This final implementation runs as fast as the previous implementation, with coroutines. Moreover, as it does not do busy waits, it uses just as much CPU as the sequential implementation.

Exercises

Exercise 26.1: Implement and run the code presented in this chapter.

Part IV. The C API

Table of Contents

Chapter 27. An Overview of the C API

Lua is an *embedded language*. This means that Lua is not a stand-alone application, but a library that we can link with other applications to incorporate Lua facilities into them.

You may be wondering: if Lua is not a stand-alone program, how come we have been using Lua stand-alone through the whole book until now? The solution to this puzzle is the Lua interpreter —the executable `lua`. This executable is a small application, around six hundred lines of code, that uses the Lua library to implement the stand-alone interpreter. The program handles the interface with the user, taking her files and strings to feed them to the Lua library, which does the bulk of the work (such as actually running Lua code).

This ability to be used as a library to extend an application is what makes Lua an *embeddable language*. At the same time, a program that uses Lua can register new functions in the Lua environment; such functions are implemented in C (or another language), so that they can add facilities that cannot be written directly in Lua. This is what makes Lua an *extensible language*.

These two views of Lua (as an embeddable language and as an extensible language) correspond to two kinds of interaction between C and Lua. In the first kind, C has the control and Lua is the library. The C code in this kind of interaction is what we call *application code*. In the second kind, Lua has the control and C is the library. Here, the C code is called *library code*. Both application code and library code use the same API to communicate with Lua, the so-called C API.

The C API is the set of functions, constants, and types that allow C code to interact with Lua.[1] It comprises functions to read and write Lua global variables, to call Lua functions, to run pieces of Lua code, to register C functions so that they can later be called by Lua code, and so on. Virtually anything that Lua code can do can also be done by C code through the C API.

The C API follows the *modus operandi* of C, which is quite different from that of Lua. When programming in C, we must care about type checking, error recovery, memory-allocation errors, and several other sources of complexity. Most functions in the API do not check the correctness of their arguments; it is our responsibility to make sure that the arguments are valid before calling a function.[2] If we make mistakes, we can get a crash instead of a well-behaved error message. Moreover, the API emphasizes flexibility and simplicity, sometimes at the cost of ease of use. Common tasks may involve several API calls. This may be boring, but it gives us full control over all details.

As its title says, the goal of this chapter is to give an overview of what is involved when we use Lua from C. Do not try to understand all the details of what is going on now; we will fill them in later. Nevertheless, do not forget that you always can find more details about specific functions in the Lua reference manual. Moreover, you can find several examples of API uses in the Lua distribution itself. The Lua stand-alone interpreter (`lua.c`) provides examples of application code, while the standard libraries (`lmathlib.c`, `lstrlib.c`, etc.) provide examples of library code.

From now on, we are wearing a C programmer's hat.

A First Example

We will start this overview with a simple example of an application program: a stand-alone Lua interpreter. We can write a bare-bones stand-alone interpreter as in Figure 27.1, “A bare-bones stand-alone Lua interpreter”.

[1] Throughout this text, the term “function” actually means “function or macro”. The API implements several facilities as macros.

[2] You can compile Lua with the macro `LUA_USE_APICHECK` defined to enable some checks; this option is particularly useful when debugging your C code. Nevertheless, several errors simply cannot be detected in C, such as invalid pointers.

Figure 27.1. A bare-bones stand-alone 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 the 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 error message from the stack */
    }
  }

  lua_close(L);
  return 0;
}
```

The header file `lua.h` declares the basic functions provided by Lua. It includes functions to create a new Lua environment, to invoke Lua functions, to read and write global variables in the environment, to register new functions to be called by Lua, and so on. Everything declared in `lua.h` has a `lua_` prefix (e.g., `lua_pcall`).

The header file `lauxlib.h` declares the functions provided by the *auxiliary library* (auxlib). All its declarations start with `luaL_` (e.g., `luaL_loadstring`). The auxiliary library uses the basic API provided by `lua.h` to provide a higher abstraction level, in particular with abstractions used by the standard libraries. The basic API strives for economy and orthogonality, whereas the auxiliary library strives for practicality for a few common tasks. Of course, it is very easy for your program to create other abstractions that it needs, too. Keep in mind that the auxiliary library has no access to the internals of Lua. It does its entire job through the official basic API declared in `lua.h`. Whatever it does, your program can do too.

The Lua library defines no C global variables at all. It keeps all its state in the dynamic structure `lua_State`; all functions inside Lua receive a pointer to this structure as an argument. This design makes Lua reentrant and ready to be used in multithreaded code.

As its name implies, the function `luaL_newstate` creates a new Lua state. When `luaL_newstate` creates a fresh state, its environment contains no predefined functions, not even `print`. To keep Lua small, all standard libraries come as separate packages, so that we do not have to use them if we do not need to. The header file `lualib.h` declares functions to open the libraries. The function `luaL_openlibs` opens all standard libraries.

After creating a state and populating it with the standard libraries, it is time to handle user input. For each line the user enters, the program first compiles it with `luaL_loadstring`. If there are no errors, the call returns zero and pushes the resulting function on the stack. (We will discuss this mysterious stack in the next section.) Then the program calls `lua_pcall`, which pops the function from the stack and runs it in protected mode. Like `luaL_loadstring`, `lua_pcall` returns zero if there are no errors. In case of

error, both functions push an error message on the stack; we then get this message with `lua_tostring` and, after printing it, remove it from the stack with `lua_pop`.

Real error handling can be quite complex in C, and how to do it depends on the nature of our application. The Lua core never writes anything directly to any output stream; it signals errors by returning error messages. Each application can handle these messages in a way most appropriate to its needs. To simplify our discussions, we will assume for our next examples a simple error handler like the following one, which prints an error message, closes the Lua state, and finishes the whole 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);
}
```

Later we will discuss more about error handling in the application code.

Because we can compile Lua as either C or C++ code, `lua.h` does not include the following boilerplate commonly used in C libraries:

```
#ifdef __cplusplus
extern "C" {
#endif
   ...
#ifdef __cplusplus
}
#endif
```

If we have compiled Lua as C code and are using it in C++, we can include `lua.hpp` instead of `lua.h`. It is defined as follows:

```
extern "C" {
#include "lua.h"
}
```

The Stack

A major component in the communication between Lua and C is an omnipresent virtual *stack*. Almost all API calls operate on values on this stack. All data exchange from Lua to C and from C to Lua occurs through this stack. Moreover, we can use the stack to keep intermediate results, too.

We face two problems when trying to exchange values between Lua and C: the mismatch between a dynamic and a static type system and the mismatch between automatic and manual memory management.

In Lua, when we write `t[k] = v`, both `k` and `v` can have several different types; even `t` can have different types, due to metatables. If we want to offer this operation in C, however, any given `settable` function

must have a fixed type. We would need dozens of different functions for this single operation (one function for each combination of types for the three arguments).

We could solve this problem by declaring some kind of union type in C —let us call it `lua_Value`— that could represent all Lua values. Then, we could declare `settable` as

```
void lua_settable (lua_Value a, lua_Value k, lua_Value v);
```

This solution has two drawbacks. First, it can be difficult to map such a complex type to other languages; we designed Lua to interface easily not only with C/C++, but also with Java, Fortran, C#, and the like. Second, Lua does garbage collection: if we keep a Lua table in a C variable, the Lua engine has no way to know about this use; it may (wrongly) assume that this table is garbage and collect it.

Therefore, the Lua API does not define anything like a `lua_Value` type. Instead, it uses the stack to exchange values between Lua and C. Each slot in this stack can hold any Lua value. Whenever we want to ask for a value from Lua (such as the value of a global variable), we call Lua, which pushes the required value onto the stack. Whenever we want to pass a value to Lua, we first push the value onto the stack, and then we call Lua (which will pop the value). We still need a different function to push each C type onto the stack and a different function to get each C type from the stack, but we avoid combinatorial explosion. Moreover, because this stack is part of the Lua state, the garbage collector knows which values C is using.

Nearly all functions in the API use the stack. As we saw in our first example, `luaL_loadstring` leaves its result on the stack (either the compiled chunk or an error message); `lua_pcall` gets the function to be called from the stack and leaves any error message there too.

Lua manipulates this stack in a strict LIFO discipline (Last In, First Out). When we call Lua, it changes only the top part of the stack. Our C code has more freedom; specifically, it can inspect any element in the stack and even insert and delete elements at any position.

Pushing elements

The API has a push function for each Lua type with a direct representation in C: `lua_pushnil` for the constant nil, `lua_pushboolean` for Booleans (integers, in C), `lua_pushnumber` for doubles,[3] `lua_pushinteger` for integers, `lua_pushlstring` for arbitrary strings (a pointer to `char` plus a length), and `lua_pushstring` for zero-terminated 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_pushlstring  (lua_State *L, const char *s, size_t len);
void lua_pushstring   (lua_State *L, const char *s);
```

There are also functions to push C functions and userdata values onto the stack; we will discuss them later.

The type `lua_Number` is the numeric float type in Lua. It is `double` by default, but we can configure Lua at compile time to use `float` or even `long double`. The type `lua_Integer` is the numeric integer type in Lua. Usually, it is defined as `long long`, which is a signed 64-bit integer. Again, it is trivial to configure Lua to use `int` or `long` for this type. The combination `float`–`int`, with 32-bit floats and integers, creates what we call Small Lua, which is particularly interesting for small machines and restricted hardware.[4]

[3] For historical reasons, the term “number” in the API refers to doubles.
[4] For these configurations, have a look in the file `luaconf.h`.

Strings in Lua are not zero-terminated; they can contain arbitrary binary data. In consequence, the basic function to push a string onto the stack is `lua_pushlstring`, which requires an explicit length as an argument. For zero-terminated strings, we can use also `lua_pushstring`, which uses `strlen` to supply the string length. Lua never keeps pointers to external strings (or to any other external object except C functions, which are always static). For any string that it has to keep, Lua either makes an internal copy or reuses one. Therefore, we can free or modify our buffers as soon as these functions return.

Whenever we push an element onto the stack, it is our responsibility to ensure that the stack has space for it. Remember, you are a C programmer now; Lua will not spoil you. When Lua starts and any time that Lua calls C, the stack has at least 20 free slots. (The header file `lua.h` defines this constant as `LUA_MINSTACK`.) This space is more than enough for most common uses, so usually we do not even think about it. However, some tasks need more stack space, in particular if we have a loop pushing elements onto the stack. In those cases, we need to call `lua_checkstack`, which checks whether the stack has enough space for our needs:

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

Here, `sz` is the number of extra slots we need. If possible, `lua_checkstack` grows the stack to accommodate the required extra size. Otherwise, it returns zero.

The auxiliary library offers a higher-level function to check for stack space:

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

This function is similar to `lua_checkstack` but, if it cannot fulfill the request, it raises an error with the given message, instead of returning an error code.

Querying elements

To refer to elements on the stack, the API uses *indices*. The first element pushed on the stack has index 1, the next one has index 2, and so on. We can also access elements using the top of the stack as our reference, with negative indices. In this case, -1 refers to the element on top (that is, the last element pushed), -2 to the previous element, and so on. For instance, the call `lua_tostring(L, -1)` returns the value on the top of the stack as a string. As we will see, there are several occasions when it is natural to index the stack from the bottom (that is, with positive indices), and several other occasions when the natural way is to use negative indices.

To check whether a stack element has a specific type, the API offers a family of functions called `lua_is*`, where the `*` can be any Lua type. So, there are `lua_isnil`, `lua_isnumber`, `lua_isstring`, `lua_istable`, and the like. All these functions have the same prototype:

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

Actually, `lua_isnumber` does not check whether the value has that specific type, but whether the value can be converted to that type; `lua_isstring` is similar: in particular, any number satisfies `lua_isstring`.

There is also a function `lua_type`, which returns the type of an element on the stack. Each type is represented by a respective constant: `LUA_TNIL`, `LUA_TBOOLEAN`, `LUA_TNUMBER`, `LUA_TSTRING`, etc. We use this function mainly in conjunction with a switch statement. It is also useful when we need to check for strings and numbers without potential coercions.

To get a value from the stack, there are the `lua_to*` functions:

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

```
const char  *lua_tolstring (lua_State *L, int index,
                                          size_t *len);
lua_State   *lua_tothread (lua_State *L, int index);
lua_Number   lua_tonumber  (lua_State *L, int index);
lua_Integer  lua_tointeger (lua_State *L, int index);
```

We can call any of these functions even when the given element does not have an appropriate type. The function `lua_toboolean` works for any type, converting any Lua value to a C Boolean according to the Lua rules for conditions: zero for the values nil and **false**, and one for any other Lua value. The functions `lua_tolstring` and `lua_tothread` return `NULL` for values with incorrect types. The numeric functions, however, have no way to signal a wrong type, so they simply return zero. Formerly we would need to call `lua_isnumber` to check the type, but Lua 5.2 introduced the following new functions:

```
lua_Number   lua_tonumberx (lua_State *L, int idx, int *isnum);
lua_Integer  lua_tointegerx (lua_State *L, int idx, int *isnum);
```

The out parameter `isnum` returns a Boolean that indicates whether the Lua value was successfully coerced to the desired type.

The function `lua_tolstring` returns a pointer to an internal copy of the string and stores the string's length in the position given by `len`. We must not change this internal copy (there is a `const` there to remind us). Lua ensures that this pointer is valid as long as the corresponding string value is on the stack. When a C function called by Lua returns, Lua clears its stack; therefore, as a rule, we should never store pointers to Lua strings outside the function that got them.

Any string that `lua_tolstring` returns always has an extra zero at its end, but it can have other zeros inside it. The size returned through the third argument, `len`, is the real string's length. In particular, assuming that the value on the top of the stack is a string, the following assertions are always valid:

```
size_t len;
const char *s = lua_tolstring(L, -1, &len); /* any Lua string */
assert(s[len] == '\0');
assert(strlen(s) <= len);
```

We can call `lua_tolstring` with `NULL` as its third argument if we do not need the length. Better yet, we can use the macro `lua_tostring`, which simply calls `lua_tolstring` with a `NULL` third argument.

To illustrate the use of these functions, Figure 27.2, “Dumping the stack” presents a useful helper function that dumps the entire content of the stack.

Figure 27.2. Dumping the stack

```
static void stackDump (lua_State *L) {
  int i;
  int top = lua_gettop(L);  /* depth of the stack */
  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 */
}
```

This function traverses the stack from bottom to top, printing each element according to its type. It prints strings between quotes; for numbers it uses a `"%g"` format; for values with no C equivalents (tables, functions, etc.), it prints only their types. (`lua_typename` converts a type code to a type name.)

In Lua 5.3, we can still print all numbers with `lua_tonumber` and the `"%g"` format, as integers are always coercible to floats. However, we may prefer to print integers as integers, to avoid losing precision. In that case, we can use the new function `lua_isinteger` to distinguish integers from floats:

```
      case LUA_TNUMBER: {  /* numbers */
        if (lua_isinteger(L, i))  /* integer? */
          printf("%lld", lua_tointeger(L, i));
        else  /* float */
          printf("%g", lua_tonumber(L, i));
        break;
      }
```

Other stack operations

Besides the previous functions, which exchange values between C and the stack, the API offers also the following operations for generic stack manipulation:

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

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

The function `lua_gettop` returns the number of elements on the stack, which is also the index of the top element. The function `lua_settop` sets the top (that is, the number of elements on the stack) to a specific value. If the previous top was higher than the new one, the function discards the extra top values. Otherwise, it pushes nils on the stack to get the given size. In particular, `lua_settop(L, 0)` empties the stack. We can also use negative indices with `lua_settop`. Using this facility, the API offers the following macro, which pops `n` elements from the stack:

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

The function `lua_pushvalue` pushes on the stack a copy of the element at the given index.

The function `lua_rotate` is new in Lua 5.3. As the name implies, it rotates the stack elements from the given index to the top of the stack by `n` positions. A positive `n` rotates the elements in the direction of the top; a negative `n` rotates in the other direction. This is a quite versatile function, and two other API operations are defined as macros using it. One is `lua_remove`, which removes the element at the given index, shifting down the elements above this position to fill in the gap. Its definition is as follows:

```
#define lua_remove(L,idx)  \
          (lua_rotate(L, (idx), -1), lua_pop(L, 1))
```

That is, it rotates the stack by one position, moving the desired element to the top, and then pops that element. The other macro is `lua_insert`, which moves the top element into the given position, shifting up the elements above this position to open space:

