Programming in Lua (first edition)

This is an online version of the first edition of the book

Programming in Lua
by Roberto Ierusalimschy
Lua.org, December 2003
ISBN 85-903798-1-7

The book is a detailed and authoritative introduction to all aspects of Lua programming, by Lua's chief architect. The first edition was aimed at Lua 5.0 and remains largely relevant.

If you find this online version useful, please consider buying a copy of the second edition, which updates the text to Lua 5.1 and brings substantial new material. This helps to support the Lua project.

For the official definition of the Lua language, see the reference manual.

Contents

1 - Getting Started

To keep with the 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`) with the name of the text file that contains your program. For instance, if you write the above program in a file `hello.lua`, the following command should run it:

```
prompt> lua hello.lua
```

As a slightly more complex example, the following 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("*number")        -- read a number
print(fact(a))
```

If you are using Lua embedded in an application, such as CGILua or IUPLua, you may need to refer to the application manual (or to a "local guru") to learn how to run your programs. Nevertheless, Lua is still the same language; most things that we will see here are valid regardless of how you are using Lua. For a start, we recommend that you use the stand-alone interpreter (that is, the `lua` executable) to run your first examples and experiments.

1 - Getting Started

To keep with the 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`) with the name of the text file that contains your program. For instance, if you write the above program in a file `hello.lua`, the following command should run it:

```
prompt> lua hello.lua
```

As a slightly more complex example, the following 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("*number")        -- read a number
print(fact(a))
```

If you are using Lua embedded in an application, such as CGILua or IUPLua, you may need to refer to the application manual (or to a "local guru") to learn how to run your programs. Nevertheless, Lua is still the same language; most things that we will see here are valid regardless of how you are using Lua. For a start, we recommend that you use the stand-alone interpreter (that is, the `lua` executable) to run your first examples and experiments.

1.1 - Chunks

Each piece of code that Lua executes, such as a file or a single line in interactive mode, is a *chunk*. More specifically, a chunk is simply a sequence of statements.

A semicolon may optionally follow any statement. Usually, I use semicolons only to separate two or more statements written in the same line, but this is just a convention. Line breaks play no role in Lua's syntax; for instance, the following four chunks are all valid and equivalent:

```
    a = 1
    b = a*2

    a = 1;
    b = a*2;

    a = 1 ; b = a*2

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

A chunk may be as simple as a single statement, such as in the "hello world" example, or it may be composed of a mix of statements and function definitions (which are assignments actually, as we will see later), such as the factorial example. A chunk may be as large as you 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 sizes.

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

```
    Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
    >
```

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

In interactive mode, Lua usually interprets each line that you type as a complete chunk. However, if it detects that the line cannot form a complete chunk, it waits for more input, until it has a complete chunk. When Lua is waiting for a line continuation, it shows a different prompt (typically >>). Therefore, you can enter a multi-line definition, such as the `factorial` function, directly in interactive mode. Sometimes, however, it is more convenient to put such definitions in a file, and then call Lua to run that file.

You can execute a sequence of chunks by giving them all as arguments to the stand-alone interpreter, with the `-l` option. For instance, if you have a file `a` with a single statement `x=1` and another file `b` with the statement `print(x)`, the command line

```
    prompt> lua -la -lb
```

will run the chunk in `a`, then the one in `b`, which will print the expected 1. (The `-l` option actually calls `require`, which looks for the files in a specific path. So, the previous example will not work if this path does not include the current directory. We will discuss the `require` function in more details in Section 8.1.)

You may use the `-i` option to instruct Lua to start an interactive session after running the given chunks. A command line like

```
    prompt> lua -i -la -lb
```

will run the chunk in `a`, then the one in `b`, and then prompt you 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 link chunks is with the `dofile` function, which immediately executes a file. For instance, you may have a file `lib1.lua`:

```
-- file 'lib1.lua'

function norm (x, y)
  local n2 = x^2 + y^2
  return math.sqrt(n2)
end

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

Then, in interactive mode, you can type

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

The `dofile` function is useful also when you are testing a piece of code. You can work with two windows: One of them is a text editor with your program (in a file `prog.lua`, say) and the other is a console running Lua in interactive mode. After saving a modification that you make to your program, you execute `dofile("prog.lua")` in the Lua console to load the new code; then you can exercise the new code, calling its functions and printing the results.

1.2 - Global Variables

Global variables do not need declarations. You simply assign a value to a global variable to create it. It is not an error to access a non-initialized variable; you just get the special value **nil** as the result:

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

Usually you do not need to delete global variables; if your variable is going to have a short life, you should use a local variable. But, if you need to delete a global variable, just assign **nil** to it:

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

After that, it is as if the variable had never been used. In other words, a global variable is *existent* if (and only if) it has a non-nil value.

1.3 - Some Lexical Conventions

Identifiers 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 uppercase letters (e.g., `_VERSION`); they are reserved for special uses in Lua. Usually, I reserve the identifier _ (a single underscore) for a dummy variable.

In Lua, the concept of what is a letter is locale dependent. Therefore, with a proper locale, you can use variable names such as `índice` or `ação`. However, such names will make your program unsuitable to run in systems that do not support that locale.

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

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

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

A comment starts anywhere with a double hyphen (`--`) and runs until the end of the line. Lua also offers block comments, which start with `--[[` and run until the corresponding `]]`. A common trick, when we want to comment out a piece of code, is to write the following:

```
--[[
print(10)          -- no action (comment)
--]]
```

Now, if we add a single hyphen to the first line, the code is in again:

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

In the first example, the `--` in the last line is still inside the block comment. In the second example, the sequence `---[[` does not start a block comment; so, the `print` is outside comments. In this case, the last line becomes an independent comment, as it starts with `--`.

1.4 - 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 that line starts with a number sign (`` `#´ ``). That feature allows the use of Lua as a script interpreter in Unix systems. If you start your program 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 you can call the program 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 in interactive mode.

The `-e` option allows us to enter code directly into the command line. For instance,

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

(Unix needs the double quotes to stop the shell from interpreting the parentheses.) As we previously saw, `-l` loads a file and `-i` enters interactive mode after running the other arguments. So, for instance, the call

```
prompt> lua -i -l a.lua -e "x = 10"
```

will load the file `a.lua`, then execute the assignment `x = 10`, and finally present a prompt for interaction.

Whenever the global variable `_PROMPT` is defined, `lua` uses its value as the prompt when interacting. So, you can change the prompt with a call like this:

```
prompt> lua -i -e "_PROMPT=' lua> '"
 lua>
```

We are assuming that `"prompt"` is the system's prompt. In the example, the outer quotes stop the shell from interpreting the inner quotes, which are interpreted by Lua. More exactly, Lua receives the following command to run:

```
_PROMPT=' lua> '
```

which assigns the string `" lua> "` to the global variable `_PROMPT`.

Before it starts running arguments, `lua` looks for an environment variable called `LUA_INIT`. If there is such a variable and its content is `@`*filename*, then `lua` loads the given file. If `LUA_INIT` is defined but does not start with `` `@´ ``, then `lua` assumes that it contains Lua code and runs it. This variable gives you great power when configuring the stand-alone interpreter, because you have the full power of Lua in the configuration. You can pre-load packages, change the prompt and the path, define your own functions, rename or delete functions, and so on.

A main script can retrieve its arguments in the global variable `arg`. In a call like

```
prompt> lua script a b c
```

`lua` creates the table `arg` with all the command-line arguments, before running the script. The script name goes into index 0; its first argument (`a` in the example), goes to index 1, and so on. Eventual options go to negative indices, as they appear before the script. For instance, in the call

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

`lua` 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, the script only uses the positive indices (`arg[1]` and `arg[2]`, in the example).

2 - 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 `type` function gives the type name of a given value:

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

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

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

```
print(type(a))    --> nil   (`a' is not initialized)
a = 10
print(type(a))    --> number
a = "a string!!"
print(type(a))    --> string
a = print         -- yes, this is valid!
a(type(a))        --> function
```

Notice the last two lines: Functions are first-class values in Lua; so, we can manipulate them like any other value. (More about that in Chapter 6.)

Usually, when you 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 exceptional condition.