```
#define lua_insert(L,idx)       lua_rotate(L, (idx), 1)
```

The function `lua_replace` pops a value and sets it as the value of the given index, without moving anything; finally, `lua_copy` copies the value at one index to another, leaving the original untouched.[5] Note that the following operations have no effect on a non-empty stack:

```
lua_settop(L, -1);  /* set top to its current value */
lua_insert(L, -1);  /* move top element to the top */
lua_copy(L, x, x);  /* copy an element to its own position */
lua_rotate(L, x, 0);  /* rotates by zero positions */
```

The program in Figure 27.3, “Example of stack manipulation” uses `stackDump` (defined in Figure 27.2, “Dumping the stack”) to illustrate these stack operations.

[5]The function `lua_copy` was introduced in Lua 5.2.

Figure 27.3. Example of stack manipulation

```
#include <stdio.h>
#include "lua.h"
#include "lauxlib.h"

static void stackDump (lua_State *L) {
  as in Figure 27.2, “Dumping the stack”
}

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);
    /* will print:  true  10  nil  'hello'  */

  lua_pushvalue(L, -4); stackDump(L);
    /* will print:  true  10  nil  'hello'  true  */

  lua_replace(L, 3); stackDump(L);
    /* will print:  true  10  true  'hello'  */

  lua_settop(L, 6); stackDump(L);
    /* will print:  true  10  true  'hello'  nil  nil  */

  lua_rotate(L, 3, 1); stackDump(L);
    /* will print:  true  10  nil  true  'hello'  nil  */

  lua_remove(L, -3); stackDump(L);
    /* will print:  true  10  nil  'hello'  nil */

  lua_settop(L, -5); stackDump(L);
    /* will print:  true  */

  lua_close(L);
  return 0;
}
```

Error Handling with the C API

All structures in Lua are dynamic: they grow as needed, and eventually shrink again when possible. This means that the possibility of a memory-allocation failure is pervasive in Lua. Almost any operation can face this eventuality. Moreover, many operations can raise other errors; for instance, an access to a global variable can trigger an `__index` metamethod and that metamethod may raise an error. Finally, operations that allocate memory eventually trigger the garbage collector, which may invoke finalizers, which can raise errors too. In short, the vast majority of functions in the Lua API can result in errors.

Instead of using error codes for each operation in its API, Lua uses exceptions to signal errors. Unlike C++ or Java, the C language does not offer an exception handling mechanism. To circumvent this difficulty, Lua

uses the `setjmp` facility from C, which results in a mechanism somewhat similar to exception handling. Therefore, most API functions can raise an error (that is, call `longjmp`) instead of returning.

When we write library code (C functions to be called from Lua), the use of long jumps requires no extra work from our part, because Lua catches any error. When we write application code (C code that calls Lua), however, we must provide a way to catch those errors.

Error handling in application code

When our application calls functions in the Lua API, it is exposed to errors. As we just discussed, Lua usually signals these errors through long jumps. However, if there is no corresponding `setjmp`, the interpreter cannot make a long jump. In that case, any error in the API causes Lua to call a panic function and, if that function returns, exit the application. We can set our own panic function with `lua_atpanic`, but there is not much that it can do.

To properly handle errors in our application code, we must call our code through Lua, so that it sets an appropriate context to catch errors —that is, it runs the code in the context of a `setjmp`. In the same way that we can run Lua code in protected mode using `pcall`, we can run C code using `lua_pcall`. More specifically, we pack the code in a function and call that function through Lua, using `lua_pcall`. With this setting, our C code will run in protected mode. Even in case of memory-allocation failure, `lua_pcall` returns a proper error code, leaving the interpreter in a consistent state. The following fragment shows the idea:

```
static int foo (lua_State *L) {
  code to run in protected mode
  return 0;
}

int secure_foo (lua_State *L) {
  lua_pushcfunction(L, foo);  /* push 'foo' as a Lua function */
  return (lua_pcall(L, 0, 0, 0) == 0);
}
```

In this example, no matter what happens, a call to `secure_foo` will return a Boolean signaling the success of `foo`. In particular, note that the stack already has some preallocated slots and that `lua_pushcfunction` does not allocate memory, so it cannot raise any error. (The prototype of the function `foo` is a requirement of `lua_pushcfunction`, which creates a function in Lua representing a C function. We will cover the details about C functions in Lua in the section called “C Functions”.)

Error handling in library code

Lua is a *safe* language. This means that no matter what we write in Lua, no matter how wrong it is, we can always understand the behavior of a program in terms of Lua itself. Moreover, errors are detected and explained in terms of Lua, too. You can contrast that with C, where the behavior of many wrong programs can be explained only in terms of the underlying hardware (e.g., error positions are given as instruction addresses).

Whenever we add new C functions to Lua, we can break its safety. For instance, a function equivalent to the BASIC command `poke`, which stores an arbitrary byte at an arbitrary memory address, could cause all sorts of memory corruption. We must strive to ensure that our add-ons are safe to Lua and provide good error handling.

As we discussed earlier, C programs have to set their error handling through `lua_pcall`. When we write library functions for Lua, however, usually they do not need to handle errors. Errors raised by a

library function will be caught either by a `pcall` in Lua or by a `lua_pcall` in the application code. So, whenever a function in a C library detects an error, it can simply call `lua_error` (or better yet `luaL_error`, which formats the error message and then calls `lua_error`). The function `lua_error` tidies any loose ends in the Lua system and jumps back to the protected call that originated that execution, passing along the error message.

Memory Allocation

The Lua core does not assume anything about how to allocate memory. It calls neither `malloc` nor `realloc` to allocate memory. Instead, it does all its memory allocation and deallocation through one single *allocation function*, which the user must provide when she creates a Lua state.

The function `luaL_newstate`, which we have been using to create states, is an auxiliary function that creates a Lua state with a default allocation function. This default allocation function uses the standard functions `malloc`–`realloc`–`free` from the C standard library, which are (or should be) good enough for most applications. However, it is quite easy to get full control over Lua allocation, by creating our state with the primitive `lua_newstate`:

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

This function takes two arguments: an allocation function and a user data. A state created in this way does all its allocation and deallocation by calling `f`; even the structure `lua_State` is allocated by `f`.

An allocation function must match the type `lua_Alloc`:

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

The first parameter is always the user data provided to `lua_newstate`; the second parameter is the address of the block being (re)allocated or released; the third parameter is the original block size; and the last parameter is the requested block size. If `ptr` is not `NULL`, Lua ensures that it was previously allocated with size `osize`. (When `ptr` is `NULL`, the previous size of the block was clearly zero, so Lua uses `osize` for some debug information.)

Lua uses `NULL` to represent a block of size zero. When `nsize` is zero, the allocation function must free the block pointed to by `ptr` and return `NULL`, which corresponds to a block of the required size (zero). When `ptr` is `NULL`, the function must allocate and return a block with the given size; if it cannot allocate the given block, it must return `NULL`. If `ptr` is `NULL` and `nsize` is zero, both rules apply: the net result is that the allocation function does nothing and returns `NULL`.

Finally, when `ptr` is non-`NULL` and `nsize` is non-zero, the allocation function should reallocate the block, like `realloc`, and return the new address (which may or may not be the same as the original). Again, in case of errors, it must return `NULL`. Lua assumes that the allocation function never fails when the new size is smaller than or equal to the old one. (Lua shrinks some structures during garbage collection, and it is unable to recover from errors there.)

The standard allocation function used by `luaL_newstate` has the following definition (extracted directly from the file `lauxlib.c`):

```
void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {
  (void)ud; (void)osize;  /* not used */
  if (nsize == 0) {
```

```
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

It assumes that `free(NULL)` does nothing and that `realloc(NULL, size)` is equivalent to `malloc(size)`. The ISO C standard mandates both behaviors.

We can recover the memory allocator of a Lua state by calling `lua_getallocf`:

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

If `ud` is not `NULL`, the function sets `*ud` with the value of the user data for this allocator. We can change the memory allocator of a Lua state by calling `lua_setallocf`:

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

Keep in mind that any new allocator will be responsible for freeing blocks that were allocated by the previous one. More often than not, the new function is a wrapper around the old one, for instance to trace allocations or to synchronize accesses to the heap.

Internally, Lua does not cache free memory blocks for reuse. It assumes that the allocation function does this caching; good allocators do. Lua also does not attempt to minimize fragmentation. Studies show that fragmentation is more the result of poor allocation strategies than of program behavior; good allocators do not create much fragmentation.

It is difficult to beat a well-implemented allocator, but sometimes you may try. For instance, Lua gives you the old size of any block that it frees or reallocates. Therefore, a specialized allocator does not need to keep information about the block size, reducing the memory overhead for each block.

Another situation where you can improve memory allocation is in multithreading systems. Such systems typically demand synchronization for their memory-allocation functions, as they use a global resource (the heap). However, the access to a Lua state must be synchronized too —or, better yet, restricted to one thread, as in our implementation of `lproc` in Chapter 33, *Threads and States*. So, if each Lua state allocates memory from a private pool, the allocator can avoid the costs of extra synchronization.

Exercises

Exercise 27.1: Compile and run the simple stand-alone interpreter (Figure 27.1, “A bare-bones stand-alone Lua interpreter”).

Exercise 27.2: Assume the stack is empty. What will be its contents after the following sequence of calls?

```
lua_pushnumber(L, 3.5);
lua_pushstring(L, "hello");
lua_pushnil(L);
lua_rotate(L, 1, -1);
lua_pushvalue(L, -2);
lua_remove(L, 1);
lua_insert(L, -2);
```

Exercise 27.3: Use the function `stackDump` (Figure 27.2, “Dumping the stack”) to check your answer to the previous exercise.

Exercise 27.4: Write a library that allows a script to limit the total amount of memory used by its Lua state. It may offer a single function, `setlimit`, to set that limit.

The library should set its own allocation function. This function, before calling the original allocator, checks the total memory in use and returns `NULL` if the requested memory exceeds the limit.

(Hint: the library can use the user data of the allocation function to keep its state: the byte count, the current memory limit, etc.; remember to use the original user data when calling the original allocation function.)

Chapter 28. Extending Your Application

An important use of Lua is as a *configuration* language. In this chapter, we will illustrate how we can use Lua to configure a program, starting with a simple example and evolving it to perform increasingly complex tasks.

The Basics

As our first task, let us imagine a simple configuration scenario: our C program has a window and we want the user to be able to specify the initial window size. Clearly, for such a simple task, there are several options simpler than using Lua, like environment variables or files with name-value pairs. But even using a simple text file, we have to parse it somehow; so, we decide to use a Lua configuration file (that is, a plain text file that happens to be a Lua program). In its simplest form, this file can contain something like the following:

```
-- define window size
width = 200
height = 300
```

Now, we must use the Lua API to direct Lua to parse this file and then to get the values of the global variables `width` and `height`. The function `load`, in Figure 28.1, “Getting user information from a configuration file”, does this job.

Figure 28.1. Getting user information from a configuration file

```
int getglobint (lua_State *L, const char *var) {
  int isnum, result;
  lua_getglobal(L, var);
  result = (int)lua_tointegerx(L, -1, &isnum);
  if (!isnum)
    error(L, "'%s' should be a number\n", var);
  lua_pop(L, 1);   /* remove result from the stack */
  return result;
}

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));
  *w = getglobint(L, "width");
  *h = getglobint(L, "height");
}
```

It assumes that we have already created a Lua state, following what we saw in the previous chapter. It calls `luaL_loadfile` to load the chunk from the file `fname`, and then calls `lua_pcall` to run the compiled chunk. In case of errors (e.g., a syntax error in our configuration file), these functions push the error message onto the stack and return a non-zero error code; our program then uses `lua_tostring` with index -1 to get the message from the top of the stack. (We defined the function `error` in the section called “A First Example”.)

After running the chunk, the program needs to get the values of the global variables. For that, it calls the auxiliary function `getglobint` (also in Figure 28.1, “Getting user information from a configuration

file") twice. This function first calls `lua_getglobal`, whose single parameter (besides the omnipresent `lua_State`) is the variable name, to push the corresponding global value onto the stack. Next, `getglobint` uses `lua_tointegerx` to convert this value to an integer, ensuring that it has the correct type.

Is it worth using Lua for that task? As I said before, for such a simple task, a simple file with only two numbers in it would be easier to use than Lua. Even so, the use of Lua brings some advantages. First, Lua handles all syntax details for us; our configuration file can even have comments! Second, the user is already able to do some complex configurations with it. For instance, the script may prompt the user for some information, or it can query an environment variable to choose a proper size:

```
-- configuration file
if getenv("DISPLAY") == ":0.0" then
  width = 300; height = 300
else
  width = 200; height = 200
end
```

Even in such simple configuration scenarios, it is hard to anticipate what users will want; but as long as the script defines the two variables, our C application works without changes.

A final reason for using Lua is that now it is easy to add new configuration facilities to our program; this ease fosters an attitude that results in programs that are more flexible.

Table Manipulation

Let us adopt that attitude: now, we want to configure a background color for the window, too. We will assume that the final color specification is composed of three numbers, where each number is a color component in RGB. Usually, in C, these numbers are integers in some range like *[0,255]*. In Lua, we will use the more natural range *[0,1]*.

A naive approach here is to ask the user to set each component in a different global variable:

```
-- configuration file
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

This approach has two drawbacks: it is too verbose (real programs may need dozens of different colors, for window background, window foreground, menu background, etc.); and there is no way to predefine common colors, so that, later, the user can simply write something like `background = WHITE`. To avoid these drawbacks, we will use a table to represent a color:

```
background = {red = 0.30, green = 0.10, blue = 0}
```

The use of tables gives more structure to the script; now it is easy for the user (or for the application) to predefine colors for later use in the configuration file:

```
BLUE = {red = 0, green = 0, blue = 1.0}
other color definitions

background = BLUE
```

To get these values in C, we can do as follows:

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
  error(L, "'background' is not a table");

red = getcolorfield(L, "red");
green = getcolorfield(L, "green");
blue = getcolorfield(L, "blue");
```

We first get the value of the global variable `background` and ensure that it is a table; then we use `getcolorfield` to get each color component.

Of course, the function `getcolorfield` is not part of the Lua API; we must define it. Again, we face the problem of polymorphism: there are potentially many versions of `getcolorfield` functions, varying the key type, value type, error handling, etc. The Lua API offers one function, `lua_gettable`, that works for all types. It takes the position of the table on the stack, pops the key from the stack, and pushes the corresponding value. Our private `getcolorfield`, defined in Figure 28.2, "A particular `getcolorfield` implementation",

Figure 28.2. A particular `getcolorfield` implementation

```
#define MAX_COLOR       255

/* assume that table is on the top of the stack */
int getcolorfield (lua_State *L, const char *key) {
  int result, isnum;
  lua_pushstring(L, key);  /* push key */
  lua_gettable(L, -2);  /* get background[key] */
  result = (int)(lua_tonumberx(L, -1, &isnum) * MAX_COLOR);
  if (!isnum)
    error(L, "invalid component '%s' in color", key);
  lua_pop(L, 1);  /* remove number */
  return result;
}
```

assumes that the table is on the top of the stack; so, after pushing the key with `lua_pushstring`, the table will be at index -2. Before returning, `getcolorfield` pops the retrieved value from the stack, leaving the stack at the same level that it was before the call.

We will extend our example a little further and introduce color names for the user. The user can still use color tables, but she can also use predefined names for the more common colors. To implement this feature, we need a color table in our C application:

```
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}  /* sentinel */
```

```
};
```

Our implementation will create global variables with the color names and initialize these variables using color tables. The result is the same as if the user had the following lines in her script:

```
WHITE = {red = 1.0, green = 1.0, blue = 1.0}
RED   = {red = 1.0, green = 0,   blue = 0}
other colors
```

To set the table fields, we define an auxiliary function, `setcolorfield`; it pushes the index and the field value on the stack, and then calls `lua_settable`:

```
/* assume that table is on top */
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);
}
```

Like other API functions, `lua_settable` works for many different types, so it gets all its operands from the stack. It takes the table index as an argument and pops the key and the value. The function `setcolorfield` assumes that before the call the table is on the top of the stack (index -1); after pushing the index and the value, the table will be at index -3.