2.1 - Nil

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

2.2 - Booleans

The boolean type has two values, **false** and **true**, which represent the traditional boolean values. However, they do not hold a monopoly of condition values: In Lua, any value may represent a condition. Conditionals (such as the ones in control structures) consider **false** and **nil** as false and anything else as true. Beware that, unlike some other scripting languages, Lua considers both zero and the empty string as true in conditional tests.

2.3 - Numbers

The number type represents real (double-precision floating-point) numbers. Lua has no integer type, as it does not need it. There is a widespread misconception about floating-point arithmetic errors and some people fear that even a simple increment can go weird with floating-point numbers. The fact is that, when you use a double to represent an integer, there is no rounding error at all (unless the number is greater than 100,000,000,000,000). Specifically, a Lua number can represent any long integer without rounding problems. Moreover, most modern CPUs do floating-point arithmetic as fast as (or even faster than) integer arithmetic.

It is easy to compile Lua so that it uses another type for numbers, such as longs or single-precision floats. This is particularly useful for platforms without hardware support for floating point. See the distribution for detailed instructions.

We can write numeric constants with an optional decimal part, plus an optional decimal exponent. Examples of valid numeric constants are:

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

2.4 - Strings

Strings have the usual meaning: a sequence of characters. Lua is eight-bit clean and so strings may contain characters with any numeric value, including embedded zeros. That means that you can store any binary data into a string. Strings in Lua are immutable values. You cannot change a character inside a string, as you may in C; instead, you 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 Lua objects. That means that you do not have to worry about allocation and deallocation of strings; Lua handles this for you. A string may contain a single letter or an entire book. Lua handles long strings quite efficiently. Programs that manipulate strings with 100K or 1M characters are not unusual in Lua.

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

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

As a matter of style, you should use always the same kind of quotes (single or double) in a program, unless the string itself has quotes; then you use the other quote, or escape those quotes with backslashes. 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
`\[`	left square bracket
`\]`	right square bracket

We illustrate their use in the following examples:

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

We can specify a character in a string also by its numeric value through the escape sequence `\ddd`, where `ddd` is a sequence of up to three *decimal* digits. As a somewhat complex example, the two literals `"alo\n123\""` and `'\97lo\10\04923"'` have the same value, in a system using ASCII: 97 is the ASCII code for `a`, 10 is the code for newline, and 49 (`\049` in the example) is the code for the digit `1`.

We can delimit literal strings also by matching double square brackets `[[...]]`. Literals in this bracketed form may run for several lines, may nest, and do not interpret escape sequences. Moreover, this form ignores the first character of the string when this character is a newline. This form is especially convenient for writing strings that contain program pieces; for instance,

```
page = [[
<HTML>
<HEAD>
<TITLE>An HTML Page</TITLE>
</HEAD>
<BODY>
 <A HREF="http://www.lua.org">Lua</A>
 [[a text between double brackets]]
</BODY>
</HTML>
]]

write(page)
```

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:

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

Lua applies such coercions not only in arithmetic operators, but also in other places that expect a number. Conversely, whenever it finds a number where it expects a string, Lua converts the number to a string:

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

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

Despite those automatic conversions, strings and numbers are different things. A comparison like `10 == "10"` is always false, because 10 is a number and `"10"` is a string. If you need to convert a string to a number explicitly, you can use the function `tonumber`, which returns **nil** if the string does not denote a proper number:

```
line = io.read()      -- read a line
n = tonumber(line)    -- try to convert it to a number
if n == nil then
  error(line .. " is not a valid number")
```

```
    else
      print(n*2)
    end
```

To convert a number to a string, you can call the function `tostring` or concatenate the number with the empty string:

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

Such conversions are always valid.

2.5 - Tables

The table type implements associative arrays. An associative array is an array that can be indexed not only with numbers, but also with strings or any other value of the language, except **nil**. Moreover, tables have no fixed size; you can add as many elements as you want to a table dynamically. Tables are the main (in fact, the only) data structuring mechanism in Lua, and a powerful one. We use tables to represent ordinary arrays, symbol tables, sets, records, queues, and other data structures, in a simple, uniform, and efficient way. Lua uses tables to represent packages as well. When we write `io.read`, we mean "the `read` entry from the `io` package". For Lua, that means "index the table `io` using the string `"read"` as the key".

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 we mean. However, if your idea of an array comes from C or Pascal, you have to open your mind a bit. You may think of a table as a dynamically allocated object; your program only manipulates references (or pointers) to them. There are no hidden copies or creation of new tables behind the scenes. Moreover, you do not have to declare a table in Lua; in fact, there is no way to declare one. You create tables by means of a *constructor expression*, which in its simplest form is written as `{}`:

```
    a = {}     -- create a table and store its reference in `a'
    k = "x"
    a[k] = 10        -- new entry, with key="x" and value=10
    a[20] = "great"  -- new entry, with key=20 and value="great"
    print(a["x"])    --> 10
    k = 20
    print(a[k])      --> "great"
    a["x"] = a["x"] + 1     -- increments entry "x"
    print(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'
    print(b["x"])  --> 10
    b["x"] = 20
    print(a["x"])  --> 20
    a = nil    -- now only `b' still refers to the table
    b = nil    -- now there are no references left to the table
```

When a program has no references to a table left, Lua memory management will eventually delete the table and reuse its memory.

Each table may store values with different types of indices and it grows as it needs to accommodate new entries:

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

Notice the last line: Like global variables, table fields evaluate to **nil** if they are not initialized. Also like global variables, you can assign **nil** to a table field to delete it. That is not a coincidence: Lua stores global variables in ordinary tables. More about this subject in Chapter 14.

To represent records, you use the field name as an index. Lua supports this representation by providing `a.name` as syntactic sugar for `a["name"]`. So, we could write the last lines of the previous example in a cleanlier manner as

```
a.x = 10                    -- same as a["x"] = 10
print(a.x)                  -- same as print(a["x"])
print(a.y)                  -- same as print(a["y"])
```

For Lua, the two forms are equivalent and can be intermixed freely; but for a human reader, each form may signal a different intention.

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"
print(a[x])   --> 10       -- value of field "y"
print(a.x)    --> nil      -- value of field "x" (undefined)
print(a.y)    --> 10       -- value of field "y"
```

To represent a conventional array, you simply use a table with integer keys. There is no way to declare its size; you just initialize the elements you need:

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

When you iterate over the elements of the array, the first non-initialized index will result in **nil**; you can use this value as a sentinel to represent the end of the array. For instance, you could print the lines read in the last example with the following code:

```
-- print the lines
for i,line in ipairs(a) do
  print(line)
end
```

The basic Lua library provides `ipairs`, a handy function that allows you to iterate over the elements of an array, following the convention that the array ends at its first nil element.

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

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 (according to equality) and therefore denote different positions in a table. By the same token, the strings `"+1"`, `"01"`, and `"1"` all denote different positions. 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] = "one value"
a[j] = "another value"
a[k] = "yet another value"
print(a[j])            --> another value
print(a[k])            --> yet another value
print(a[tonumber(j)])  --> one value
print(a[tonumber(k)])  --> one value
```

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

2.6 - Functions

Functions are first-class values in Lua. That means that functions can be stored in variables, passed as arguments to other functions, and returned as results. Such facilities give great flexibility to the language: A program may redefine a function to add new functionality, or simply erase a function to create a secure environment when running a piece of untrusted code (such as code received through a network). Moreover, Lua offers good support for functional programming, including nested functions with proper lexical scoping; just wait. Finally, first-class functions play a key role in Lua's object-oriented facilities, as we will see in Chapter 16.

Lua can call functions written in Lua and functions written in C. All the standard library in Lua is written in C. It comprises functions for string manipulation, table manipulation, I/O, access to basic operating system facilities, mathematical functions, and debugging. Application programs may define other functions in C.

2.7 - Userdata and Threads

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 files. We will discuss more about userdata later, when we get to the C API.

We will explain the thread type in Chapter 9, where we discuss coroutines.

3 - Expressions

Expressions denote values. Expressions in Lua include the numeric constants and string literals, variables, unary and binary operations, and function calls. Expressions can be also the unconventional function definitions and table constructors.

3.1 - Arithmetic Operators

Lua supports the usual arithmetic operators: the binary `+´ (addition), `-´ (subtraction), `*´ (multiplication), `/´ (division), and the unary `-´ (negation). All of them operate on real numbers.

Lua also offers partial support for `^´ (exponentiation). One of the design goals of Lua is to have a tiny core. An exponentiation operation (implemented through the pow function in C) would mean that we should always need to link Lua with the C mathematical library. To avoid this need, the core of Lua offers only the syntax for the `^´ binary operator, which has the higher precedence among all operations. The mathematical library (which is standard, but not part of the Lua core) gives to this operator its expected meaning.

3.2 - Relational Operators

Lua provides the following relational operators:

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

All these operators always result in **true** or **false**.

The operator == tests for equality; the operator ~= is the negation of equality. We can apply both operators to any two values. If the values have different types, Lua considers them different values. Otherwise, Lua compares them according to their types. Specifically, **nil** is equal only to itself.

Lua compares tables, userdata, and functions by reference, that is, two such values are considered equal only if they are the very same object. For instance, after the code

```
    a = {}; a.x = 1; a.y = 0
    b = {}; b.x = 1; b.y = 0
    c = a
```

you have that `a==c` but `a~=b`.

We can apply the order operators only to two numbers or to two strings. Lua compares numbers in the usual way. Lua compares strings in alphabetical order, which follows the locale set for Lua. For instance, with the European Latin-1 locale, we have `"acai" < "açaí" < "acorde"`. Other types can be compared only for equality (and inequality).

When comparing values with different types, you must be careful: Remember that `"0"==0` is false. Moreover, `2<15` is obviously true, but `"2"<"15"` is false (alphabetical order!). To avoid inconsistent results, Lua raises an error when you mix strings and numbers in an order comparison, such as `2<"15"`.

3.3 - Logical Operators

The logical operators are **and**, **or**, and **not**. Like control structures, all logical operators consider **false** and **nil** as false and anything else as true. The operator **and** returns its first argument if it is false; otherwise, it returns its second argument. The operator **or** returns its first argument if it is not false; otherwise, it returns its second argument:

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

Both **and** and **or** use short-cut evaluation, that is, they evaluate their second operand only when necessary.

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

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

i.e., 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`, because **and** has a higher precedence than **or**), which 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 a statement like

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

When `x > y`, the first expression of the **and** is true, so the **and** results in its second expression (`x`) (which is also true, because it is a number), and then the **or** expression results in the value of its first expression, `x`. When `x > y` is false, the **and** expression is false and so the **or** results in its second expression, `y`.

The operator **not** always returns **true** or **false**:

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

3.4 - Concatenation

Lua denotes the string concatenation operator by "`..`" (two dots). If any of its operands is a number, Lua converts that number to a string.

```
    print("Hello " .. "World")  --> Hello World
    print(0 .. 1)               --> 01
```

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

```
    a = "Hello"
    print(a .. " World")   --> Hello World
    print(a)               --> Hello
```

3.5 - Precedence

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

```
             ^
             not  - (unary)
             *   /
             +   -
```

```
          ..
          <   >   <=  >=  ~=  ==
          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 up in the manual and probably you will have the same doubt when you read the code again.

3.6 - 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, `{}`, which creates an empty table; we saw it before. Constructors also initialize arrays (called also *sequences* or *lists*). For instance, the statement

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

will initialize `days[1]` with the string `"Sunday"` (the first element has always index 1, not 0), `days[2]` with `"Monday"`, and so on:

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

Constructors do not need to use only constant expressions. We can use any kind of expression for the value of each element. For instance, we can build a short sine table as

```
    tab = {sin(1), sin(2), sin(3), sin(4),
           sin(5), sin(6), sin(7), sin(8)}
```

To initialize a table to be used as a record, Lua offers the following syntax:

```
    a = {x=0, y=0}
```

which is equivalent to

```
    a = {}; a.x=0; a.y=0
```

No matter what constructor we use to create a table, we can always add and remove other fields of any type to it:

```
    w = {x=0, y=0, label="console"}
    x = {sin(0), sin(1), sin(2)}
    w[1] = "another field"
    x.f = w
    print(w["x"])   --> 0
    print(w[1])     --> another field
    print(x.f[1])   --> another field
    w.x = nil       -- remove field "x"
```

That is, *all tables are created equal*; constructors only affect their initialization.

Every time Lua evaluates a constructor, it creates and initializes a new table. Consequently, we can use tables to implement linked lists:

```
list = nil
for line in io.lines() do
  list = {next=list, value=line}
end
```

This code reads lines from the standard input and stores them in a linked list, in reverse order. Each node in the list is a table with two fields: `value`, with the line contents, and `next`, with a reference to the next node. The following code prints the list contents:

```
l = list
while l do
  print(l.value)
  l = l.next
end
```

(Because we implemented our list as a stack, the lines will be printed in reverse order.) Although instructive, we hardly use the above implementation in real Lua programs; lists are better implemented as arrays, as we will see in Chapter 11.

We can mix record-style and list-style initializations in the same constructor:

```
polyline = {color="blue", thickness=2, npoints=4,
             {x=0,   y=0},
             {x=-10, y=0},
             {x=-10, y=1},
             {x=0,   y=1}
           }
```

The above example also illustrates how we can nest constructors to represent more complex data structures. Each of the elements `polyline[1]`, ..., `polyline[4]` is a table representing a record:

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

Those two constructor forms have their limitations. For instance, you cannot initialize fields with negative indices, or with string indices that are not proper identifiers. For such needs, there is another, more general, format. In this format, we explicitly write the index to be initialized 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])          --> ---
```

That syntax is more cumbersome, but more flexible too: Both the list-style and the record-style forms are special cases of this more general one. The constructor

```
{x=0, y=0}
```

is equivalent to

```
{["x"]=0, ["y"]=0}
```

and the constructor

```
{"red", "green", "blue"}
```

is equivalent to

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

For those that really want their arrays starting at 0, it is not difficult to write the following:

```
days = {[0]="Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"}
```

Now, the first value, `"Sunday"`, is at index 0. That zero does not affect the other fields, but `"Monday"` naturally goes to index 1, because it is the first list value in the constructor; the other values follow it. Despite this facility, I do not recommend the use of arrays starting at 0 in Lua. Remember that most functions assume that arrays start at index 1, and therefore will not handle such arrays correctly.

You 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",}
```

Such flexibility makes it easier to write programs that generate Lua tables, because they do not need to handle the last element as a special case.

Finally, you can always use a semicolon instead of a comma in a constructor. We usually reserve semicolons to delimit different sections in a constructor, for instance to separate its list part from its record part:

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

4 - Statements

Lua supports an almost conventional set of statements, similar to those in C or Pascal. The conventional statements include assignment, control structures, and procedure calls. Lua also supports some not so conventional statements, such as multiple assignments and local variable declarations.

4.1 - Assignment

Assignment is the basic means of changing the value of a variable or a table field:

```
a = "hello" .. "world"
t.n = t.n + 1
```

Lua allows *multiple assignment*, where a list of values is assigned to a list of variables in one step. Both lists have their elements separated by commas. For instance, in the assignment

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

the variable `a` gets the value 10 and `b` gets `2*x`.

In a multiple assignment, Lua first evaluates all values and only then executes the assignments.

Therefore, we can use a multiple assignment to swap two values, as in

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

Lua always *adjusts* the number of values to the number of variables: When the list of values is shorter than the list of variables, the extra variables receive **nil** as their values; when the list of values is longer, the extra values are silently discarded:

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

The last assignment in the above example shows a common mistake. To initialize a set of variables, you must provide a value for each one:

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

Actually, most of the previous examples are somewhat artificial. I seldom use multiple assignment simply to write several assignments in one line. But often we really need multiple assignment. We already saw an example, to swap two values. A more frequent use is to collect multiple returns from function calls. As we will discuss in detail later, a function call can return multiple values. In such cases, a single expression can supply the values for several variables. For instance, in the assignment

```
    a, b = f()
```

`f()` returns two results: `a` gets the first and `b` gets the second.

4.2 - Local Variables and Blocks

Besides global variables, Lua supports local variables. We create local variables with the **local** statement:

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

Unlike global variables, local variables have their *scope* limited to the block where they are declared. A block is the body of a control structure, the body of a function, or a chunk (the file or string with the code 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)
    else
```

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

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

Beware that this example will not work as expected if you enter it in interactive mode. The second line, `local i = 1`, is a complete chunk by itself. As soon as you enter this line, Lua runs it and starts a new chunk in the next line. By then, the **local** declaration is already out of scope. To run such examples in interactive mode, you should enclose all the code in a **do** block.