The next function, `setcolor`, defines a single color. It creates a table, sets the appropriate fields, and assigns this table to the corresponding global variable:

```
void setcolor (lua_State *L, struct ColorTable *ct) {
  lua_newtable(L);                /* creates a table */
  setcolorfield(L, "red", ct->red);
  setcolorfield(L, "green", ct->green);
  setcolorfield(L, "blue", ct->blue);
  lua_setglobal(L, ct->name);      /* 'name' = table */
}
```

The function `lua_newtable` creates an empty table and pushes it on the stack; the three calls to `setcolorfield` set the table fields; finally, `lua_setglobal` pops the table and sets it as the value of the global with the given name.

With these previous functions, the following loop will register all colors for the configuration script:

```
int i = 0;
while (colortable[i].name != NULL)
  setcolor(L, &colortable[i++]);
```

Remember that the application must execute this loop before running the script.

Figure 28.3, “Colors as strings or tables” shows another option for implementing named colors.

Figure 28.3. Colors as strings or tables

```
lua_getglobal(L, "background");
if (lua_isstring(L, -1)) {   /* value is a string? */
  const char *name = lua_tostring(L, -1);  /* get string */
  int i;   /* search the color 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, "red");
  green = getcolorfield(L, "green");
  blue = getcolorfield(L, "blue");
} else
    error(L, "invalid value for 'background'");
```

Instead of global variables, the user can denote color names with strings, writing her settings as `background = "BLUE"`. Therefore, `background` can be either a table or a string. With this design, the application does not need to do anything before running the user's script. Instead, it needs more work to get a color. When it gets the value of the variable `background`, it must test whether the value is a string, and then look up the string in the color table.

What is the best option? In C programs, the use of strings to denote options is not a good practice, because the compiler cannot detect misspellings. In Lua, however, the error message for a misspelt color will probably be seen by the author of the configuration “program”. The distinction between programmer and user is blurred, and so the difference between a compilation error and a run-time error is blurred, too.

With strings, the value of `background` would be the misspelled string; hence, the application can add this information to the error message. The application can also compare strings regardless of case, so that a user can write `"white"`, `"WHITE"`, or even `"White"`. Moreover, if the user script is small and there are many colors, it may be inefficient to register hundreds of colors (and to create hundreds of tables and global variables) when the user needs only a few. With strings, we avoid this overhead.

Some short cuts

Although the C API strives for simplicity, Lua is not radical. So, the API offers short cuts for several common operations. Let us see some of them.

Because indexing a table with a string key is so common, Lua has a specialized version of `lua_gettable` for this case: `lua_getfield`. Using this function, we can rewrite the two lines

```
lua_pushstring(L, key);
lua_gettable(L, -2);  /* get background[key] */
```

in `getcolorfield` as

```
lua_getfield(L, -1, key);  /* get background[key] */
```

(As we do not push the string onto the stack, the table index is still -1 when we call `lua_getfield`.)

Because it is common to check the type of a value returned by `lua_gettable`, in Lua 5.3 this function (and similar ones like `lua_getfield`) now returns the type of its result. Therefore, we can simplify further the access and the check in `getcolorfield`:

```
  if (lua_getfield(L, -1, key) != LUA_TNUMBER)
    error(L, "invalid component in background color");
```

As you might expect, Lua offers also a specialized version of `lua_settable` for string keys, called `lua_setfield`. Using this function, we can rewrite our previous definition for `setcolorfield` as follows:

```
void setcolorfield (lua_State *L, const char *index, int value) {
  lua_pushnumber(L, (double)value / MAX_COLOR);
  lua_setfield(L, -2, index);
}
```

As a small optimization, we can also replace our use of `lua_newtable` in the function `setcolor`. Lua offers another function, `lua_createtable`, where we create a table and pre-allocate space for entries. Lua declares these functions like this:

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

#define lua_newtable(L)      lua_createtable(L, 0, 0)
```

The parameter `narr` is the expected number of elements in the sequence part of the table (that is, entries with sequential integer indices), and `nrec` is the expected number of other elements. In `setcolor`, we could write `lua_createtable(L, 0, 3)` as a hint that the table will get three entries. (Lua code does a similar optimization when we write a constructor.)

Calling Lua Functions

A great strength of Lua is that a configuration file can define functions to be called by the application. For instance, we can write in C an application to plot the graph of a function and define in Lua the function to be plotted.

The API protocol to call a function is simple: first, we push the function to be called; second, we push the arguments to the call; then we use `lua_pcall` to do the actual call; finally, we get the results from the stack.

As an example, let us assume that our configuration file has a function like this:

```
function f (x, y)
  return (x^2 * math.sin(y)) / (1 - x)
end
```

We want to evaluate, in C, `z = f(x, y)` for given `x` and `y`. Assuming that we have already opened the Lua library and run the configuration file, the function `f` in Figure 28.4, “Calling a Lua function from C” evaluates that code.

Figure 28.4. Calling a Lua function from C

```
/* call a function 'f' defined in Lua */
double f (lua_State *L, double x, double y) {
  int isnum;
  double z;

  /* push functions and arguments */
  lua_getglobal(L, "f");  /* function to be called */
  lua_pushnumber(L, x);   /* push 1st argument */
  lua_pushnumber(L, y);   /* push 2nd argument */

  /* do the call (2 arguments, 1 result) */
  if (lua_pcall(L, 2, 1, 0) != LUA_OK)
    error(L, "error running function 'f': %s",
             lua_tostring(L, -1));

  /* retrieve result */
  z = lua_tonumberx(L, -1, &isnum);
  if (!isnum)
    error(L, "function 'f' should return a number");
  lua_pop(L, 1);  /* pop returned value */
  return z;
}
```

The second and third arguments to `lua_pcall` are the number of arguments we are passing and the number of results we want. The fourth argument indicates a message-handling function; we will discuss it in a moment. As in a Lua assignment, `lua_pcall` adjusts the actual number of results to what we have asked for, pushing nils or discarding extra values as needed. Before pushing the results, `lua_pcall` removes from the stack the function and its arguments. When a function returns multiple results, the first result is pushed first; for instance, if there are three results, the first one will be at index -3 and the last at index -1.

If there is any error while `lua_pcall` is running, `lua_pcall` returns an error code; moreover, it pushes the error message on the stack (but still pops the function and its arguments). Before pushing the message, however, `lua_pcall` calls the message handler function, if there is one. To specify a message handler function, we use the last argument of `lua_pcall`. Zero means no message handler function; that is, the final error message is the original message. Otherwise, this argument should be the index on the stack where the message handler function is located. In such cases, we should push the handler on the stack somewhere below the function to be called.

For normal errors, `lua_pcall` returns the error code `LUA_ERRRUN`. Two special kinds of errors deserve different codes, because they never run the message handler. The first kind is a memory allocation error. For such errors, `lua_pcall` returns `LUA_ERRMEM`. The second kind is an error while Lua is running the message handler itself. In this case, it is of little use to call the handler again, so `lua_pcall` returns immediately with a code `LUA_ERRERR`. Since version 5.2, Lua differentiates a third kind of error: when a finalizer raises an error, `lua_pcall` returns the code `LUA_ERRGCMM` (*error in a GC metamethod*). This code indicates that the error is not directly related to the call itself.

A Generic Call Function

As a more advanced example, we will build a wrapper for calling Lua functions, using the `stdarg` facility in C. Our wrapper function, let us call it `call_va`, takes the name of a global function to be called, a

string describing the types of the arguments and results, then the list of arguments, and finally a list of pointers to variables to store the results; it handles all the details of the API. With this function, we could write our example in Figure 28.4, “Calling a Lua function from C” simply like this:

```
call_va(L, "f", "dd>d", x, y, &z);
```

The string `"dd>d"` means “two arguments of type double, one result of type double”. This descriptor can use the letters `d` for double, `i` for integer, and `s` for strings; a `>` separates arguments from the results. If the function has no results, the `>` is optional.

Figure 28.5, “A generic call function” shows the implementation of `call_va`.

Figure 28.5. A generic call function

```
#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 function */

  push and count arguments (Figure 28.6, “Pushing arguments for the generic

  nres = strlen(sig);  /* number of expected results */

  if (lua_pcall(L, narg, nres, 0) != 0)  /* do the call */
    error(L, "error calling '%s': %s", func,
                                       lua_tostring(L, -1));

  retrieve results (Figure 28.7, “Retrieving results for the generic call fu

  va_end(vl);
}
```

Despite its generality, this function follows the same steps of our first example: it pushes the function, pushes the arguments (Figure 28.6, “Pushing arguments for the generic call function”), does the call, and gets the results (Figure 28.7, “Retrieving results for the generic call function”).

Figure 28.6. Pushing arguments for the generic call function

```
for (narg = 0; *sig; narg++) {  /* repeat 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;  /* break the loop */

    default:
      error(L, "invalid option (%c)", *(sig - 1));
  }

}
endargs:
```

Figure 28.7. Retrieving results for the generic call function

```
nres = -nres;  /* stack index of first result */
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++;
}
```

Most of its code is straightforward, but there are some subtleties. First, it does not need to check whether `func` is a function: `lua_pcall` will trigger that error. Second, because it pushes an arbitrary number of arguments, it must ensure that there is enough stack space. Third, because the function can return strings, `call_va` cannot pop the results from the stack. It is up to the caller to pop them, after it finishes using any string results (or after copying them to appropriate buffers).

Exercises

Exercise 28.1: Write a C program that reads a Lua file defining a function `f` from numbers to numbers and plots that function. (You do not need to do anything fancy; the program can plot the results printing ASCII asterisks as we did in the section called “Compilation”.)

Exercise 28.2: Modify the function `call_va` (Figure 28.5, “A generic call function”) to handle Boolean values.

Exercise 28.3: Let us suppose a program that needs to monitor several weather stations. Internally, it uses a four-byte string to represent each station, and there is a configuration file to map each string to the actual URL of the corresponding station. A Lua configuration file could do this mapping in several ways:

- a bunch of global variables, one for each station;
- a table mapping string codes to URLs;
- a function mapping string codes to URLs.

Discuss the pros and cons of each option, considering things like the total number of stations, the regularity of the URLs (e.g., there may be a formation rule from codes to URLs), the kind of users, etc.

Chapter 29. Calling C from Lua

When we say that Lua can call C functions, this does not mean that Lua can call any C function.[1] As we saw in the previous chapter, when C calls a Lua function, it must follow a simple protocol to pass the arguments and to get the results. Similarly, for Lua to call a C function, the C function must follow a protocol to get its arguments and to return its results. Moreover, for Lua to call a C function, we must register the function, that is, we must give its address to Lua in an appropriate way.

When Lua calls a C function, it uses the same kind of stack that C uses to call Lua. The C function gets its arguments from the stack and pushes the results on the stack.

An important point here is that the stack is not a global structure; each function has its own private local stack. When Lua calls a C function, the first argument will always be at index 1 of this local stack. Even when a C function calls Lua code that calls the same (or another) C function again, each of these invocations sees only its own private stack, with its first argument at index 1.

C Functions

As a first example, let us see how to implement a simplified version of a function that returns the sine of a given number:

```
static int l_sin (lua_State *L) {
  double d = lua_tonumber(L, 1);  /* get argument */
  lua_pushnumber(L, sin(d));  /* push result */
  return 1;  /* number of results */
}
```

Any function registered with Lua must have this same prototype, defined in `lua.h` as `lua_CFunction`:

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

From the point of view of C, a C function gets as its single argument the Lua state and returns an integer with the number of values it is returning on the stack. Therefore, the function does not need to clear the stack before pushing its results. After it returns, Lua automatically saves its results and clears its entire stack.

Before we can call this function from Lua, we must register it. We do this bit of magic with `lua_pushcfunction`: it gets a pointer to a C function and creates a value of type `"function"` that represents this function inside Lua. Once registered, a C function behaves like any other function inside Lua.

A quick-and-dirty way to test our function `l_sin` is to put its code directly into our basic interpreter (Figure 27.1, “A bare-bones stand-alone Lua interpreter”) and add the following lines right after the call to `luaL_openlibs`:

```
lua_pushcfunction(L, l_sin);
lua_setglobal(L, "mysin");
```

The first line pushes a value of type function; the second line assigns it to the global variable `mysin`. After these modifications, we can use the new function `mysin` in our Lua scripts. In the next section, we will discuss better ways to link new C functions with Lua. Here, we will explore how to write better C functions.

[1] There are packages that allow Lua to call any C function, but they are neither as portable as Lua nor safe.

For a more professional sine function, we must check the type of its argument. The auxiliary library helps us with this task. The function `luaL_checknumber` checks whether a given argument is a number: in case of error, it throws an informative error message; otherwise, it returns the number. The modification to our function is 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 the above definition, if you call `mysin('a')`, you get an error like this one:

```
bad argument #1 to 'mysin' (number expected, got string)
```

The function `luaL_checknumber` automatically fills the message with the argument number (`#1`), the function name (`"mysin"`), the expected parameter type (`number`), and the actual parameter type (`string`).

As a more complex example, let us write a function that returns the contents of a given directory. Lua does not provide this function in its standard libraries, because ISO C does not offer functions for this job. Here, we will assume that we have a POSIX compliant system. Our function—we will call it `dir` in Lua, `l_dir` in C—gets as argument a string with the directory path and returns a list with the directory entries. For instance, a call like `dir("/home/lua")` may return the table `{".", "..", "src", "bin", "lib"}`. The complete code for this function is in Figure 29.1, “A function to read a directory”.

Figure 29.1. A function to read a directory

```
#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 the directory? */
    lua_pushnil(L);  /* return nil... */
    lua_pushstring(L, strerror(errno));  /* and error message */
    return 2;  /* number of results */
  }

  /* create result table */
  lua_newtable(L);
  i = 1;
  while ((entry = readdir(dir)) != NULL) {  /* for each entry */
    lua_pushinteger(L, i++);  /* push key */
    lua_pushstring(L, entry->d_name);  /* push value */
    lua_settable(L, -3);    /* table[i] = entry name */
  }

  closedir(dir);
  return 1;  /* table is already on top */
}
```

It starts getting the directory path with `luaL_checkstring`, which is the equivalent of `luaL_checknumber` for strings. Then it opens this directory with `opendir`. In case it cannot open the directory, the function returns nil plus an error message that it gets with `strerror`. After opening the directory, the function creates a new table and populates it with the directory entries. (Each time we call `readdir`, it returns a next entry.) Finally, it closes the directory and returns 1, in C, meaning that it is returning the value on top of its stack to Lua. (Remember that `lua_settable` pops the key and the value from the stack. Therefore, after the loop, the element on the top of the stack is the result table.)

In some conditions, this implementation of `l_dir` may cause a memory leak. Three of the Lua functions that it calls can fail due to insufficient memory: `lua_newtable`, `lua_pushstring`, and `lua_settable`. If any of these functions fails, it will raise an error and interrupt `l_dir`, which therefore will not call `closedir`. In Chapter 32, *Managing Resources*, we will see an alternative implementation for a directory function that corrects this problem.

Continuations

Through `lua_pcall` and `lua_call`, a C function called from Lua can call Lua back. Several functions in the standard library do that: `table.sort` can call an order function; `string.gsub` can call a re-

placement function; `pcall` and `xpcall` call functions in protected mode. If we remember that the main Lua code was itself called from C (the host program), we have a call sequence like C (host) calls Lua (script) that calls C (library) that calls Lua (callback).

Usually, Lua handles these sequences of calls without problems; after all, this integration with C is a hallmark of the language. There is one situation, however, where this interlacement can cause difficulties: coroutines.

Each coroutine in Lua has its own stack, which keeps information about the pending calls of the coroutine. Specifically, the stack stores the return address, the parameters, and the local variables of each call. For calls to Lua functions, the interpreter needs only this stack, which we call the *soft stack*. For calls to C functions, however, the interpreter must use the C stack, too. After all, the return address and the local variables of a C function live in the C stack.

It is easy for the interpreter to have multiple soft stacks, but the runtime of ISO C has only one internal stack. Therefore, coroutines in Lua cannot suspend the execution of a C function: if there is a C function in the call path from a resume to its respective yield, Lua cannot save the state of that C function to restore it in the next resume. Consider the next example, in Lua 5.1:

```
co = coroutine.wrap(function ()
                      print(pcall(coroutine.yield))
                    end)
co()
  --> false    attempt to yield across metamethod/C-call boundary
```

The function `pcall` is a C function; therefore, Lua 5.1 cannot suspend it, because there is no way in ISO C to suspend a C function and resume it later.

Lua 5.2 and later versions ameliorated that difficulty with *continuations*. Lua 5.2 implements yields using long jumps, in the same way that it implements errors. A long jump simply throws away any information about C functions in the C stack, so it is impossible to resume those functions. However, a C function `foo` can specify a continuation function `foo_k`, which is another C function to be called when it is time to resume `foo`. That is, when the interpreter detects that it should resume `foo`, but that a long jump threw away the entry for `foo` in the C stack, it calls `foo_k` instead.

To make things a little more concrete, let us see the implementation of `pcall` as an example. In Lua 5.1, this function had 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 == LUA_OK));  /* status */
  lua_insert(L, 1);  /* status is first result */
  return lua_gettop(L);  /* return status + all results */
}
```

If the function being called through `lua_pcall` yielded, it would be impossible to resume `luaB_pcall` later. Therefore, the interpreter raised an error whenever we attempted to yield inside a protected call. Lua 5.3 implements `pcall` roughly like in Figure 29.2, “Implementation of `pcall` with continuations”.[2]

[2]The API for continuations in Lua 5.2 is a little different. Check the reference manual for details.