It is good programming style to use local variables whenever possible. Local variables help you avoid cluttering the global environment with unnecessary names. Moreover, the access to local variables is faster than to global ones.

Lua handles local variable declarations as statements. As such, you can write local declarations anywhere you can write a statement. The scope begins after the declaration and goes until the end of the block. The declaration may include an initial assignment, which works the same way as a conventional assignment: Extra values are thrown away; extra variables get **nil**. As a specific case, 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`. That idiom is 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`; it also speeds up access to `foo`.

Because many languages force you to declare all local variables at the beginning of a block (or a procedure), 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 you need it, you seldom need to declare it without an initial value (and therefore you seldom forget to initialize it). Moreover, you shorten the scope of the variable, which increases readability.

We can delimit a block explicitly, bracketing it with the keywords **do**-**end**. These **do** blocks can be useful when you need finer control over the scope of one or more local variables:

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

4.3 - Control Structures

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

The condition expression of a control structure may result in any value. Lua treats as true all values different from **false** and **nil**.

4.3.1 - 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
```

When you write nested **if**s, you 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
```

4.3.2 - while

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
```

4.3.3 - repeat

As the name implies, a **repeat-until** statement repeats its body until its condition is true. The test is done after the body, so the body is always executed at least once.

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

4.3.4 - Numeric for

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

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

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

That 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. As typical examples of such loops, we have

```
for i=1,f(x) do print(i) end

for i=10,1,-1 do print(i) 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. For instance, in the first example, `f(x)` is called only once. Second, the control variable is a local variable automatically declared by the **for** statement and 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! `i' here is global
```

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

```
-- find a value in a list
local found = nil
for i=1,a.n do
  if a[i] == value then
    found = i      -- save value of `i'
    break
  end
end
print(found)
```

Third, you should never change the value of the control variable: The effect of such changes is unpredictable. If you want to break a **for** loop before its normal termination, use **break**.

4.3.5 - Generic for

The generic **for** loop allows you to traverse all values returned by an iterator function. We have already seen examples of the generic **for**:

```
-- print all values of array `a'
for i,v in ipairs(a) do print(v) end
```

For each step in that code, `i` gets an index, while `v` gets the value associated with that index. A similar example shows how we traverse all keys of a table:

```
-- print all keys of table `t'
for k in pairs(t) do print(k) end
```

Despite its apparent simplicity, the generic **for** is powerful. With proper iterators, we can traverse almost anything, and do it in a readable fashion. The standard libraries provide several iterators, which allow us to iterate over the lines of a file (`io.lines`), the pairs in a table (`pairs`), the words of a string (`string.gfind`, which we will see in Chapter 20), and so on. 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. We will cover this topic later, in Chapter 7.

The generic loop shares two properties with the numeric loop: The loop variables are local to the loop body and you should never assign any value to the loop variables.

Let us see a more concrete example of the use of a generic **for**. Suppose you have a table with the names of the days of the week:

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

Now you want to translate a name into its position in the week. You can search the table, looking for the given name. Frequently, however, a more efficient approach in Lua is to build a *reverse table*, say `revDays`, that has the names as indices and the numbers as values. That table would look like this:

```
revDays = {["Sunday"] = 1, ["Monday"] = 2,
           ["Tuesday"] = 3, ["Wednesday"] = 4,
           ["Thursday"] = 5, ["Friday"] = 6,
           ["Saturday"] = 7}
```

Then, all you 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 manually declare the reverse table. We can build it automatically from the original one:

```
revDays = {}
for i,v in ipairs(days) do
  revDays[v] = i
end
```

The loop will do the assignment for each element of `days`, with the variable `i` getting the index (1, 2, ...) and `v` the value (`"Sunday"`, `"Monday"`, ...).

4.4 - break and return

The **break** and **return** statements allow us to jump out from an inner block.

You 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 occasional results from a function or simply finishes a function. There is an implicit return at the end of any function, so you do not need to use one if your function ends naturally, without returning any value.

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

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

Usually, these are the places where we use these statements, because any other statement following them is unreachable. Sometimes, however, it may be useful to write a **return** (or a **break**) in the middle of a block; for instance, if you are debugging a function and want to avoid its execution. In such cases, you 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
  ...             -- statements not reached
end
```

5 - 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 *procedure* or *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, we write a list of arguments enclosed in parentheses. If the function call has no arguments, we must write an empty list `()` to indicate the call. There is a special case to this rule: If the function has one single argument and this 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)` is just another way to write `o.foo(o, x)`, that is, to call `o.foo` adding `o` as a first extra argument. In Chapter 16 we will discuss such calls (and object-oriented programming) in more detail.

Functions used by a Lua program can be defined both in Lua and in C (or in any other language used by the host application). For instance, all library functions are written in C; but this fact has no relevance to Lua programmers. When calling a function, there is no difference between functions defined in Lua and functions defined in C.

As we have seen in other examples, a function definition has a conventional syntax; for instance

```
-- add all elements of array `a'
function add (a)
  local sum = 0
  for i,v in ipairs(a) do
    sum = sum + v
  end
  return sum
end
```

In that syntax, a function definition has a *name* (`add`, in the previous example), a list of *parameters*, and a *body*, which is a list of statements.

Parameters work exactly as local variables, initialized with the actual arguments given in the function call. You 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, as it does in a multiple assignment: Extra arguments are thrown away; extra parameters get **nil**. For instance, if we have a function like

```
function f(a, b) return a or b end
```

we will have the following mapping from arguments to parameters:

```
CALL                 PARAMETERS

f(3)                 a=3, b=nil
f(3, 4)              a=3, b=4
f(3, 4, 5)           a=3, b=4    (5 is discarded)
```

Although this behavior can lead to programming errors (easily spotted at run time), it is also useful, especially for default arguments. For instance, consider the following function, to increment a global counter.

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

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

5.1 - Multiple Results

An unconventional, but quite convenient feature of Lua is that functions may return multiple results. Several predefined functions in Lua return multiple values. An example is the `string.find` function, which locates a pattern in a string. It returns two indices: the index of the character where the pattern match starts and the one where it ends (or **nil** if it cannot find the pattern). A multiple assignment allows the program to get both results:

```
s, e = string.find("hello Lua users", "Lua")

print(s, e)   -->  7      9
```

Functions written 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 an array can return both the maximum value and its location:

```
function maximum (a)
  local mi = 1          -- maximum index
  local m = a[mi]       -- maximum value
  for i,val in ipairs(a) do
    if val > m then
      mi = i
      m = val
    end
  end
  return m, mi
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 of its results. When we use a call as an expression, 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 assignment, arguments to function calls, table constructors, and **return** statements. To illustrate all these uses, 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'
```

If a function has no results, or not as many results as we need, Lua produces **nil**s:

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

A function call that is not the last element in the list always produces one result:

```
x,y = foo2(), 20      -- x='a', y=20
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 have seen examples of this construction already, with `print`:

```
print(foo0())          -->
print(foo1())          -->  a
print(foo2())          -->  a   b
print(foo2(), 1)       -->  a   1
print(foo2() .. "x")   -->  ax         (see below)
```

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

The `print` function may receive a variable number of arguments. (In the next section we will see how to write functions with variable number of arguments.) If we write `f(g())` and `f` has a fixed number of arguments, Lua adjusts the number of results of `g` to the number of parameters of `f`, as we saw previously.

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

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

As always, this behavior happens only when the call is the last in the list; otherwise, any call produces exactly one result:

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

You 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, so any pair of parentheses placed there counts as an extra pair. That is, a statement like `return (f())` always returns one single value, no matter how many values `f` returns. Maybe this is what you want, maybe not.

A special function with multiple returns is `unpack`. It receives an array and returns as results all elements from the array, starting from index 1:

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

An important use for `unpack` is in a *generic call* mechanism. A generic call mechanism allows you to call any function, with any arguments, dynamically. In ANSI C, for instance, there is no way to do that. You can declare a function that receives a variable number of arguments (with `stdarg.h`) and you can call a variable function, using pointers to functions. However, you 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 you want to call a variable function `f` with variable arguments in an array `a`, you simply write