Figure 29.2. Implementation of `pcall` with continuations

```
static int finishpcall (lua_State *L, int status, intptr_t ctx) {
  (void)ctx;   /* unused parameter */
  status = (status != LUA_OK && status != LUA_YIELD);
  lua_pushboolean(L, (status == 0));  /* status */
  lua_insert(L, 1);  /* status is first result */
  return lua_gettop(L);  /* return status + all results */
}

static int luaB_pcall (lua_State *L) {
  int status;
  luaL_checkany(L, 1);
  status = lua_pcallk(L, lua_gettop(L) - 1, LUA_MULTRET, 0,
                      0, finishpcall);
  return finishpcall(L, status, 0);
}
```

There are three important differences from the Lua 5.1 version: first, the new version replaces the call to `lua_pcall` by a call to `lua_pcallk`; second, it puts everything done after that call in a new auxiliary function `finishpcall`; third, the status returned by `lua_pcallk` can be `LUA_YIELD`, besides `LUA_OK` or an error.

If there are no yields, `lua_pcallk` works exactly like `lua_pcall`. If there is a yield, however, then things are quite different. If a function called by the original `lua_pcall` tries to yield, Lua 5.3 raises an error, like Lua 5.1. But when a function called by the new `lua_pcallk` yields, there is no error: Lua does a long jump and discards the entry for `luaB_pcall` from the C stack, but keeps in the soft stack of the coroutine a reference to the *continuation function* given to `lua_pcallk` (`finishpcall`, in our example). Later, when the interpreter detects that it should return to `luaB_pcall` (which is impossible), it instead calls the continuation function.

The continuation function `finishpcall` can also be called when there is an error. Unlike the original `luaB_pcall`, `finishpcall` cannot get the value returned by `lua_pcallk`. So, it gets this value as an extra parameter, `status`. When there are no errors, `status` is `LUA_YIELD` instead of `LUA_OK`, so that the continuation function can check how it is being called. In case of errors, `status` is the original error code.

Besides the status of the call, the continuation function also receives a *context*. The fifth parameter to `lua_pcallk` is an arbitrary integer that is passed as the last parameter to the continuation function. (The type of this parameter, `intptr_t`, allows pointers to be passed as context, too.) This value allows the original function to pass some arbitrary information directly to its continuation. (Our example does not use this facility.)

The continuation system of Lua 5.3 is an ingenious mechanism to support yields, but it is not a panacea. Some C functions would need to pass too much context to their continuations. Examples include `table.sort`, which uses the C stack for recursion, and `string.gsub`, which must keep track of captures and a buffer for its partial result. Although it is possible to rewrite them in a “yieldable” way, the gains do not seem to be worth the extra complexity and performance losses.

C Modules

A Lua module is a chunk that defines several Lua functions and stores them in appropriate places, typically as entries in a table. A C module for Lua mimics this behavior. Besides the definition of its C functions, it must also define a special function that plays the role of the main chunk in a Lua library. This function

should register all C functions of the module and store them in appropriate places, again typically as entries in a table. Like a Lua main chunk, it should also initialize anything else that needs initialization in the module.

Lua perceives C functions through this registration process. Once a C function is represented and stored in Lua, Lua calls it through a direct reference to its address (which is what we give to Lua when we register a function). In other words, Lua does not depend on a function name, package location, or visibility rules to call a function, once it is registered. Typically, a C module has one single public (extern) function, which is the function that opens the library. All other functions can be private, declared as `static` in C.

When we extend Lua with C functions, it is a good idea to design our code as a C module, even when we want to register only one C function: sooner or later (usually sooner) we will need other functions. As usual, the auxiliary library offers a helper function for this job. The macro `luaL_newlib` takes an array of C functions with their respective names and registers all of them inside a new table. As an example, suppose we want to create a library with the function `l_dir` that we defined earlier. First, we must define the library functions:

```
static int l_dir (lua_State *L) {
   as before
}
```

Next, we declare an array with all functions in the module with their respective names. This array has elements of type `luaL_Reg`, which is a structure with two fields: a function name (a string) and a function pointer.

```
static const struct luaL_Reg mylib [] = {
  {"dir", l_dir},
  {NULL, NULL}  /* sentinel */
};
```

In our example, there is only one function (`l_dir`) to declare. The last pair in the array is always `{NULL, NULL}`, to mark its end. Finally, we declare a main function, using `luaL_newlib`:

```
int luaopen_mylib (lua_State *L) {
  luaL_newlib(L, mylib);
  return 1;
}
```

The call to `luaL_newlib` creates a new table and fills it with the pairs name–function specified by the array `mylib`. When it returns, `luaL_newlib` leaves on the stack the new table wherein it opened the library. The function `luaopen_mylib` then returns 1 to return this table to Lua.

After finishing the library, we must link it to the interpreter. The most convenient way to do it is with the dynamic linking facility, if your Lua interpreter supports this facility. In this case, you must create a dynamic library with your code (`mylib.dll` in Windows, `mylib.so` in Linux-like systems) and put it somewhere in the C path. After these steps, you can load your library directly from Lua, with `require`:

```
local mylib = require "mylib"
```

This call links the dynamic library `mylib` with Lua, finds the function `luaopen_mylib`, registers it as a C function, and calls it, opening the module. (This behavior explains why `luaopen_mylib` must have the same prototype as any other C function.)

The dynamic linker must know the name of the function `luaopen_mylib` in order to find it. It will always look for `luaopen_` concatenated with the name of the module. Therefore, if our module is called

`mylib`, that function should be called `luaopen_mylib`. (We discussed the details of this function name in Chapter 17, *Modules and Packages*.)

If your interpreter does not support dynamic linking, then you have to recompile Lua with your new library. Besides this recompilation, you need some way of telling the stand-alone interpreter that it should open this library when it opens a new state. A simple way to do this is to add `luaopen_mylib` into the list of standard libraries to be opened by `luaL_openlibs`, in the file `linit.c`.

Exercises

Exercise 29.1: Write a variadic `summation` function, in C, that computes the sum of its variable number of 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 29.2: Implement a function equivalent to `table.pack`, from the standard library.

Exercise 29.3: Write a function that takes any number of parameters and returns them in reverse order.

```
print(reverse(1, "hello", 20))   --> 20    hello    1
```

Exercise 29.4: Write a function `foreach` that takes a table and a function and calls that function for each key–value pair in the table.

```
foreach({x = 10, y = 20}, print)
  --> x    10
  --> y    20
```

(Hint: check the function `lua_next` in the Lua manual.)

Exercise 29.5: Rewrite the function `foreach`, from the previous exercise, so that the function being called can yield.

Exercise 29.6: Create a C module with all functions from the previous exercises.

Chapter 30. Techniques for Writing C Functions

Both the official API and the auxiliary library provide several mechanisms to help writing C functions. In this chapter, we cover the mechanisms for array manipulation, string manipulation, and storing Lua values in C.

Array Manipulation

An "array", in Lua, is just a table used in a specific way. We can manipulate arrays using the same generic functions we use to manipulate tables, namely `lua_settable` and `lua_gettable`. However, the API provides special functions to access and update tables with integer keys:

```
void lua_geti (lua_State *L, int index, int key);
void lua_seti (lua_State *L, int index, int key);
```

Lua versions prior to 5.3 offered only raw versions of these functions, `lua_rawgeti` and `lua_rawseti`. They are similar to `lua_geti` and `lua_seti`, but do raw accesses (that is, without invoking metamethods). When the difference in unimportant (e.g., the table has no metamethods), the raw versions can be slightly faster.

The description of `lua_geti` and `lua_seti` is a little confusing, as it involves two indices: `index` refers to where the table is on the stack; `key` refers to where the element is in the table. The call `lua_geti(L, t, key)` is equivalent to the following sequence when `t` is positive (otherwise, we must compensate for the new item on the stack):

```
lua_pushnumber(L, key);
lua_gettable(L, t);
```

The call `lua_seti(L, t, key)` (again for `t` positive) is equivalent to this sequence:

```
lua_pushnumber(L, key);
lua_insert(L, -2);  /* put 'key' below previous value */
lua_settable(L, t);
```

As a concrete example of the use of these functions, Figure 30.1, "The function `map` in C" implements the function map: it applies a given function to all elements of an array, replacing each element by the result of the call.

Figure 30.1. The function `map` in C

```
int l_map (lua_State *L) {
  int i, n;

  /* 1st argument must be a table (t) */
  luaL_checktype(L, 1, LUA_TTABLE);

  /* 2nd argument must be a function (f) */
  luaL_checktype(L, 2, LUA_TFUNCTION);

  n = luaL_len(L, 1);  /* get size of table */

  for (i = 1; i <= n; i++) {
    lua_pushvalue(L, 2);   /* push f */
    lua_geti(L, 1, i);  /* push t[i] */
    lua_call(L, 1, 1);     /* call f(t[i]) */
    lua_seti(L, 1, i);  /* t[i] = result */
  }

  return 0;  /* no results */
}
```

This example also introduces three new functions: `luaL_checktype`, `luaL_len`, and `lua_call`.

The function `luaL_checktype` (from `lauxlib.h`) ensures that a given argument has a given type; otherwise, it raises an error.

The primitive `lua_len` (not used in the example) is equivalent to the length operator. Because of metamethods, this operator may result in any kind of object, not only numbers; therefore, `lua_len` returns its result on the stack. The function `luaL_len` (the one used in the example, from the auxiliary library) returns the length as an integer, raising an error if the coercion is not possible.

The function `lua_call` does an unprotected call. It is similar to `lua_pcall`, but it propagates errors, instead of returning an error code. When we are writing the main code in an application, we should not use `lua_call`, because we want to catch any errors. When we are writing functions, however, it is usually a good idea to use `lua_call`; if there is an error, just leave it to someone who cares about it.

String Manipulation

When a C function receives a string argument from Lua, there are only two rules that it must observe: not to pop the string from the stack while using it and never to modify the string.

Things get more demanding when a C function needs to create a string to return to Lua. Now, it is up to the C code to take care of buffer allocation/deallocation, buffer overflows, and other tasks that are difficult in C. So, the Lua API provides some functions to help with these tasks.

The standard API provides support for two of the most basic string operations: substring extraction and string concatenation. To extract a substring, remember that the basic operation `lua_pushlstring` gets the string length as an extra argument. Therefore, if we want to pass to Lua a substring of a string `s` ranging from position `i` to `j` (inclusive), all we have to do is this:

```
lua_pushlstring(L, s + i, j - i + 1);
```

As an example, suppose we want a function that splits a string according to a given separator (a single character) and returns a table with the substrings. For instance, the call `split("hi:ho:there", ":")` should return the table `{"hi", "ho", "there"}`. Figure 30.2, “Splitting a string” presents a simple implementation for this function.

Figure 30.2. Splitting a string

```
static int l_split (lua_State *L) {
  const char *s = luaL_checkstring(L, 1);      /* subject */
  const char *sep = luaL_checkstring(L, 2);  /* separator */
  const char *e;
  int i = 1;

  lua_newtable(L);  /* result table */

  /* repeat for each separator */
  while ((e = strchr(s, *sep)) != NULL) {
    lua_pushlstring(L, s, e - s);  /* push substring */
    lua_rawseti(L, -2, i++);   /* insert it in table */
    s = e + 1;  /* skip separator */
  }

  /* insert last substring */
  lua_pushstring(L, s);
  lua_rawseti(L, -2, i);

  return 1;  /* return the table */
}
```

It uses no buffers and can handle arbitrarily long strings: Lua takes care of all the memory allocation. (As we created the table, we know it has no metatable; so, we can manipulate it with the raw operations.)

To concatenate strings, Lua provides a specific function, called `lua_concat`. It is equivalent to the concatenation operator (`..`) in Lua: it converts numbers to strings and triggers metamethods when necessary. Moreover, it can concatenate more than two strings at once. The call `lua_concat(L, n)` will concatenate (and pop) the top-most `n` values on the stack and push the result.

Another helpful function is `lua_pushfstring`:

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

It is somewhat similar to the C function `sprintf`, in that it creates a string according to a format string and some extra arguments. Unlike `sprintf`, however, we do not need to provide a buffer. Lua dynamically creates the string for us, as large as it needs to be. The function pushes the resulting string on the stack and returns a pointer to it. This function accepts the following directives:

`%s`	inserts a zero-terminated string
`%d`	inserts an `int`
`%f`	inserts a Lua float
`%p`	inserts a pointer
`%I`	inserts a Lua integer
`%c`	inserts an `int` as a one-byte character

%U	inserts an `int` as a UTF-8 byte sequence
%%	inserts a percent sign

It accepts no modifiers, such as width or precision.[1]

Both `lua_concat` and `lua_pushfstring` are useful when we want to concatenate only a few strings. However, if we need to concatenate many strings (or characters) together, a one-by-one approach can be quite inefficient, as we saw in the section called “String Buffers”. Instead, we can use the *buffer facility* provided by the auxiliary library.

In its simpler usage, the buffer facility works with two functions: one gives us a buffer of any size where we can compose our string; the other converts the contents of the buffer into a Lua string.[2] Figure 30.3, “The function `string.upper`” illustrates those functions with the implementation of `string.upper`, right from the source file `lstrlib.c`.

Figure 30.3. The function `string.upper`

```
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;
}
```

The first step for using a buffer from the auxiliary library is to declare a variable with type `luaL_Buffer`. The next step is to call `luaL_buffinitsize` to get a pointer for a buffer with the given size; we can then use this buffer freely to create our string. The last step is to call `luaL_pushresultsize` to convert the buffer contents into a new Lua string and push that sting onto the stack. The size in this second 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 can be smaller. If we do not know the exact size of the resulting string, but have an upper bound, we can conservatively allocate a larger size.

Note that `luaL_pushresultsize` does not get a Lua state as its first argument. After the initialization, a buffer keeps a reference to the state, so we do not need to pass it when calling other functions that manipulate buffers.

We can also use the auxlib buffers by adding content to them piecemeal, without knowing an upper bound on the size of the result. The auxiliary library offers several functions to add things to a buffer: `luaL_addvalue` adds a Lua string that is on the top of the stack; `luaL_addlstring` adds strings with an explicit length; `luaL_addstring` adds zero-terminated strings; and `luaL_addchar` adds single characters. These functions 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);
```

[1] The directive `p` was introduced in Lua 5.2. The directives `I` and `U` were introduced in Lua 5.3.
[2] These two functions were introduced in Lua 5.2.

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

Figure 30.4, “A simplified implementation for `table.concat`” illustrates the use of these functions with a simplified implementation of the function `table.concat`.

Figure 30.4. A simplified implementation for `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_geti(L, 1, i);  /* get string from table */
    luaL_addvalue(b);   /* add it to the buffer */
  }
  luaL_pushresult(&b);
  return 1;
}
```

In that function, we first call `luaL_buffinit` to initialize the buffer. We then add elements to the buffer one by one, in this example using `luaL_addvalue`. Finally, `luaL_pushresult` flushes the buffer and leaves the final string on the top of the stack.

When we use the auxlib buffer, we have to worry about one detail. After we initialize a buffer, it may keep some internal data in the Lua stack. Therefore, we cannot assume that the stack top will remain where it was before we started using the buffer. Moreover, although we can use the stack for other tasks while using a buffer, the push/pop count for these uses must be balanced every time we access the buffer. The only exception to this rule is `luaL_addvalue`, which assumes that the string to be added to the buffer is on the top of the stack.

Storing State in C Functions

Frequently, C functions need to keep some non-local data, that is, data that outlive their invocation. In C, we typically use global (`extern`) or static variables for this need. When we are programming library functions for Lua, however, neither works well. First, we cannot store a generic Lua value in a C variable. Second, a library that uses such variables will not work with multiple Lua states.