```
f(unpack(a))
```

The call to `unpack` returns all values in `a`, which become the arguments to `f`. For instance, if we execute

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

then the call `f(unpack(a))` returns 3 and 4, exactly the same as the static call `string.find("hello", "ll")`.

Although the predefined `unpack` is written in C, we could write it also in Lua, using recursion:

```
function unpack (t, i)
  i = i or 1
  if t[i] ~= nil then
    return t[i], unpack(t, i + 1)
  end
end
```

The first time we call it, with a single argument, `i` gets 1. Then the function returns `t[1]` followed by all results from `unpack(t, 2)`, which in turn returns `t[2]` followed by all results from `unpack(t, 3)`, and so on, until the last non-nil element.

5.2 - Variable Number of Arguments

Some functions in Lua receive a variable number of arguments. For instance, we have already called `print` with one, two, and more arguments.

Suppose now that we want to redefine `print` in Lua: Perhaps our system does not have a `stdout` and so, instead of printing its arguments, `print` stores them in a global variable, for later use. We can write this new function in Lua as follows:

```
printResult = ""

function print (...)
  for i,v in ipairs(arg) do
    printResult = printResult .. tostring(v) .. "\t"
  end
  printResult = printResult .. "\n"
end
```

The three dots (`...`) in the parameter list indicate that the function has a variable number of arguments. When this function is called, all its arguments are collected in a single table, which the function accesses as a hidden parameter named `arg`. Besides those arguments, the `arg` table has an extra field, `n`, with the actual number of arguments collected.

Sometimes, a function has some fixed parameters plus a variable number of parameters. Let us see an example. When we write a function that returns multiple values into an expression, only its first result is used. However, sometimes we want another result. A typical solution is to use dummy variables; for instance, if we want only the second result from `string.find`, we may write the following code:

```
local _, x = string.find(s, p)
-- now use `x'
...
```

An alternative solution is to define a `select` function, which selects a specific return from a function:

```
print(string.find("hello hello", " hel"))         --> 6  9
print(select(1, string.find("hello hello", " hel"))) --> 6
print(select(2, string.find("hello hello", " hel"))) --> 9
```

Notice that a call to `select` has always one fixed argument, the *selector*, plus a variable number of extra arguments (the returns of a function). To accommodate this fixed argument, a function may have regular parameters before the dots. Then, Lua assigns the first arguments to those parameters and only the extra arguments (if any) go to `arg`. To better illustrate this point, assume a definition like

```
function g (a, b, ...) end
```

Then, we have the following mapping from arguments to parameters:

```
CALL            PARAMETERS

g(3)             a=3, b=nil, arg={n=0}
g(3, 4)          a=3, b=4, arg={n=0}
g(3, 4, 5, 8)    a=3, b=4, arg={5, 8; n=2}
```

Using those regular parameters, the definition of `select` is straightforward:

```
function select (n, ...)
  return arg[n]
```

```
    end
```

Sometimes, a function with a variable number of arguments needs to pass them all to another function. All it has to do is to call the other function using `unpack(arg)` as argument: `unpack` will return all values in `arg`, which will be passed to the other function. A good example of this use is a function to write formatted text. Lua provides separate functions to format text (`string.format`, similar to the `sprintf` function from the C library) and to write text (`io.write`). Of course, it is easy to combine both functions into a single one, except that this new function has to pass a variable number of values to `format`. This is a job for `unpack`:

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

5.3 - Named Arguments

The parameter passing mechanism in Lua is *positional*: When we call a function, arguments match parameters by their positions. The first argument gives the value to the first parameter, and so on. Sometimes, however, it is useful to specify the arguments by name. To illustrate this point, let us consider the function `rename` (from the `os` library), which renames a file. Quite often, we forget which name comes first, the new or the old; therefore, we may want to redefine this function to receive its two arguments by name:

```
    -- invalid code
    rename(old="temp.lua", new="temp1.lua")
```

Lua has no direct support for that syntax, but we can have the same final effect, with a small syntax change. The idea here is to pack all arguments into a table and use that table as the only argument to the function. The special syntax that Lua provides for function calls, with just one table constructor as argument, helps the trick:

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

Accordingly, we define `rename` with only one parameter and get the actual arguments from this parameter:

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

This style of parameter passing is especially helpful when the function has many parameters, and most of them are optional. For instance, a function that creates a new window in a GUI library may have dozens of arguments, most of them optional, which are best specified by names:

```
    w = Window{ x=0, y=0, width=300, height=200,
                title = "Lua", background="blue",
                border = true
              }
```

The `Window` function then has the freedom to check for mandatory arguments, add default values, and the like. Assuming a primitive `_Window` function that actually creates the new window (and that needs all arguments), we could define `Window` as follows:

```
    function Window (options)
      -- check mandatory options
      if type(options.title) ~= "string" then
        error("no title")
      elseif type(options.width) ~= "number" then
        error("no width")
      elseif type(options.height) ~= "number" then
        error("no height")
      end

      -- everything else is optional
      _Window(options.title,
              options.x or 0,    -- default value
              options.y or 0,    -- default value
              options.width, options.height,
              options.background or "white",   -- default
              options.border      -- default is false (nil)
             )
    end
```

6 - More about Functions

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 conventional values like numbers and strings. Functions can be stored in variables (both global and local) and in tables, can be passed as arguments, and can be returned by other functions.

What does it mean for functions to have "lexical scoping"? It means that functions can access variables of its enclosing functions. (It also means that Lua contains the lambda calculus properly.) As we will see in this chapter, this apparently innocuous property brings great power to the language, because it allows 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 those techniques, because they can make your programs smaller and simpler.

A somewhat difficult notion in Lua is that functions, like all other values, are anonymous; they do not have names. When we talk about a function name, say `print`, we are actually talking about a variable that holds that function. Like any other variable holding any other value, we can manipulate such variables in many ways. The following example, although a little silly, shows the point:

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

Later we will see more useful applications for this facility.

If functions are values, are there any expressions that create functions? Yes. In fact, the usual way to write a function in Lua, like

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

is just an instance of what we call *syntactic sugar*; in other words, it is just a pretty way to write

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

That is, a function definition is in fact a statement (an assignment, more specifically) that assigns a value of type "`function`" to a variable. We can see the expression `function (x) ... end` as a function constructor, just as `{}` is a table constructor. We call the result of such function constructors an *anonymous function*. Although we usually assign functions to global names, giving them something like a name, there are several occasions when functions remain anonymous. Let us see some examples.

The table library provides a 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 receives two elements and returns whether the first must come before the second in the sort. For instance, suppose we have a table of records such as

```
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

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

See how handy the anonymous function is in that statement.

A function that gets 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. But remember that higher-order functions have no special rights; they are a simple consequence of the ability of Lua to handle functions as first-class values.

To illustrate the use of functions as arguments, we will write a naive implementation of a common higher-order function, `plot`, that plots a mathematical function. Below we show this implementation, using some escape sequences to draw on an ANSI terminal. (You may need to change these control sequences to adapt this code to your terminal type.)

```
function eraseTerminal ()
  io.write("\27[2J")
end

-- writes an `*' at column `x' , row `y'
function mark (x,y)
  io.write(string.format("\27[%d;%dH*", y, x))
end

-- Terminal size
TermSize = {w = 80, h = 24}

-- plot a function
-- (assume that domain and image are in the range [-1,1])
function plot (f)
  eraseTerminal()
  for i=1,TermSize.w do
     local x = (i/TermSize.w)*2 - 1
     local y = (f(x) + 1)/2 * TermSize.h
     mark(i, y)
  end
  io.read()  -- wait before spoiling the screen
```

```
    end
```

With that definition in place, you can plot the sine function with a call like

```
    plot(function (x) return math.sin(x*2*math.pi) end)
```

(We need to massage the data a little to put values in the proper range.) When we call `plot`, its parameter `f` gets the value of the given anonymous function, which is then called inside the **for** loop repeatedly to provide the values for the plotting.

Because functions are first-class values in Lua, we can store them not only in global variables, but also in local variables and in table fields. As we will see later, the use of functions in table fields is a key ingredient for some advanced uses of Lua, such as packages and object-oriented programming.

6.1 - Closures

When a function is written enclosed in another function, it has full access to local variables from the enclosing function; this feature is called *lexical scoping*. Although that may sound obvious, it is not. Lexical scoping, plus first-class functions, is a powerful concept in a programming language, but few languages support that concept.

Let us start with a simple example. Suppose you have a list of student names and a table that associates names to grades; you want to sort the list of names, according to their grades (higher grades first). You 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 you 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 the example is that the anonymous function given to `sort` accesses the parameter `grades`, which is local to the enclosing function `sortbygrade`. Inside this anonymous function, `grades` is neither a global variable nor a local variable. We call it an *external local variable*, or an *upvalue*. (The term "upvalue" is a little misleading, because `grades` is a variable, not a value. However, this term has historical roots in Lua and it is shorter than "external local variable".)

Why is that so interesting? Because functions are first-class values. Consider the following code:

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

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

```
print(c1())  --> 2
```

Now, the anonymous function uses an upvalue, `i`, to keep its counter. However, by the time we call the anonymous function, `i` is already out of scope, because the function that created that variable (`newCounter`) has returned. Nevertheless, Lua handles that situation correctly, using the concept of *closure*. Simply put, a closure is a function plus all it needs to access its upvalues correctly. If we call `newCounter` again, it will create a new local variable `i`, so we will get a new closure, acting over that new variable:

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

So, `c1` and `c2` are different closures over the same function and each acts upon an independent instantiation of the local variable `i`. Technically speaking, what is a value in Lua is the closure, not the function. The function itself is just a prototype for closures. Nevertheless, we will continue to use the term "function" to refer to a closure whenever there is no possibility of 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; this mechanism allows Lua programs to incorporate fancy programming techniques from the functional world. Closures are useful for *callback* functions, too. The typical example here occurs when you create buttons in a typical GUI toolkit. Each button has a callback function to be called when the user presses the button; you want different buttons to do slightly different things when pressed. For instance, a digital calculator needs ten similar buttons, one for each digit. You can create each of them with a function like the next one:

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

In this example, we assume 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. (It is actually a closure, because it accesses the upvalue `digit`.) The callback function can be called a long time after `digitButton` did its task and after the local variable `digit` went out of scope, but it can still access that 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 Lua is so flexible. Frequently, however, when you redefine a function you need the original function in the new implementation. For instance, suppose you want to redefine the function `sin` to operate in degrees instead of radians. This new function must convert its argument, and then call the original `sin` function to do the real work. Your code could look like

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

A cleaner way to do that is as follows:

```
do
  local oldSin = math.sin
```

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

Now, we keep the old version in a private variable; the only way to access it is through the new version.

You can use this same feature 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 a program can access, we can redefine the `open` function (from the `io` library) using closures:

```
  do
    local oldOpen = io.open
    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 that redefinition, there is no way for the program to call the unrestricted `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 facility, you can build Lua sandboxes in Lua itself, with the usual benefit: flexibility. Instead of a one-size-fits-all solution, Lua offers you a meta-mechanism, so that you can tailor your environment for your specific security needs.

6.2 - 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 the regular syntax for functions and for tables:

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

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 yet another syntax to define such functions:

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

```
function Lib.goo (x,y)
  return x - y
end
```

This last fragment is exactly equivalent to the first example.

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 may declare local functions, which are visible only inside the chunk. Lexical scoping ensures that other functions in the package can use these local functions:

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

local g = function (...)
  ...
  f()   -- external local `f' is visible here
  ...
end
```

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

```
local function f (...)
  ...
end
```

A subtle point arises in the definition of recursive local functions. The naive approach does not work here:

```
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, that expression calls a global `fact`, not the local one. To solve that problem, we must first define the local variable and then define 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. That is the way Lua expands its syntactic sugar for local functions, so you can use it for recursive functions without worrying:

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

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

```
local f, g    -- `forward' declarations

function g ()
  ...  f() ...
end

function f ()
  ...  g() ...
end
```

6.3 - Proper Tail Calls

Another interesting feature of functions in Lua is that they do proper tail calls. (Several authors use the term *proper tail recursion*, although the concept does not involve recursion directly.)

A *tail call* is a kind of 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)
  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 in the stack. 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 those implementations support *proper tail calls*.

Because a proper tail call uses no stack space, there is no limit on the number of "nested" tail calls that a program can make. For instance, we can call the following function with any number as argument; it will never overflow the stack:

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

A subtle point when we use proper tail calls is what is a tail call. Some obvious candidates fail the criteria that the calling function has nothing 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)
  return
end
```

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

```
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 in the format `return g(...)` is a tail call. However, both `g` 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)
```

As I said earlier, a tail call is a kind of goto. As such, a quite useful application of proper tail calls in Lua is for programming state machines. Such applications can represent each state by a function; to change state is to go to (or to call) a specific function. As an 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 that 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 such maze with one function for each room. We use tail calls to move from one room to another. A small maze with four rooms could look like this:

```
function room1 ()
  local move = io.read()
  if move == "south" then return room3()
  elseif move == "east" then return room2()
  else print("invalid move")
       return room1()   -- stay in the same room
  end
end

function room2 ()
  local move = io.read()
  if move == "south" then return room4()
  elseif move == "west" then return room1()
  else print("invalid move")
       return room2()
  end
end

function room3 ()
  local move = io.read()
  if move == "north" then return room1()
  elseif move == "east" then return room4()
  else print("invalid move")
       return room3()
  end
end

function room4 ()
  print("congratulations!")
end
```

We start the game with a call to the initial room:

```
room1()
```

Without proper tail calls, each user move would create a new stack level. After some number of moves, there would be a stack overflow. With proper tail calls, there is no limit to the number of moves that a user can make, because each move actually performs a goto to another function, not a conventional call.

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.

7 - Iterators and the Generic for

In this chapter, we cover how to write iterators for the generic **for**. We start with simple iterators, then we learn how to use all the power of the generic **for** to write more efficient iterators.

7.1 - Iterators and Closures

An *iterator* is any construction that allows you to iterate over the elements of a collection. In Lua, we typically represent iterators by functions: Each time we call that function, it returns a "next" element from the collection.

Any iterator needs to keep some state between successive calls, so that it knows where it is and how to proceed from there. Closures provide an excellent mechanism for that task. Remember that a closure is a function that accesses one or more local variables from its enclosing function. Those 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 external local variables. Therefore, a closure construction typically involves two functions: the closure itself; and a *factory*, the function that creates the closure.

As a simple example, let us write a simple iterator for a list. Unlike `ipairs`, this iterator does not return the index of each element, only the value:

```
function list_iter (t)
  local i = 0
  local n = table.getn(t)
  return function ()
           i = i + 1
           if i <= n then return t[i] end
         end
end
```

In this example, `list_iter` is the factory. Each time we call it, it creates a new closure (the iterator itself). That closure keeps its state in its external variables (`t`, `i`, and `n`) so that, each time we call it, it returns a next value from the list `t`. When there are no more values in the list, the iterator returns **nil**.