A better approach is to get some help from Lua. A Lua function has two places to store non-local data: global variables and non-local variables. The C API offers two similar places to store non-local data: the registry and upvalues.

The registry

The *registry* is a global table that can be accessed only by C code.[3] Typically, we use it to store data to be shared among several modules.

The registry is always located at the *pseudo-index* `LUA_REGISTRYINDEX`. A pseudo-index is like an index into the stack, except that its associated value is not on the stack. Most functions in the Lua API

[3]Actually, we can access it from Lua using the debug function `debug.getregistry`, but we really should not use this function except for debugging.

that accept indices as arguments also accept pseudo-indices —the exceptions being those functions that manipulate the stack itself, such as `lua_remove` and `lua_insert`. For instance, to get a value stored with key `"Key"` in the registry, we can use the following call:

```
lua_getfield(L, LUA_REGISTRYINDEX, "Key");
```

The registry is a regular Lua table. As such, we can index it with any non-nil Lua value. However, because all C modules share the same registry, we must choose with care what values we use as keys, to avoid collisions. String keys are particularly useful when we want to allow other independent libraries to access our data, because all they need to know is the key name. For those keys, there is no bulletproof method of choosing names, but there are some good practices, such as avoiding common names and prefixing our names with the library name or something like it. (Prefixes like `lua` or `lualib` are not good choices.)

We should never use our own numbers as keys in the registry, because Lua reserves numeric keys for its *reference system*. This system comprises a pair of functions in the auxiliary library that allow us to store values in a table without worrying about how to create unique keys. The function `luaL_ref` creates new references:

```
int ref = luaL_ref(L, LUA_REGISTRYINDEX);
```

The previous call pops a value from the stack, stores it into the registry with a fresh integer key, and returns this key. We call this key a *reference*.

As the name implies, we use references mainly when we need to store a reference to a Lua value inside a C structure. As we have seen, we should never store pointers to Lua strings outside the C function that retrieved them. Moreover, Lua does not even offer pointers to other objects, such as tables or functions. So, we cannot refer to Lua objects through pointers. Instead, when we need such pointers, we create a reference and store it in C.

To push the value associated with a reference `ref` onto the stack, we simply write this:

```
lua_rawgeti(L, LUA_REGISTRYINDEX, ref);
```

Finally, to release both the value and the reference, we call `luaL_unref`:

```
luaL_unref(L, LUA_REGISTRYINDEX, ref);
```

After this call, a new call to `luaL_ref` may return this reference again.

The reference system treats nil as a special case. Whenever we call `luaL_ref` for a nil value, it does not create a new reference, but instead returns the constant reference `LUA_REFNIL`. The following call has no effect:

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

The next one pushes a nil, as expected:

```
lua_rawgeti(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

The reference system also defines the constant `LUA_NOREF`, which is an integer different from any valid reference. It is useful to signal that a value treated as a reference is invalid.

When we create a Lua state, the registry comes with two predefined references:

`LUA_RIDX_MAINTHREAD` keeps the Lua state itself, which is also its main thread.

`LUA_RIDX_GLOBALS` keeps the global environment.

Another safe way to create unique keys in the registry is to use as key the address of a static variable in our code: The C link editor ensures that this key is unique across all loaded libraries. To use this option, we need the function `lua_pushlightuserdata`, which pushes on the stack a value representing a C pointer. The following code shows how to store and retrieve a string from the registry using this method:

```
/* variable with a unique address */
static char Key = 'k';

/* store a string */
lua_pushlightuserdata(L, (void *)&Key);  /* push address */
lua_pushstring(L, myStr);  /* push value */
lua_settable(L, LUA_REGISTRYINDEX);  /* registry[&Key] = myStr */

/* retrieve a string */
lua_pushlightuserdata(L, (void *)&Key);  /* push address */
lua_gettable(L, LUA_REGISTRYINDEX);  /* retrieve value */
myStr = lua_tostring(L, -1);  /* convert to string */
```

We will discuss light userdata in more detail in the section called “Light Userdata”.

To simplify the use of variable addresses as unique keys, Lua 5.2 introduced two new functions: `lua_rawgetp` and `lua_rawsetp`. They are similar to `lua_rawgeti` and `lua_rawseti`, but they use C pointers (translated to light userdata) as keys. With them, we can write the previous code like this:

```
static char Key = 'k';

/* store a string */
lua_pushstring(L, myStr);
lua_rawsetp(L, LUA_REGISTRYINDEX, (void *)&Key);

/* retrieve a string */
lua_rawgetp(L, LUA_REGISTRYINDEX, (void *)&Key);
myStr = lua_tostring(L, -1);
```

Both functions use raw accesses. As the registry does not have a metatable, a raw access has the same behavior as a regular access, and it is slightly more efficient.

Upvalues

While the registry offers global variables, the *upvalue* mechanism implements an equivalent of C static variables that are visible only inside a particular function. Every time we create a new C function in Lua, we can associate with it any number of upvalues, each one holding a single Lua value. Later, when we call the function, it has free access to any of its upvalues, using pseudo-indices.

We call this association of a C function with its upvalues a *closure*. A C closure is a C approximation to a Lua closure. In particular, we can create different closures using the same function code, but with different upvalues.

To see a simple example, let us create a function `newCounter` in C. (We defined a similar function in Lua in Chapter 9, *Closures*.) This function is a factory: it returns a new counter function each time it is called, as in this example:

```
c1 = newCounter()
```

```
print(c1(), c1(), c1())   --> 1    2    3
c2 = newCounter()
print(c2(), c2(), c1())   --> 1    2    4
```

Although all counters share the same C code, each one keeps its own independent counter. The factory function is like this:

```
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 key function here is `lua_pushcclosure`, which creates a new closure. Its second argument is the base function (`counter`, in the example) and the third is the number of upvalues (1, in the example). Before creating a new closure, we must push on the stack the initial values for its upvalues. In our example, we push zero as the initial value for the single upvalue. As expected, `lua_pushcclosure` leaves the new closure on the stack, so the closure is ready to be returned as the result of `newCounter`.

Now, let us see the definition of `counter`:

```
static int counter (lua_State *L) {
  int val = lua_tointeger(L, lua_upvalueindex(1));
  lua_pushinteger(L, ++val);  /* new value */
  lua_copy(L, -1, lua_upvalueindex(1));  /* update upvalue */
  return 1;  /* return new value */
}
```

Here, the key element is the macro `lua_upvalueindex`, which produces the pseudo-index of an upvalue. In particular, the expression `lua_upvalueindex(1)` gives the pseudo-index of the first upvalue of the running function. Again, this pseudo-index is like any stack index, except that it does not live on the stack. So, the call to `lua_tointeger` retrieves the current value of the first (and only) upvalue as an integer. Then, the function `counter` pushes the new value `++val`, copies it as the new upvalue's value, and returns it.

As a more advanced example, we will implement tuples using upvalues. A *tuple* is a kind of constant structure with anonymous fields; we can retrieve a specific field with a numerical index, or we can retrieve all fields at once. In our implementation, we represent tuples as functions that store their values in their upvalues. When called with a numerical argument, the function returns that specific field. When called without arguments, it returns all its fields. The following code illustrates 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 will represent all tuples by the same function `t_tuple`, presented in Figure 30.5, “An implementation of tuples”.

Figure 30.5. An implementation of tuples

```
#include "lauxlib.h"

int t_tuple (lua_State *L) {
  lua_Integer op = luaL_optinteger(L, 1, 0);
  if (op == 0) {  /* no arguments? */
    int i;
    /* push each valid upvalue 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 */
  }
  else {  /* get field 'op' */
    luaL_argcheck(L, 0 < op && op <= 256, 1,
                     "index out of range");
    if (lua_isnone(L, lua_upvalueindex(op)))
      return 0;  /* no such field */
    lua_pushvalue(L, lua_upvalueindex(op));
    return 1;
  }
}

int t_new (lua_State *L) {
  int top = lua_gettop(L);
  luaL_argcheck(L, top < 256, top, "too many fields");
  lua_pushcclosure(L, t_tuple, top);
  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;
}
```

Because we can call a tuple with or without a numeric argument, `t_tuple` uses `luaL_optinteger` to get its optional argument. This function is similar to `luaL_checkinteger`, but it does not complain if the argument is absent; instead, it returns a given default value (0, in the example).

The maximum number of upvalues to a C function is 255, and the maximum index we can use with `lua_upvalueindex` is 256. So, we use `luaL_argcheck` to ensure these limits.

When we index a non-existent upvalue, the result is a pseudo-value whose type is `LUA_TNONE`. (When we access a stack index above the current top, we also get a pseudo-value with this type `LUA_TNONE`.) Our function `t_tuple` uses `lua_isnone` to test whether it has a given upvalue. However, we should never use `lua_upvalueindex` with a negative index or with an index greater than 256 (which is one plus the maximum number of upvalues for a C function), so we must check for this condition when the user provides the index. The function `luaL_argcheck` checks a given condition, raising an error with a nice message if the condition fails:

```
> t = tuple.new(2, 4, 5)
> t(300)
  --> stdin:1: bad argument #1 to 't' (index out of range)
```

The third argument to `luaL_argcheck` provides the argument number for the error message (1, in the example), and the fourth argument provides a complement to the message (`"index out of range"`).

The function to create tuples, `t_new` (also in Figure 30.5, “An implementation of tuples”), is trivial: because its arguments are already on the stack, it first checks that the number of fields respects the limit for upvalues in a closure and then call `lua_pushcclosure` to create a closure of `t_tuple` with all its arguments as upvalues. Finally, the array `tuplelib` and the function `luaopen_tuple` (also in Figure 30.5, “An implementation of tuples”) are the standard code to create a library `tuple` with that single function `new`.

Shared upvalues

Often, we need to share some values or variables among all functions in a library. Although we can use the registry for that task, we can also use upvalues.

Unlike Lua closures, C closures cannot share upvalues. Each closure has its own independent upvalues. However, we can set the upvalues of different functions to refer to a common table, so that this table becomes a common environment where the functions can share data.

Lua offers a function that eases the task of sharing an upvalue among all functions of a library. We have been opening C libraries with `luaL_newlib`. Lua implements this function as the following macro:

```
#define luaL_newlib(L,lib)  \
     (luaL_newlibtable(L,lib), luaL_setfuncs(L,lib,0))
```

The macro `luaL_newlibtable` just creates a new table for the library. (This table has a preallocated size equal to the number of functions in the given library.) The function `luaL_setfuncs` then adds the functions in the list `lib` to that new table, which is on the top of the stack.

The third parameter to `luaL_setfuncs` is what we are interested in here. It gives the number of shared upvalues the new functions in the library will have. The initial values for these upvalues should be on the stack, as happens with `lua_pushcclosure`. Therefore, to create a library where all functions share a common table as their single upvalue, we can use the following code:

```
/* create library table ('lib' is its list of functions) */
luaL_newlibtable(L, lib);
/* create shared upvalue */
lua_newtable(L);
/* add functions in list 'lib' to the new library, sharing
   previous table as upvalue */
luaL_setfuncs(L, lib, 1);
```

The last call also removes the shared table from the stack, leaving there only the new library.

Exercises

Exercise 30.1: Implement a filter function in C. It should receive a list and a predicate and return a new list with all elements from the given list that satisfy the predicate:

```
t = filter({1, 3, 20, -4, 5}, function (x) return x < 5 end)
-- t = {1, 3, -4}
```

(A predicate is just a function that tests some condition, returning a Boolean.)

Exercise 30.2: Modify the function `l_split` (from Figure 30.2, “Splitting a string”) so that it can work with strings containing zeros. (Among other changes, it should use `memchr` instead of `strchr`.)

Exercise 30.3: Reimplement the function `transliterate` (Exercise 10.3) in C.

Exercise 30.4: Implement a library with a modification of `transliterate` so that the transliteration table is not given as an argument, but instead is kept by the library. Your library should offer the following functions:

```
lib.settrans (table)   -- set the transliteration table
lib.gettrans ()        -- get the transliteration table
lib.transliterate(s)    -- transliterate 's' according to the
                           current table
```

Use the registry to keep the transliteration table.

Exercise 30.5: Repeat the previous exercise using an upvalue to keep the transliteration table.

Exercise 30.6: Do you think it is a good design to keep the transliteration table as part of the state of the library, instead of being a parameter to `transliterate`?

Chapter 31. 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 new types written in C. We will start with a small example; through the chapter, we will extend it with metamethods and other goodies.

Our running example in this chapter will be a quite simple type: Boolean arrays. The main motivation for this example is that it does not involve complex algorithms, so we can concentrate on API issues. Nevertheless, the example is useful by itself. Of course, we can use tables to implement arrays of Booleans in Lua. But a C implementation, where we store each entry in one single bit, uses less than 3% of the memory used by a table.

Our implementation will 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))
```

`BITS_PER_WORD` is the number of bits in an unsigned integer. The macro `I_WORD` computes the word that stores the bit corresponding to a given index, and `I_BIT` computes a mask to access the correct bit inside this word.

We will represent our arrays with the following structure:

```
typedef struct BitArray {
  int size;
  unsigned int values[1];  /* variable part */
} BitArray;
```

We declare the array `values` with size 1 only as a placeholder, because C 89 does not allow an array with size 0; we will set the actual size when we allocate the array. The next expression computes the total size for an array with `n` elements:

```
sizeof(BitArray) + I_WORD(n - 1) * sizeof(unsigned int)
```

(We subtract one from `n` because the original structure already includes space for one element.)

Userdata

In this first version, we will use explicit calls to set and get values, as in the next example:

```
a = array.new(1000)
for i = 1, 1000 do
  array.set(a, i, i % 2 == 0)     -- a[i] = (i % 2 == 0)
end
print(array.get(a, 10))           --> true
print(array.get(a, 11))           --> false
print(array.size(a))              --> 1000
```

Later we will see how to support both an object-oriented style, like `a:get(i)`, and a conventional syntax, like `a[i]`. For all versions, the underlying functions are the same, defined in Figure 31.1, “Manipulating a Boolean array”.

Figure 31.1. Manipulating a Boolean array

```
static int newarray (lua_State *L) {
  int i;
  size_t nbytes;
  BitArray *a;

  int n = (int)luaL_checkinteger(L, 1);   /* number of bits */
  luaL_argcheck(L, n >= 1, 1, "invalid size");
  nbytes = sizeof(BitArray) + I_WORD(n - 1)*sizeof(unsigned int);
  a = (BitArray *)lua_newuserdata(L, nbytes);

  a->size = n;
  for (i = 0; i <= I_WORD(n - 1); i++)
    a->values[i] = 0;  /* initialize array */

  return 1;  /* new userdata is already on the stack */
}

static int setarray (lua_State *L) {
  BitArray *a = (BitArray *)lua_touserdata(L, 1);
  int index = (int)luaL_checkinteger(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 bit */
  else
    a->values[I_WORD(index)] &= ~I_BIT(index);  /* reset bit */
  return 0;
}

static int getarray (lua_State *L) {
  BitArray *a = (BitArray *)lua_touserdata(L, 1);
  int index = (int)luaL_checkinteger(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;
}
```

Let us see them, bit by bit.

Our first concern is how to represent a C structure in Lua. Lua provides a basic type specifically for this task, called *userdata*. A userdata offers a raw memory area, with no predefined operations in Lua, which we can use to store anything.

The function `lua_newuserdata` allocates a block of memory with a given size, pushes the corresponding userdata on the stack, and returns the block address:

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

If for some reason we need to allocate memory by other means, it is very easy to create a userdata with the size of a pointer and to store there a pointer to the real memory block. We will see examples of this technique in Chapter 32, *Managing Resources*.

Our first function in Figure 31.1, "Manipulating a Boolean array", `newarray`, uses `lua_newuserdata` to create new arrays. Its code is straightforward. It checks its sole parameter (the array size, in bits), computes the array size in bytes, creates a userdata with the appropriate size, initializes its fields, and returns the userdata to Lua.

The next function is `setarray`, which receives three arguments: the array, the index, and the new value. It assumes that indices start at one, as usual in Lua. Because Lua accepts any value for a Boolean, we use `luaL_checkany` for the third parameter: it ensures only that there is a value (any value) for this parameter. If we call `setarray` with bad arguments, we get explanatory error messages, as in the following examples:

```
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 last function in Figure 31.1, "Manipulating a Boolean array" is `getarray`, the function to retrieve an entry. It is similar to `setarray`.

We will also define a function to retrieve the size of an array and some extra code to initialize our library; see Figure 31.2, "Extra code for the Boolean array library".

Figure 31.2. Extra code for the Boolean array library

```
static int getsize (lua_State *L) {
  BitArray *a = (BitArray *)lua_touserdata(L, 1);
  luaL_argcheck(L, a != NULL, 1, "'array' expected");
  lua_pushinteger(L, a->size);
  return 1;
}

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 `luaL_newlib`, from the auxiliary library. It creates a table and fills it with the pairs name–function specified by the array `arraylib`.

Metatables

Our current implementation has a major vulnerability. Suppose the user writes something like `array.set(io.stdin, 1, false)`. The value of `io.stdin` is a userdata with a pointer to a stream (`FILE *`). Because it is a userdata, `array.set` will gladly accept it as a valid argument; the probable result will be a memory corruption (with luck we will get an index-out-of-range error instead). Such behavior is unacceptable for any Lua library. No matter how we use a library, it should neither corrupt C data nor cause the Lua system to crash.

The usual method to distinguish one type of userdata from another is to create a unique metatable for that type. Every time we create a userdata, we mark it with the corresponding metatable; every time we get a userdata, we check whether it has the right metatable. Because Lua code cannot change the metatable of a userdata, it cannot deceive these checks.

We also need a place to store this new metatable, so that we can access it to create new userdata and to check whether a given userdata has the correct type. As we saw earlier, there are two options for storing the metatable: in the registry or as an upvalue for the functions in the library. It is customary, in Lua, to register any new C type into the registry, using a *type name* as the index and the metatable as the value. As with any other registry index, we must choose a type name with care, to avoid clashes. Our example will use the name `"LuaBook.array"` for its new type.

As usual, the auxiliary library offers some functions to help us here. The new auxiliary functions we will use are these:

```
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);
```

The function `luaL_newmetatable` creates a new table (to be used as a metatable), leaves the new table on the top of the stack, and maps the table to the given name in the registry. The function `luaL_getmetatable` retrieves the metatable associated with `tname` from the registry. Finally, `luaL_checkudata` checks whether the object at the given stack position is a userdata with a metatable that matches the given name. It raises an error if the object is not a userdata or if it does not have the correct metatable; otherwise, it returns the userdata address.

Now we can start our modifications. The first step is to change the function that opens the library so that it creates the metatable for arrays:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_newlib(L, arraylib);
  return 1;
}
```

The next step is to change `newarray` so that it sets this metatable in all arrays that it creates:

```
static int newarray (lua_State *L) {

  as before

  luaL_getmetatable(L, "LuaBook.array");
  lua_setmetatable(L, -2);
```

```
  return 1;  /* new userdata is already on the stack */
}
```

The function `lua_setmetatable` pops a table from the stack and sets it as the metatable of the object at the given index. In our case, this object is the new userdata.

Finally, `setarray`, `getarray`, and `getsize` have to check whether they have got a valid array as their first argument. To simplify their tasks, we define the following macro:

```
#define checkarray(L) \
          (BitArray *)luaL_checkudata(L, 1, "LuaBook.array")
```

Using this macro, the new definition for `getsize` is straightforward:

```
static int getsize (lua_State *L) {
  BitArray *a = checkarray(L);
  lua_pushinteger(L, a->size);
  return 1;
}
```

Because `setarray` and `getarray` also share code to read and check the index as their second argument, we factor out their common parts in a new auxiliary function (`getparams`).

Figure 31.3. New versions for `setarray`/`getarray`

```
static unsigned int *getparams (lua_State *L,
                                unsigned int *mask) {
  BitArray *a = checkarray(L);
  int index = (int)luaL_checkinteger(L, 2) - 1;

  luaL_argcheck(L, 0 <= index && index < a->size, 2,
                   "index out of range");

  *mask = I_BIT(index);  /* mask to access correct bit */
  return &a->values[I_WORD(index)]; /* word address */
}

static int setarray (lua_State *L) {
  unsigned int mask;
  unsigned int *entry = getparams(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 = getparams(L, &mask);
  lua_pushboolean(L, *entry & mask);
  return 1;
}
```

With this new function, `setarray` and `getarray` are straightforward, see Figure 31.3, “New versions for `setarray`/`getarray`”. Now, if we call them with an invalid userdata, we will get a proper error message:

```
a = array.get(io.stdin, 10)
--> bad argument #1 to 'get' (LuaBook.array expected, got FILE*)
```

Object-Oriented Access

Our next step is to transform our new type into an object, so that we can operate on its instances using the usual object-oriented syntax, like this:

```
a = array.new(1000)
print(a:size())     --> 1000
a:set(10, true)
print(a:get(10))    --> true
```

Remember that `a:size()` is equivalent to `a.size(a)`. Therefore, we have to arrange for the expression `a.size` to return our function `getsize`. The key mechanism here is the `__index` metamethod. For tables, Lua calls this metamethod whenever it cannot find a value for a given key. For userdata, Lua calls it in every access, because userdata have no keys at all.

Assume that we run the following code:

```
do
  local metaarray = getmetatable(array.new(1))
  metaarray.__index = metaarray
  metaarray.set = array.set
  metaarray.get = array.get
  metaarray.size = array.size
end
```

In the first line, we create an array only to get its metatable, which we assign to `metaarray`. (We cannot set the metatable of a userdata from Lua, but we can get it.) Then we set `metaarray.__index` to `metaarray`. When we evaluate `a.size`, Lua cannot find the key `"size"` in the object `a`, because the object is a userdata. Therefore, Lua tries to get this value from the field `__index` of the metatable of `a`, which happens to be `metaarray` itself. But `metaarray.size` is `array.size`, so `a.size(a)` results in `array.size(a)`, as we wanted.

Of course, we can write the same thing in C. We can do even better: now that arrays are objects, with their own operations, we do not need to have these operations in the table `array` anymore. The only function that our library still has to export is `new`, to create new arrays. All other operations come only as methods. The C code can register them directly as such.

The operations `getsize`, `getarray`, and `setarray` do not change from our previous approach. What will change is how we register them. That is, we have to change the code that opens the library. First, we need two separate function lists: 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 open function `luaopen_array` has to create the metatable, assign it to its own `__index` field, register all the methods there, and create and fill the `array` table:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");  /* create metatable */
  lua_pushvalue(L, -1);  /* duplicate the metatable */
  lua_setfield(L, -2, "__index");  /* mt.__index = mt */
  luaL_setfuncs(L, arraylib_m, 0);  /* register metamethods */
  luaL_newlib(L, arraylib_f);  /* create lib table */
  return 1;
}
```

Here we use `luaL_setfuncs` again, to set the functions from the list `arraylib_m` into the metatable, which is on the top of the stack. Then we call `luaL_newlib` to create a new table and register the functions from the list `arraylib_f` there.

As a final touch, we will add a `__tostring` method to our new type, so that `print(a)` prints `"array"` plus the size of the array inside parentheses. The function itself is here:

```
int array2string (lua_State *L) {
  BitArray *a = checkarray(L);
  lua_pushfstring(L, "array(%d)", a->size);
  return 1;
}
```

The call to `lua_pushfstring` formats the string and leaves it on the top of the stack. We also have to add `array2string` to the list `arraylib_m`, to include it in the metatable of array objects:

```
static const struct luaL_Reg arraylib_m [] = {
  {"__tostring", array2string},
  other methods
};
```

Array Access

A better alternative to the object-oriented notation is to use a regular array notation to access our arrays. Instead of writing `a:get(i)`, we could simply write `a[i]`. For our example, this is easy to do, because our functions `setarray` and `getarray` already receive their arguments in the order that they are given to the corresponding metamethods. A quick solution is to define these metamethods directly in Lua:

```
local metaarray = getmetatable(array.new(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
metaarray.__len = array.size
```

(We must run this code on the original implementation for arrays, without the modifications for object-oriented access.) That is all we need to use the standard syntax:

```
a = array.new(1000)
a[10] = true         -- 'setarray'
```

```
print(a[10])          -- 'getarray'   --> true
print(#a)             -- 'getsize'    --> 1000
```

If we prefer, we can register these metamethods in our C code. For this, we again modify our initialization function; see Figure 31.4, “New initialization code for the Bit Array library”.

Figure 31.4. New initialization code for the Bit Array library

```
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 new version, again we have only one public function, `new`. All other functions are available only as metamethods for specific operations.

Light Userdata

The kind of userdata that we have been using until now is called *full userdata*. Lua offers another kind of userdata, called *light userdata*.

A light userdata is a value that represents a C pointer, that is, a `void *` value. A light userdata is a value, not an object; we do not create them (in the same way that we do not create numbers). To put a light userdata onto the stack, we call `lua_pushlightuserdata`:

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

Despite their common name, light userdata and full userdata are quite different things. Light userdata are not buffers, but bare pointers. They have no metatables. Like numbers, light userdata are not managed by the garbage collector.

Sometimes, people use light userdata as a cheap alternative to full userdata. This is not a typical use, however. First, light userdata do not have metatables, so there is no way to know their types. Second, despite the name, full userdata are inexpensive, too. They add little overhead compared to a `malloc` for the given memory size.

The real use of light userdata comes from equality. As a full userdata is an object, it is only equal to itself. A light userdata, on the other hand, represents a C pointer value. As such, it is equal to any userdata that represents the same pointer. Therefore, we can use light userdata to find C objects inside Lua.

We have already seen a typical use of light userdata, as keys in the registry (the section called “The registry”). There, the equality of light userdata was fundamental. Every time we push the same address with `lua_pushlightuserdata`, we get the same Lua value and, therefore, the same entry in the registry.

Another typical scenario in Lua is to have Lua objects acting as proxies to corresponding C objects. For instance, the I/O library uses Lua userdata to represent C streams inside Lua. When the action goes from Lua to C, the mapping from the Lua object to the C object is easy. Again using the example of the I/O library, each Lua stream keeps a pointer to its corresponding C stream. However, when the action goes from C to Lua, the mapping can be tricky. As an example, suppose we have some kind of callback in our I/O system (e.g., to tell that there is data to be read). The callback receives the C stream where is should operate. From that, how can we find its corresponding Lua object? Because the C stream is defined by the C standard library, not by us, we cannot store anything there.

Light userdata provide a nice solution for this mapping. We keep a table where the indices are light userdata with the stream addresses, and the values are the full userdata that represent the streams in Lua. In a callback, once we have a stream address, we use it —as a light userdata— as an index into that table to retrieve its corresponding Lua object. (The table should probably have weak values; otherwise, those full userdata would never be collected.)

Exercises

Exercise 31.1: Modify the implementation of `setarray` so that it accepts only Boolean values.

Exercise 31.2: We can see a Boolean array as a set of integers (the indices with true values in the array). Add to the implementation of Boolean arrays functions to compute the union and intersection of two arrays. These functions should receive two arrays and return a new one, without modifying its parameters.

Exercise 31.3: Extend the previous exercise so that we can use addition to get the union of two arrays and multiplication for the intersection.

Exercise 31.4: Modify the implementation of the `__tostring` metamethod so that it shows the full contents of the array in an appropriate way. Use the buffer facility (the section called “String Manipulation”) to create the resulting string.

Exercise 31.5: Based on the example for Boolean arrays, implement a small C library for integer arrays.

Chapter 32. Managing Resources

In our implementation of Boolean arrays in the previous chapter, we did not need to worry about managing resources. Those arrays need only memory. Each userdata representing an array has its own memory, which is managed by Lua. When an array becomes garbage (that is, inaccessible by the program), Lua eventually collects it and frees its memory.

Life is not always that easy. Sometimes, an object needs other resources besides raw memory, such as file descriptors, window handles, and the like. (Often these resources are just memory too, but managed by some other part of the system.) In such cases, when the object becomes garbage and is collected, somehow these other resources must be released too.

As we saw in the section called “Finalizers”, Lua provides finalizers in the form of the `__gc` metamethod. To illustrate the use of this metamethod in C and of the API as a whole, in this chapter we will develop two Lua bindings for external facilities. The first example is another implementation for a function to traverse a directory. The second (and more substantial) example is a binding to *Expat*, an open source XML parser.

A Directory Iterator

In the section called “C Functions”, we implemented a function `dir` to traverse directories that returned a table with all files from a given directory. Our new implementation will return an iterator that returns a new entry each time it is called. With this new implementation, we will be able to traverse a directory with a loop like this:

```
for fname in dir.open(".") do
  print(fname)
end
```

To iterate over a directory, in C, we need a `DIR` structure. Instances of `DIR` are created by `opendir` and must be explicitly released with a call to `closedir`. Our previous implementation kept its `DIR` instance as a local variable and closed this instance after retrieving the last file name. Our new implementation cannot keep this `DIR` instance in a local variable, because it must query this value over several calls. Moreover, it cannot close the directory only after retrieving the last name; if the program breaks the loop, the iterator will never retrieve this last name. Therefore, to make sure that the `DIR` instance is always released, we will store its address in a userdata and use the `__gc` metamethod of this userdata to release the directory structure.

Despite its central role in our implementation, this userdata representing a directory does not need to be visible to Lua. The function `dir.open` returns an iterator function, and this function is what Lua sees. The directory can be an upvalue of the iterator function. As such, the iterator function has direct access to this structure, but Lua code does not (and does not need to).

In all, we need three C functions. First, we need the function `dir.open`, a factory function that Lua calls to create iterators; it must open a `DIR` structure and create a closure of the iterator function with this structure as an upvalue. Second, we need the iterator function. Third, we need the `__gc` metamethod, which closes a `DIR` structure. As usual, we also need an extra function to make initial arrangements, such as to create and initialize a metatable for directories.

Let us start our code with the function `dir.open`, shown in Figure 32.1, “The `dir.open` factory function”.

Figure 32.1. The `dir.open` factory function

```
#include <dirent.h>
#include <errno.h>
#include <string.h>

#include "lua.h"
#include "lauxlib.h"

/* forward declaration for the iterator 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 to store a DIR address */
  DIR **d = (DIR **)lua_newuserdata(L, sizeof(DIR *));

  /* pre-initialize it */
  *d = NULL;

  /* set its metatable */
  luaL_getmetatable(L, "LuaBook.dir");
  lua_setmetatable(L, -2);

  /* try to open the given directory */
  *d = opendir(path);
  if (*d == NULL)  /* error opening the directory? */
    luaL_error(L, "cannot open %s: %s", path, strerror(errno));

  /* creates and returns the iterator function;
     its sole upvalue, the directory userdata,
     is already on the top of the stack */
  lua_pushcclosure(L, dir_iter, 1);
  return 1;
}
```

A subtle point in this function is that it must create the userdata before opening the directory. If it first opens the directory, and then the call to `lua_newuserdata` raises a memory error, the function loses and leaks the `DIR` structure. With the correct order, the `DIR` structure, once created, is immediately associated with the userdata; whatever happens after that, the `__gc` metamethod will eventually release the structure.

Another subtle point is the consistency of the userdata. Once we set its metatable, the `__gc` metamethod will be called no matter what. So, before setting the metatable, we pre-initialize the userdata with `NULL` to ensure that it has some well-defined value.

The next function is `dir_iter` (in Figure 32.2, “Other functions for the `dir` library”), the iterator itself.

Figure 32.2. Other functions for the `dir` library

```
static int dir_iter (lua_State *L) {
  DIR *d = *(DIR **)lua_touserdata(L, lua_upvalueindex(1));
  struct dirent *entry = readdir(d);
  if (entry != NULL) {
    lua_pushstring(L, entry->d_name);
    return 1;
  }
  else return 0;  /* no more values to return */
}

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 its __gc field */
  lua_pushcfunction(L, dir_gc);
  lua_setfield(L, -2, "__gc");

  /* create the library */
  luaL_newlib(L, dirlib);
  return 1;
}
```

Its code is straightforward. It gets the `DIR` structure's address from its upvalue and calls `readdir` to read the next entry.

The function `dir_gc` (also in Figure 32.2, “Other functions for the `dir` library”) is the `__gc` metamethod. This metamethod closes a directory. As we mentioned before, it must take one precaution: in case of errors in the initialization, the directory can be `NULL`.

The last function in Figure 32.2, “Other functions for the `dir` library”, `luaopen_dir`, is the function that opens this one-function library.

This complete example has an interesting subtlety. At first, it may seem that `dir_gc` should check whether its argument is a directory and whether it has not been closed already. Otherwise, a malicious user could call it with another kind of userdata (a file, for instance) or finalize a directory twice, with disastrous consequences. However, there is no way for a Lua program to access this function: it is stored only in the metatable of directories, which in turn are stored as upvalues of the iteration functions. Lua programs cannot access these directories.