We can use such iterator with a **while**:

```
t = {10, 20, 30}
iter = list_iter(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 that kind of iteration:

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

The generic **for** does all the bookkeeping from an iteration loop: It calls the iterator factory; keeps the iterator function internally, so we do not need the `iter` variable; calls the iterator at each new iteration; and stops the loop when the iterator returns **nil**. (Later we will see that the generic **for** actually does more than that.)

As a more advanced example, we will write an iterator to traverse all the words from the current input file. To do this traversal, we need to keep two values: the current line and where we are in that line. With this data, we can always generate the next word. To keep it, we use two external local variables, `line` and `pos`:

```
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 s, e = string.find(line, "%w+", pos)
      if s then           -- found a word?
        pos = e + 1       -- next position is after this word
        return string.sub(line, s, e)     -- 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
```

The main part of the iterator function is the call to `string.find`. This call 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 the word, the function updates the current position to the first character after the word and returns that word. (The `string.sub` call extracts a substring from `line` between the given positions). 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 be difficult to write, but are easy to use. This is not a big problem; more often than not, end users programming in Lua do not define iterators, but only use those provided by the application.

7.2 - The Semantics of the Generic for

One drawback of those previous iterators is that we need to create a new closure for each new loop. For most 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 a few situations this overhead can be undesirable. In such cases, we can use the generic **for** itself to keep the iteration 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
```

where `<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. More often than not, the expression list has only one element, a call to an iterator factory. For instance, in the code

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

the list of variables is `k, v`; the list of expressions has the single element `pairs(t)`. Often the list of variables has only one variable too, as in

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

We call the first 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 **nil**s added as needed. (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. (Notice that, for the **for** structure, the invariant state has no meaning at all. It only gets this value from the initialization step and passes it when it calls the iterator function.) Then the **for** assigns the values returned by the iterator function to variables declared by its variable list. If the first value returned (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`.

7.3 - Stateless Iterators

As the name implies, a stateless iterator is an iterator that does not keep any state by itself. Therefore, we may use the same stateless iterator in multiple loops, avoiding the cost of creating new closures.

On each iteration, 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 arguments. A typical example of this kind of iterator is `ipairs`, which iterates over all elements in an array, as illustrated next:

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

The state of the iteration is the table being traversed (the invariant state, which does not change during the loop), plus the current index (the control variable). Both `ipairs` and the iterator it returns are quite simple; we could write them in Lua as follows:

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

function ipairs (a)
  return iter, a, 0
end
```

When Lua calls `ipairs(a)` in a **for** loop, it gets three values: the `iter` function as the iterator, `a` as the invariant state, and zero as the initial value for the control variable. Then, Lua calls `iter(a, 0)`, which results in `1, a[1]` (unless `a[1]` is already **nil**). In the second iteration, it calls `iter(a, 1)`, which results in `2, a[2]`, and so on, until the first nil element.

The `pairs` function, which iterates over all elements in a table, is similar, except that the iterator function is the `next` function, 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. (It returns also the value associated with that key, as a second return value.) The call `next(t, nil)` returns a first pair. When there are no more pairs, `next` returns **nil**.

Some people prefer to use `next` directly, without calling `pairs`:

```
for k, v in next, t do
  ...
end
```

Remember that the expression list of the **for** loop is adjusted to three results, so Lua gets `next`, `t`, and **nil**, exactly what it gets when it calls `pairs(t)`.

7.4 - Iterators with Complex State

Frequently, an iterator needs to keep more state than fits into a single invariant state and a control variable. The simplest solution is to use closures. An alternative solution is to pack all it needs into a table and use this table as the invariant state for the iteration. Using a table, an iterator can keep as much data as it needs along the loop. Moreover, it can change that data as it goes. Although the state is always the same table (and therefore invariant), the table contents change along the loop. Because such iterators have all their data in the state, they typically discard the second argument provided by the generic **for** (the iterator variable).

As an example of this technique, we will rewrite the iterator `allwords`, which traverses all the words from the current input file. This time, we will keep its state using a table with two fields, `line` and `pos`.

The function that starts the iteration is simple. It must return the iterator function and the initial state:

```
local iterator   -- to be defined later

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

The `iterator` function does the real work:

```
function iterator (state)
  while state.line do        -- repeat while there are lines
    -- search for next word
    local s, e = string.find(state.line, "%w+", state.pos)
    if s then                -- found a word?
      -- update next position (after this word)
      state.pos = e + 1
      return string.sub(state.line, s, e)
    else    -- word not found
      state.line = io.read() -- try next line...
      state.pos = 1          -- ... from first position
    end
  end
  return nil                 -- no more lines: end loop
end
```

Whenever it is possible, you should try to write stateless iterators, those that keep all their state in the **for** variables. With them, you do not create new objects when you start a loop. If you cannot fit your iteration into that model, then you should try closures. Besides being more elegant, typically a closure is more efficient than an iterator using tables: First, it is cheaper to create a closure than a table; second, access to upvalues is faster than access to table fields. Later we will see yet another way to write iterators, with coroutines. This is the most powerful solution, but a little more expensive.

7.5 - 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", 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)
  -- repeat for each line in the file
  for l in io.lines() do
    -- repeat for each word in the line
    for w in string.gfind(l, "%w+") do
      -- call the function
      f(w)
    end
  end
end
```

To use such iterator, we must supply the loop body as a function. If we only want to print each word, we simply use `print`:

```
allwords(print)
```

More 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 this second style (although we can recover this easiness with coroutines). 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.)

8 - 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: Most interpreted languages do the same.) The presence of a compilation phase may sound out of place in an interpreted language like Lua. However, the distinguishing feature of interpreted languages is not that they are not compiled, but that any compiler is part of the language runtime and that, therefore, 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 allows Lua to be called an interpreted language.

Previously, we introduced `dofile` as a kind of primitive operation to run chunks of Lua code. The `dofile` function is actually an auxiliary function; `loadfile` does the hard work. Like `dofile`, `loadfile` also 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, so that we can handle the error. 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, as it does the whole job in one call. However, `loadfile` is more flexible. In case of errors, `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 is much cheaper than several calls to `dofile`, because the program compiles the file only once.

The `loadstring` function is similar to `loadfile`, except that it reads its chunk from a string, not from a file. For instance, after the code

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

`f` will be a function that, when invoked, executes `i = i + 1`:

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

The `loadstring` function is powerful; it must be used with care. It is also an expensive function (when compared to its alternatives) and may result in incomprehensible code. Before you use it, make sure that there is no simpler way to solve the problem at hand.

Lua treats any independent chunk as the body of an anonymous function. For instance, for the chunk `"a = 1"`, `loadstring` returns the equivalent of

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

Like any other function, chunks can declare local variables and return values:

```
f = loadstring("local a = 10; return a + 20")
print(f())          --> 30
```

Both `loadstring` and `loadfile` never raise errors. In case of any kind of error, both functions return **nil** plus an error message:

```
print(loadstring("i i"))
  --> nil     [string "i i"]:1: `=' expected near `i'
```

Moreover, both functions never have any kind of side effect. They only compile the chunk to an internal representation and return the result, as an anonymous function. A common mistake is to assume that `loadfile` (or `loadstring`) defines functions. In Lua, function definitions are assignments; as such, they are made 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")
```

After this command, `foo` is compiled, but it is not defined yet. To define it, you must run the chunk:

```
f()            -- defines `foo'
foo("ok")      --> ok
```

If you want to do a quick-and-dirty `dostring` (i.e., to load and run a chunk) you may call the result from `loadstring` directly:

```
loadstring(s)()
```

However, if there is any syntax error, `loadstring` will return **nil** and the final error message will be an `"attempt to call a nil value"`. For clearer error messages, use `assert`:

```
assert(loadstring(s))()
```

Usually, it does not make sense to use `loadstring` on a literal string. For instance, the code

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

is roughly equivalent to

```
f = function () i = i + 1 end
```

but the second code is much faster, because it is compiled only once, when the chunk is compiled. In the first code, each call to `loadstring` involves a new compilation. However, the two codes are not completely equivalent, because `loadstring` does not compile with lexical scoping. To see the difference, let us change the previous examples a little:

```
local i = 0
f = loadstring("i = i + 1")
g = function () i = i + 1 end
```

The `g` function manipulates the local `i`, as expected, but `f` manipulates a global `i`, because `loadstring` always compiles its strings in a global environment.

The most typical use of `loadstring` is to run external code, that is, pieces of code that come from outside your program. For instance, you may want to plot a function defined by the user; the user enters the function code and then you use `loadstring` to evaluate it. Note that `loadstring` expects a chunk, that is, statements. If you want to evaluate an expression, you must prefix it with **return**, so that you get a statement that returns the value of the given expression. See the example:

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

The function returned by `loadstring` is a regular function, so you can call it several times:

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

```
    end
```

In a production-quality program that needs to run external code, you should handle any errors reported by `loadstring`. Moreover, if the code cannot be trusted, you may want to run the new chunk in a protected environment, to avoid unpleasant side effects when running the code.