An XML Parser

Now we will look at a simplified implementation of a Lua binding for Expat, which we will call `lxp`. Expat is an open source XML 1.0 parser written in C. It implements SAX, the *Simple API for XML*. SAX is an event-based API. This means that a SAX parser reads an XML document and, as it goes, reports to the application what it finds, through callbacks. For instance, if we instruct Expat to parse a string like `"<tag cap="5">hi</tag>"`, it will generate three events: a *start-element* event, when it reads the substring `"<tag cap="5">"`; a *text* event (also called a *character data* event), when it reads `"hi"`; and an *end-element* event, when it reads `"</tag>"`. Each of these events calls an appropriate *callback handler* in the application.

Here we will not cover the entire Expat library. We will concentrate only on those parts that illustrate new techniques for interacting with Lua. Although Expat handles more than a dozen different events, we will consider only the three events that we saw in the previous example (start elements, end elements, and text).[1]

The part of the Expat API that we need for this example is small. First, we need the functions to create and destroy an Expat parser:

```
XML_Parser XML_ParserCreate (const char *encoding);
void XML_ParserFree (XML_Parser p);
```

The `encoding` argument is optional; we will use `NULL` in our binding.

After we have a parser, we must register its callback handlers:

```
void XML_SetElementHandler(XML_Parser p,
                           XML_StartElementHandler start,
                           XML_EndElementHandler end);

void XML_SetCharacterDataHandler(XML_Parser p,
                                 XML_CharacterDataHandler hndl);
```

The first function registers handlers for start and end elements. The second function registers handlers for text (*character data*, in XML parlance).

All callback handlers take a user data as their first parameter. The start-element handler receives also the tag name and its attributes:

```
typedef void (*XML_StartElementHandler)(void *uData,
                                        const char *name,
                                        const char **atts);
```

The attributes come as a NULL-terminated array of strings, where each pair of consecutive strings holds an attribute name and its value. The end-element handler has only one extra parameter, the tag name:

```
typedef void (*XML_EndElementHandler)(void *uData,
                                      const char *name);
```

Finally, a text handler receives only the text as an extra parameter. This text string is not null-terminated; instead, it has an explicit length:

```
typedef void (*XML_CharacterDataHandler)(void *uData,
```

[1] The package `LuaExpat` offers a quite complete interface to Expat.

```
                                        const char *s,
                                        int len);
```

To feed text to Expat, 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 in pieces, through successive calls to the function `XML_Parse`. The last argument to `XML_Parse`, the Boolean `isLast`, informs Expat whether that piece is the last one of a document. This function returns zero if it detects a parse error. (Expat also provides functions to retrieve error information, but we will ignore them here, for the sake of simplicity.)

The last function we need from Expat allows us to set the user data that will be passed to the handlers:

```
void XML_SetUserData (XML_Parser p, void *uData);
```

Now let us have a look at how we can use this library in Lua. A first approach is a direct approach: simply export all those functions to Lua. A better approach is to adapt the functionality to Lua. For instance, because Lua is untyped, we do not need different functions to set each kind of callback. Better yet, we can avoid the callback registering functions altogether. Instead, when we create a parser, we give a callback table that contains all callback handlers, each with an appropriate key related to its corresponding event. For instance, if we want to print a layout of a document, we could use the following callback table:

```
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,
}
```

Fed with the input `"<to> <yes/> </to>"`, these handlers would print this output:

```
+ to
+   yes
-   yes
- to
```

With this API, we do not need functions to manipulate callbacks. We manipulate them directly in the callback table. Thus, the whole API needs only three functions: one to create parsers, one to parse a piece of text, and one to close a parser. Actually, we will implement the last two functions as methods of parser objects. A typical use of the API could be like this:

```
local lxp = require "lxp"

p = lxp.new(callbacks)       -- create new parser

for l in io.lines() do       -- iterate over input lines
```

```
  assert(p:parse(l))         -- parse the line
  assert(p:parse("\n"))      -- add newline
end

assert(p:parse())            -- finish document
p:close()                    -- close parser
```

Now let us turn our attention to the implementation. The first decision is how to represent a parser in Lua. It is quite natural to use a userdata containing a C structure, but what do we need to put in it? We need at least the actual Expat parser and the callback table. We must also store a Lua state, because these parser objects are all that an Expat callback receives, and the callbacks need to call Lua. We can store the Expat parser and the Lua state (which are C values) directly in a C structure. For the callback table, which is a Lua value, one option is to create a reference to it in the registry and store that reference. (We will explore this option in Exercise 32.2). Another option is to use a *user value*. Each userdata can have one single Lua value directly associated with it; this value is called a user value.[2] With this option, the definition for a parser object is as follows:

```
#include <stdlib.h>
#include "expat.h"
#include "lua.h"
#include "lauxlib.h"

typedef struct lxp_userdata {
  XML_Parser parser;          /* associated expat parser */
  lua_State *L;
} lxp_userdata;
```

The next step is the function that creates parser objects, `lxp_make_parser`. Figure 32.3, “Function to create XML parser objects” shows its code.

[2] In Lua 5.2, this user value must be table.

Figure 32.3. Function to create XML parser objects

```
/* forward declarations for callback 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 a parser object */
  lxp_userdata *xpu = (lxp_userdata *)lua_newuserdata(L,
                                        sizeof(lxp_userdata));

  /* pre-initialize it, in case of error */
  xpu->parser = NULL;

  /* set its metatable */
  luaL_getmetatable(L, "Expat");
  lua_setmetatable(L, -2);

  /* (2) create the Expat parser */
  p = xpu->parser = XML_ParserCreate(NULL);
  if (!p)
    luaL_error(L, "XML_ParserCreate failed");

  /* (3) check and store the callback table */
  luaL_checktype(L, 1, LUA_TTABLE);
  lua_pushvalue(L, 1);  /* push table */
  lua_setuservalue(L, -2);   /* set it as the user value */

  /* (4) configure Expat parser */
  XML_SetUserData(p, xpu);
  XML_SetElementHandler(p, f_StartElement, f_EndElement);
  XML_SetCharacterDataHandler(p, f_CharData);
  return 1;
}
```

This function has four main steps:

- Its first step follows a common pattern: it first creates a userdata; then it pre-initializes the userdata with consistent values; and finally it sets its metatable. (The pre-initialization ensures that if there is any error during the initialization, the finalizer will find the userdata in a consistent state.)

- In step 2, the function creates an Expat parser, stores it in the userdata, and checks for errors.

- Step 3 ensures that the first argument to the function is actually a table (the callback table), and sets it as the user value for the new userdata.

- The last step initializes the Expat parser. It sets the userdata as the object to be passed to the callback functions and it sets the callback functions. Notice that these callback functions are the same for all parsers; after all, it is impossible to dynamically create new functions in C. Instead, those fixed C functions will use the callback table to decide which Lua functions they should call each time.

The next step is the parse method `lxp_parse` (Figure 32.4, “Function to parse an XML fragment”), which parses a piece of XML data.

Figure 32.4. Function to parse an XML fragment

```
static int lxp_parse (lua_State *L) {
  int status;
  size_t len;
  const char *s;
  lxp_userdata *xpu;

  /* get and check first argument (should be a parser) */
  xpu = (lxp_userdata *)luaL_checkudata(L, 1, "Expat");

  /* check if it is not closed */
  luaL_argcheck(L, xpu->parser != NULL, 1, "parser is closed");

  /* get second argument (a string) */
  s = luaL_optlstring(L, 2, NULL, &len);

  /* put callback table at stack index 3 */
  lua_settop(L, 2);
  lua_getuservalue(L, 1);

  xpu->L = L;  /* set Lua state */

  /* call Expat to parse string */
  status = XML_Parse(xpu->parser, s, (int)len, s == NULL);

  /* return error code */
  lua_pushboolean(L, status);
  return 1;
}
```

It gets two arguments: the parser object (the *self* of the method) and an optional piece of XML data. When called without any data, it informs Expat that the document has no more parts.

When `lxp_parse` calls `XML_Parse`, the latter function will call the handlers for each relevant element that it finds in the given piece of document. These handlers will need to access the callback table, so `lxp_parse` puts this table at stack index three (right after the parameters). There is one more detail in the call to `XML_Parse`: remember that the last argument to this function tells Expat whether the given piece of text is the last one. When we call `parse` without an argument, `s` will be `NULL`, so this last argument will be true.

Now let us turn our attention to the functions `f_CharData`, `f_StartElement`, and `f_EndElement`, which handle the callbacks. All these three functions have a similar structure: each checks whether the callback table defines a Lua handler for its specific event and, if so, prepares the arguments and then calls this Lua handler.

Let us see first the `f_CharData` handler, in Figure 32.5, “Handler for character data”.

Figure 32.5. Handler for character data

```
static void f_CharData (void *ud, const char *s, int len) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  /* get handler from callback table */
  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') */
  lua_pushlstring(L, s, len);  /* push Char data */
  lua_call(L, 2, 0);  /* call the handler */
}
```

Its code is quite simple. The handler receives a `lxp_userdata` structure as its first argument, due to our call to `XML_SetUserData` when we created the parser. After retrieving the Lua state, the handler can access the callback table at stack index 3, as set by `lxp_parse`, and the parser itself at stack index 1. Then it calls its corresponding handler in Lua (when present), with two arguments: the parser and the character data (a string).

The `f_EndElement` handler is quite similar to `f_CharData`; see Figure 32.6, “Handler for end elements”.

Figure 32.6. Handler for end elements

```
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') */
  lua_pushstring(L, name);  /* push tag name */
  lua_call(L, 2, 0);  /* call the handler */
}
```

It also calls its corresponding Lua handler with two arguments —the parser and the tag name (again a string, but now null-terminated).

Figure 32.7, “Handler for start elements” shows the last handler, `f_StartElement`.

Figure 32.7. Handler for start elements

```
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') */
  lua_pushstring(L, name);  /* push tag name */

  /* create and fill 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 the handler */
}
```

It calls the Lua handler with three arguments: the parser, the tag name, and a list of attributes. This handler is a little more complex than the others, because it needs to translate the tag's list of attributes into Lua. It uses a quite natural translation, building a table that maps attribute names to their values. For instance, a start tag like

```
<to method="post" priority="high">
```

generates the following table of attributes:

```
{method = "post", priority = "high"}
```

The last method for parsers is `close`, in Figure 32.8, “Method to close an XML parser”.

Figure 32.8. Method to close an XML parser

```
static int lxp_close (lua_State *L) {
  lxp_userdata *xpu =
               (lxp_userdata *)luaL_checkudata(L, 1, "Expat");

  /* free Expat parser (if there is one) */
  if (xpu->parser)
    XML_ParserFree(xpu->parser);
  xpu->parser = NULL;  /* avoids closing it again */
  return 0;
}
```

When we close a parser, we have to free its resources, namely the Expat structure. Remember that, due to occasional errors during its creation, a parser may not have this resource. Notice how we keep the parser in a consistent state as we close it, so there is no problem if we try to close it again or when the garbage

collector finalizes it. Actually, we will use exactly this function as the finalizer. This ensures that every parser eventually frees its resources, even if the programmer does not close it.

Figure 32.9, “Initialization code for the `lxp` library” is the final step: it shows `luaopen_lxp`, which opens the library, putting all previous parts together.

Figure 32.9. Initialization code for the `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 (only lxp.new) */
  luaL_newlib(L, lxp_funcs);
  return 1;
}
```

We use here the same scheme that we used in the object-oriented Boolean-array example from the section called “Object-Oriented Access”: we create a metatable, make its `__index` field point to itself, and put all the methods inside it. For that, we need a list with the parser methods (`lxp_meths`). We also need a list with the functions of this library (`lxp_funcs`). As is common with object-oriented libraries, this list has a single function, which creates new parsers.

Exercises

Exercise 32.1: Modify the function `dir_iter` in the directory example so that it closes the `DIR` structure as soon as it reaches the end of the traversal. With this change, the program does not need to wait for a garbage collection to release a resource that it knows it will not need anymore.

(When you close the directory, you should set the address stored in the userdata to `NULL`, to signal to the finalizer that the directory is already closed. Also, `dir_iter` will have to check whether the directory is closed before using it.)

Exercise 32.2: In the `lxp` example, we used user values to associate the callback table with the userdata that represents a parser. This choice created a small problem, because what the C callbacks receive is the

`lxp_userdata` structure, and that structure does not offer direct access to the table. We solved this problem by storing the callback table at a fixed stack index during the parse of each fragment.

An alternative design would be to associate the callback table with the userdata through references (the section called “The registry”): we create a reference to the callback table and store the reference (an integer) in the `lxp_userdata` structure. Implement this alternative. Do not forget to release the reference when closing the parser.

Chapter 33. Threads and States

Lua does not support true multithreading, that is, preemptive threads sharing memory. There are two reasons for this lack of support. The first reason is that ISO C does not offer it, and so there is no portable way to implement this mechanism in Lua. The second and stronger reason is that we do not think multithreading is a good idea for Lua.

Multithreading was developed for low-level programming. Synchronization mechanisms like semaphores and monitors were proposed in the context of operating systems (and seasoned programmers), not application programs. It is very hard to find and correct bugs related to multithreading, and several of these bugs can lead to security breaches. Moreover, multithreading can lead to performance penalties related to the need of synchronization in some critical parts of a program, such as the memory allocator.

The problems with multithreading arise from the combination of preemption with shared memory, so we can avoid them either using non-preemptive threads or not sharing memory. Lua offers support for both. Lua threads (also known as coroutines) are collaborative, and therefore avoid the problems created by unpredictable thread switching. Lua states share no memory, and therefore form a good base for parallelism in Lua. We will cover both options in this chapter.

Multiple Threads

A *thread* is the essence of a coroutine in Lua. We can think of a coroutine as a thread plus a nice interface, or we can think of a thread as a coroutine with a lower-level API.

From the C API perspective, you may find it useful to think of a thread as a stack—which is what a thread actually is, from an implementation point of view. Each stack keeps information about the pending calls of a thread, plus the parameters and local variables of each call. In other words, a stack has all the information that a thread needs to continue running. So, multiple threads mean multiple independent stacks.

Most functions in Lua's C API operate on a specific stack. How does Lua know which stack to use? When calling `lua_pushnumber`, how do we say where to push the number? The secret is that the type `lua_State`, the first argument to these functions, represents not only a Lua state, but also a thread within that state. (Many people argue that this type should be called `lua_Thread`. Maybe they are right.)

Whenever we create a Lua state, Lua automatically creates a main thread within this state and returns a `lua_State` representing this thread. This main thread is never collected. It is released together with the state, when we close the state with `lua_close`. Programs that do not bother with threads run everything in this main thread.

We can create other threads in a state calling `lua_newthread`:

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

This function pushes the new thread on the stack, as a value of type `"thread"`, and returns a `lua_State` pointer representing this new thread. For instance, consider the following statement:

```
L1 = lua_newthread(L);
```

After running it, we will have two threads, `L1` and `L`, both referring internally to the same Lua state. Each thread has its own stack. The new thread `L1` starts with an empty stack; the old thread `L` has a reference to the new thread on top of its stack:

```
printf("%d\n", lua_gettop(L1));          --> 0
printf("%s\n", luaL_typename(L, -1));    --> thread
```

Except for the main thread, threads are subject to garbage collection, like any other Lua object. When we create a new thread, the value pushed on the stack ensures that the thread is not collected. We should never use a thread that is not properly anchored in the state. (The main thread is internally anchored, so we do not have to worry about it.) Any call to the Lua API may collect a non-anchored thread, even a call using this thread. For instance, consider the following fragment:

```
lua_State *L1 = lua_newthread (L);
lua_pop(L, 1);         /* L1 now is garbage for Lua */
lua_pushstring(L1, "hello");
```

The call to `lua_pushstring` may trigger the garbage collector and collect `L1`, crashing the application, despite the fact that `L1` is in use. To avoid this, always keep a reference to the threads you are using, for instance on the stack of an anchored thread, in the registry, or in a Lua variable.

Once we have a new thread, we can use it like the main thread. We can push to and pop elements from its stack, we can use it to call functions, and the like. For instance, the following code does the call `f(5)` in the new thread and then moves the result to the old thread:

```
lua_getglobal(L1, "f");   /* assume a global function 'f' */
lua_pushinteger(L1, 5);
lua_call(L1, 1, 1);
lua_xmove(L1, L, 1);
```

The function `lua_xmove` moves Lua values between two stacks in the same state. A call like `lua_xmove(F, T, n)` pops `n` elements from the stack `F` and pushes them on `T`.

For these uses, however, we do not need a new thread; we could just use the main thread as well. The main point of using multiple threads is to implement coroutines, so that we can suspend their executions and resume them later. For that, we need the function `lua_resume`:

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

To start running a coroutine, we use `lua_resume` as we use `lua_pcall`: we push the function to be called (which is the coroutine body), push its arguments, and call `lua_resume` passing in `narg` the number of arguments. (The `from` parameter is the thread that is doing the call or `NULL`.) The behavior is also much like `lua_pcall`, with three differences. First, `lua_resume` does not have a parameter for the number of wanted results; it always returns all results from the called function. Second, it does not have a parameter for a message handler; an error does not unwind the stack, so we can inspect the stack after the error. Third, if the running function yields, `lua_resume` returns the code `LUA_YIELD` and leaves the thread in a state that can be resumed later.

When `lua_resume` returns `LUA_YIELD`, the visible part of the thread's stack contains only the values passed to `yield`. A call to `lua_gettop` will return the number of yielded values. To move these values to another thread, we can use `lua_xmove`.

To resume a suspended thread, we call `lua_resume` again. In such calls, Lua assumes that all values on the stack are to be returned by the call to `yield`. For instance, if we do not touch the thread's stack between a return from `lua_resume` and the next resume, `yield` will return exactly the values it yielded.

Typically, we start a coroutine with a Lua function as its body. This Lua function can call other functions, and any of these functions can occasionally yield, terminating the call to `lua_resume`. For instance, assume the following definitions:

```
function foo (x)  coroutine.yield(10, x)  end

function foo1 (x)  foo(x + 1); return 3  end
```

Now, we run this C code:

```
lua_State *L1 = lua_newthread(L);
lua_getglobal(L1, "foo1");
lua_pushinteger(L1, 20);
lua_resume(L1, L, 1);
```

The call to `lua_resume` will return `LUA_YIELD`, to signal that the thread yielded. At this point, the `L1` stack has the values given to `yield`:

```
printf("%d\n", lua_gettop(L1));             --> 2
printf("%lld\n", lua_tointeger(L1, 1));     --> 10
printf("%lld\n", lua_tointeger(L1, 2));     --> 21
```

When we resume the thread again, it continues from where it stopped (the call to `yield`). From there, `foo` returns to `foo1`, which in turn returns to `lua_resume`:

```
lua_resume(L1, L, 0);
printf("%d\n", lua_gettop(L1));             --> 1
printf("%lld\n", lua_tointeger(L1, 1));     --> 3
```

This second call to `lua_resume` will return `LUA_OK`, which means a normal return.

A coroutine can also call C functions, which can call back other Lua functions. We have already discussed how to use continuations to allow those Lua functions to yield (the section called “Continuations”). A C function can yield, too. In that case, it also must provide a continuation function to be called when the thread resumes. To yield, a C function must call the following function:

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

We should use this function always in a return statement, such as here:

```
static inf myCfunction (lua_State *L) {
  ...
  return lua_yieldk(L, nresults, ctx, k);
}
```

This call immediately suspends the running coroutine. The `nresults` parameter is the number of values on the stack to be returned to the respective `lua_resume`; `ctx` is the context information to be passed to the continuation; and `k` is the continuation function. When the coroutine resumes, the control goes directly to the continuation function `k`. After yielding, `myCfunction` cannot do anything else; it must delegate any further work to its continuation.

Let us see a typical example. Suppose we want to write a function that reads some data, yielding if the data is not available. We may write the function in C like this:[1]

```
int readK (lua_State *L, int status, lua_KContext ctx) {
  (void)status;  (void)ctx;  /* unused parameters */
  if (something_to_read()) {
    lua_pushstring(L, read_some_data());
    return 1;
  }
  else
    return lua_yieldk(L, 0, 0, &readK);
}
```

[1] As I already mentioned, the API for continuations prior to Lua 5.3 is a little different. In particular, the continuation function has only one parameter, the Lua state.

```
int prim_read (lua_State *L) {
  return readK(L, 0, 0);
}
```

In this example, `prim_read` does not need to do any initialization, so it calls directly the continuation function (`readK`). If there is data to read, `readK` reads and returns this data. Otherwise, it yields. When the thread resumes, it calls the continuation function again, which will try again to read some data.

If a C function has nothing else to do after yielding, it can call `lua_yieldk` without a continuation function or use the macro `lua_yield`:

```
return lua_yield(L, nres);
```

After this call, when the thread resumes, control returns to the function that called `myCfunction`.

Lua States

Each call to `luaL_newstate` (or to `lua_newstate`) creates a new Lua state. Different Lua states are completely independent of each other. They share no data at all. This means that no matter what happens inside a Lua state, it cannot corrupt another Lua state. This also means that Lua states cannot communicate directly; we have to use some intervening C code. For instance, given two states `L1` and `L2`, the following command pushes in `L2` the string on the top of the stack in `L1`:

```
lua_pushstring(L2, lua_tostring(L1, -1));
```

Because data must pass through C, Lua states can exchange only types that are representable in C, like strings and numbers. Other types, such as tables, must be serialized to be transferred.

In systems that offer multithreading, an interesting design is to create an independent Lua state for each thread. This design results in threads similar to POSIX processes, where we have concurrency without shared memory. In this section, we will develop a prototype implementation for multithreading following this approach. I will use POSIX threads (`pthreads`) for this implementation. It should not be difficult to port the code to other thread systems, as it uses only basic facilities.

The system we are going to develop is very simple. Its main purpose is to show the use of multiple Lua states in a multithreading context. After we have it up and running, we can add several advanced features on top of it. We will call our library `lproc`. It offers only four functions:

`lproc.start(chunk)`	starts a new process to run the given chunk (a string). The library implements a Lua *process* as a C *thread* plus its associated Lua state.
`lproc.send(channel, val1, val2, ...)`	sends all given values (which should be strings) to the given channel identified by its name, also a string. (The exercises will ask you to add support for sending other types.)
`lproc.receive(channel)`	receives the values sent to the given channel.
`lproc.exit()`	finishes a process. Only the main process needs this function. If this process ends without calling `lproc.exit`, the whole program terminates, without waiting for the end of the other processes.

The library identifies channels by strings and uses them to match senders and receivers. A send operation can send any number of string values, which are returned by the matching receive operation. All communication is synchronous: a process sending a message to a channel blocks until there is a process receiving from this channel, while a process receiving from a channel blocks until there is a process sending to it.

Like its interface, the implementation of `lproc` is also simple. It uses two circular double-linked lists, one for processes waiting to send a message and another for processes waiting to receive a message. It uses a single mutex to control access to these lists. Each process has an associated condition variable. When a process wants to send a message to a channel, it traverses the receiving list looking for a process waiting for that channel. If it finds one, it removes the process from the waiting list, moves the message's values from itself to the found process, and signals the other process. Otherwise, it inserts itself into the sending list and waits on its condition variable. To receive a message, it does a symmetrical operation.

A main element in the implementation is the structure that represents a process:

```
#include <pthread.h>
#include "lua.h"
#include "lauxlib.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 the process and the C thread that runs the process. The third field, `cond`, is the condition variable that the thread uses to block itself when it has to wait for a matching send/receive. The fourth field stores the channel that the process is waiting, if any. The last two fields, `previous` and `next`, are used to link the process structure into a waiting list.

The following code declares the two waiting lists and the associated mutex:

```
static Proc *waitsend = NULL;
static Proc *waitreceive = NULL;

static pthread_mutex_t kernel_access = PTHREAD_MUTEX_INITIALIZER;
```

Each process needs a `Proc` structure, and it needs to access this structure whenever its script calls `send` or `receive`. The only parameter that these functions receive is the process's Lua state; therefore, each process should store its `Proc` structure inside its Lua state. In our implementation, each state keeps its corresponding `Proc` structure as a full userdata in the registry, associated with the key `"_SELF"`. The auxiliary function `getself` retrieves the `Proc` structure associated with a 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`, moves values from a sender process to a receiver process:

```
static void movevalues (lua_State *send, lua_State *rec) {
  int n = lua_gettop(send);
  int i;
  luaL_checkstack(rec, n, "too many results");
  for (i = 2; i <= n; i++)  /* move values to receiver */
    lua_pushstring(rec, lua_tostring(send, i));
}
```

It moves to the receiver all values in the sender stack but the first, which will be the channel. Note that, as we are pushing an arbitrary number of elements, we have to check for stack space.

Figure 33.1, “Function to search for a process waiting for a channel” defines the function `searchmatch`, which traverses a list looking for a process that is waiting for a given channel.

Figure 33.1. Function to search for a process waiting for a channel

```
static Proc *searchmatch (const char *channel, Proc **list) {
  Proc *node;
  /* traverse the list */
  for (node = *list; node != NULL; node = node->next) {
    if (strcmp(channel, node->channel) == 0) {  /* match? */
      /* remove node from the list */
      if (*list == node)  /* is this node the first element? */
        *list = (node->next == node) ? NULL : node->next;
      node->previous->next = node->next;
      node->next->previous = node->previous;
      return node;
    }
  }
  return NULL;  /* no match found */
}
```

If it finds one, it removes the process from the list and returns it; otherwise, the function returns `NULL`.

The last auxiliary function, in Figure 33.2, “Function to add a process to a waiting list”, is called when a process cannot find a match.

Figure 33.2. Function to add a process to a waiting list

```
static void waitonlist (lua_State *L, const char *channel,
                                      Proc **list) {
  Proc *p = getself(L);

  /* link itself at the end of the list */
  if (*list == NULL) {  /* empty list? */
    *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;  /* waiting channel */

  do {  /* wait on its condition variable */
    pthread_cond_wait(&p->cond, &kernel_access);
  } while (p->channel);
}
```

In this case, the process links itself at the end of the appropriate waiting list and waits until another process matches with it and wakes it up. (The loop around `pthread_cond_wait` handles the spurious wakeups

allowed in POSIX threads.) When a process wakes up another, it sets the other process's field `channel` to `NULL`. So, if `p->channel` is not `NULL`, it means that nobody matched process `p`, so it has to keep waiting.

With these auxiliary functions in place, we can write `send` and `receive` (Figure 33.3, “Functions to send and receive messages”).

Figure 33.3. Functions to send and receive 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 receiver? */
    movevalues(L, p->L);  /* move values to receiver */
    p->channel = NULL;  /* mark receiver as not waiting */
    pthread_cond_signal(&p->cond);  /* wake it 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 sender? */
    movevalues(p->L, L);  /* get values from sender */
    p->channel = NULL;  /* mark sender as not waiting */
    pthread_cond_signal(&p->cond);  /* wake it up */
  }
  else
    waitonlist(L, channel, &waitreceive);

  pthread_mutex_unlock(&kernel_access);

  /* return all stack values except the channel */
  return lua_gettop(L) - 1;
}
```

The function `ll_send` starts getting the channel. Then it locks the mutex and searches for a matching receiver. If it finds one, it moves its values to this receiver, marks the receiver as ready, and wakes it up.

Otherwise, it puts itself on wait. When it finishes the operation, it unlocks the mutex and returns with no values to Lua. The function `ll_receive` is similar, but it has to return all received values.

Now let us see how to create new processes. A new process needs a new POSIX thread, and a new thread needs a body to run. We will define this body later; here is its prototype, dictated by `pthreads`:

```
static void *ll_thread (void *arg);
```

To create and run a new process, the system must create a new Lua state, start a new thread, compile the given chunk, call the chunk, and finally free its resources. The original thread does the first three tasks, and the new thread does the rest. (To simplify error handling, the system only starts the new thread after it has successfully compiled the given chunk.)

The function `ll_start` creates a new process (Figure 33.4, “Function to create new processes”).

Figure 33.4. Function to create new processes

```
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 in thread body: %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;
}
```

This function creates a new Lua state `L1` and compiles the given chunk in this new state. In case of error, it signals the error to the original state `L`. Then it creates a new thread (using `pthread_create`) with body `ll_thread`, passing the new state `L1` as the argument to the body. The call to `pthread_detach` tells the system that we will not want any final answer from this thread.

The body of each new thread is the function `ll_thread` (Figure 33.5, “Body for new threads”), which receives its corresponding Lua state (created by `ll_start`) with only the precompiled main chunk on the stack.

Figure 33.5. Body for new threads

```
int luaopen_lproc (lua_State *L);

static void *ll_thread (void *arg) {
  lua_State *L = (lua_State *)arg;
  Proc *self;  /* own control block */

  openlibs(L);  /* open standard libraries */
  luaL_requiref(L, "lproc", luaopen_lproc, 1);
  lua_pop(L, 1);  /* remove result from previous call */

  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);

  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;
}
```

First, it opens the standard Lua libraries and the `lproc` library. Second, it creates and initializes its own control block. Then, it calls its main chunk. Finally, it destroys its condition variable and closes its Lua state.

Note the use of `luaL_requiref` to open the `lproc` library.[2] This function is somewhat equivalent to `require` but, instead of searching for a loader, it uses the given function (`luaopen_lproc`, in our case) to open the library. After calling the open function, `luaL_requiref` registers the result in the `package.loaded` table, so that future calls to require the library will not try to open it again. With true as its last parameter, it also registers the library in the corresponding global variable (`lproc`, in our case).

Figure 33.6, “Extra functions for the `lproc` module” presents the last functions for the module.

[2]This function was introduced in Lua 5.2.

Figure 33.6. Extra functions for the `lproc` module

```
static int ll_exit (lua_State *L) {
  pthread_exit(NULL);
  return 0;
}

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) {
  luaL_newlib(L, ll_funcs);  /* open library */
  return 1;
}
```

Both are quite simple. The function `ll_exit` should be called only by the main process, when it finishes, to avoid the immediate end of the whole program. The function `luaopen_lproc` is a standard function for opening the module.

As I said earlier, this implementation of processes in Lua is a very simple one. There are endless improvements we can make. Here I will briefly discuss some of them.

A first obvious improvement is to change the linear search for a matching channel. A nice alternative is to use a hash table to find a channel and to use independent waiting lists for each channel.

Another improvement relates to the efficiency of process creation. The creation of new Lua states is a lightweight operation. However, the opening of all standard libraries is not that lightweight, and most processes probably will not need all standard libraries. We can avoid the cost of opening a library by using the pre-registration of libraries, as we discussed in the section called “The Function `require`”. With this approach, instead of calling `luaL_requiref` for each standard library, we just put the library opening function into the `package.preload` table. If the process calls `require "lib"`, then—and only then—`require` will call the associated function to open the library. The function `registerlib`, in Figure 33.7, “Registering libraries to be opened on demand”, does this registration.

Figure 33.7. Registering libraries to be opened on demand

```
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' tables */
}

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 results from previous calls */
  registerlib(L, "coroutine", luaopen_coroutine);
  registerlib(L, "table", luaopen_table);
  registerlib(L, "io", luaopen_io);
  registerlib(L, "os", luaopen_os);
  registerlib(L, "string", luaopen_string);
  registerlib(L, "math", luaopen_math);
  registerlib(L, "utf8", luaopen_utf8);
  registerlib(L, "debug", luaopen_debug);
}
```

It is always a good idea to open the basic library. We also need the package library; otherwise, we will not have `require` available to open the other libraries. All other libraries can be optional. Therefore, instead of calling `luaL_openlibs`, we can call our own function `openlibs` (shown also in Figure 33.7, “Registering libraries to be opened on demand”) when opening new states. Whenever a process needs one of these libraries, it requires the library explicitly, and `require` will call the corresponding `luaopen_*` function.

Other improvements involve the communication primitives. For instance, it would be useful to provide limits on how long `lproc.send` and `lproc.receive` should wait for a match. In particular, a zero limit would make these functions non-blocking. With POSIX threads, we could use `pthread_cond_timedwait` to implement this feature.

Exercises

Exercise 33.1: As we saw, if a function calls `lua_yield` (the version with no continuation), control returns to the function that called it when the thread resumes. What values does the calling function receive as results from that call?

Exercise 33.2: Modify the `lproc` library so that it can send and receive other primitive types such as Booleans and numbers without converting them to strings. (Hint: you only have to modify the function `movevalues`.)

Exercise 33.3: Modify the `lproc` library so that it can send and receive tables. (Hint: you can traverse the original table building a copy in the receiving state.)

Exercise 33.4: Implement in the `lproc` library a non-blocking `send` operation.