8.1 - The `require` Function

Lua offers a higher-level function to load and run libraries, called `require`. Roughly, `require` does the same job as `dofile`, but with two important differences. First, `require` searches for the file in a path; second, `require` controls whether a file has already been run to avoid duplicating the work. Because of these features, `require` is the preferred function in Lua for loading libraries.

The path used by `require` is a little different from typical paths. Most programs use paths as a list of directories wherein to search for a given file. However, ANSI C (the abstract platform where Lua runs) does not have the concept of directories. Therefore, the path used by `require` is a list of *patterns*, each of them specifying an alternative way to transform a virtual file name (the argument to `require`) into a real file name. More specifically, each component in the path is a file name containing optional interrogation marks. For each component, `require` replaces each `` `?´ `` by the virtual file name and checks whether there is a file with that name; if not, it goes to the next component. The components in a path are separated by semicolons (a character seldom used for file names in most operating systems). For instance, if the path is

```
    ?;?.lua;c:\windows\?;/usr/local/lua/?/?.lua
```

then the call `require"lili"` will try to open the following files:

```
    lili
    lili.lua
    c:\windows\lili
    /usr/local/lua/lili/lili.lua
```

The only things that `require` fixes is the semicolon (as the component separator) and the interrogation mark; everything else (such as directory separators or file extensions) is defined in the path.

To determine its path, `require` first checks the global variable `LUA_PATH`. If the value of `LUA_PATH` is a string, that string is the path. Otherwise, `require` checks the environment variable `LUA_PATH`. Finally, if both checks fail, `require` uses a fixed path (typically `"?;?.lua"`, although it is easy to change that when you compile Lua).

The other main job of `require` is to avoid loading the same file twice. For that purpose, it keeps a table with the names of all loaded files. If a required file is already in the table, `require` simply returns. The table keeps the virtual names of the loaded files, not their real names. Therefore, if you load the same file with two different virtual names, it will be loaded twice. For instance, the command `require"foo"` followed by `require"foo.lua"`, with a path like `"?;?.lua"`, will load the file `foo.lua` twice. You can access this control table through the global variable `_LOADED`. Using this table, you can check which files have been loaded; you can also fool `require` into running a file twice. For instance, after a successful `require"foo"`, `_LOADED["foo"]` will not be **nil**. If you then assign **nil** to `_LOADED["foo"]`, a subsequent `require"foo"` will run the file again.

A component does not need to have interrogation marks; it can be a fixed file name, such as the last component in the following path:

```
?;?.lua;/usr/local/default.lua
```

In this case, whenever `require` cannot find another option, it will run this fixed file. (Of course, it only makes sense to have a fixed component as the last component in a path.) Before `require` runs a chunk, it defines a global variable `_REQUIREDNAME` containing the virtual name of the file being required. We can use these facilities to extend the functionality of `require`. In an extreme example, we may set the path to something like `"/usr/local/lua/newrequire.lua"`, so that every call to `require` runs `newrequire.lua`, which can then use the value of `_REQUIREDNAME` to actually load the required file.

8.2 - C Packages

Because it is easy to interface Lua with C, it is also easy to write packages for Lua in C. Unlike packages written in Lua, however, C packages need to be loaded and linked with an application before use. In most popular systems, the easiest way to do that is with a dynamic linking facility. However, this facility is not part of the ANSI C specification; that is, there is no portable way to implement it.

Usually, Lua does not include any facility that cannot be implemented in ANSI C. However, dynamic linking is different. We can view it as the mother of all other facilities: Once we have it, we can dynamically load any other facility that is not in Lua. Therefore, in this particular case, Lua breaks its compatibility rules and implements a dynamic linking facility for several platforms, using conditional code. The standard implementation offers this support for Windows (DLL), Linux, FreeBSD, Solaris, and some other Unix implementations. It should not be difficult to extend this facility to other platforms; check your distribution. (To check it, run `print(loadlib())` from the Lua prompt and see the result. If it complains about bad arguments, then you have dynamic linking facility. Otherwise, the error message indicates that this facility is not supported or not installed.)

Lua provides all the functionality of dynamic linking in a single function, called `loadlib`. Its has two string arguments: the complete path of the library and the name of an initialization function. So, a typical call to it looks like the next fragment:

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

The `loadlib` function loads the given library and links Lua to it. However, it does not open the library (that is, it does not call the initialization function); instead, it returns the initialization function as a Lua function, so that we can call it directly from Lua. If there is any error loading the library or finding the initialization function, `loadlib` returns **nil** plus an error message. We can improve our previous fragment so that it checks for errors and calls the initialization function:

```
local path = "/usr/local/lua/lib/libluasocket.so"
-- or path = "C:\\windows\\luasocket.dll"
local f = assert(loadlib(path, "luaopen_socket"))
f()  -- actually open the library
```

Typically, we could expect a library distribution to include a stub file similar to that previous code fragment. Then, to install the library, we put the actual binary shared library anywhere, edit the stub to reflect the real path, and then add the stub file in a directory in our `LUA_PATH`. With this setting, we can use the regular `require` function to open the C library.

8.3 - 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 ends the current chunk and returns to the application.

Any unexpected condition that Lua encounters raises an error. Errors occur when you (that is, your program) try to add values that are not numbers, to call values that are not functions, to index values that are not tables, and so on. (You can modify this behavior using *metatables*, as we will see later.) You can also explicitly raise an error calling the `error` function; its argument is the error message. Usually, that function is the appropriate way to handle errors in your code:

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

Such combination of `if not ... then error end` is so common that Lua has a built-in function just for that job, called `assert`:

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

The `assert` function checks whether its first argument is not false and simply returns that argument; if the argument is false (that is, **false** or **nil**), `assert` raises an error. Its second argument, the message, is optional, so that if you do not want to say anything in the error message, you do not have to. Beware, however, that `assert` is a regular function. As such, Lua always evaluates its arguments before calling the function. Therefore, if you have 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 it can raise an error, calling the `error` function. There are no fixed rules for choosing between those two options, but we can provide a general guideline: An exception that is easily avoided should raise an error; otherwise, it should return an error code.

For instance, let us consider the `sin` function. 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

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

However, we could as easily check this exception *before* calling the function:

```
if not tonumber(x) then     -- error: x is not a number
  ...
```

Usually, however, we check neither the argument nor the result of a call to `sin`; if the argument is not a number, it means probably something wrong in our program. In such situations, to stop the computation and to issue an error message is the simplest and most practical way to handle the exception.

On the other hand, let us consider the `io.open` function, which opens a file. How should it behave when called to read 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 to try 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 **nil**, plus a string with the error message. In this way, you 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 you do not want to handle such situations, but still want to play safe, you simply use `assert` to guard the operation:

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

This is a typical Lua idiom: If `io.open` fails, `assert` will raise an error.

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

Notice how the error message, which is the second result from `io.open`, goes as the second argument to `assert`.

8.4 - Error Handling and Exceptions

For many applications, you do not need to do any error handling in Lua. Usually, 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 and 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 commands.

If you need to handle errors in Lua, you should use the `pcall` function (*protected call*) to encapsulate your code.

Suppose you want to run a piece of Lua code and to catch any error raised while running that code. Your first step is to encapsulate that piece of code in a function; let us call it `foo`:

```
function foo ()
    ...
  if unexpected_condition then error() end
    ...
  print(a[i])    -- potential error: `a' may not be a table
    ...
end
```

Then, you call `foo` with `pcall`:

```
if pcall(foo) then
```

```
      -- no errors while running `foo'
      ...
    else
      -- `foo' raised an error: take appropriate actions
      ...
    end
```

Of course, you can call `pcall` with an anonymous function:

```
    if pcall(function () ... end) then ...
    else ...
```

The `pcall` function calls its first argument in *protected mode*, so that it catches any errors while the function is running. 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. Any Lua value that you pass to `error` will be returned by `pcall`:

```
    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 or error.

8.5 - Error Messages and Tracebacks

Although you can use a value of any type as an error message, usually error messages 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 message; otherwise, the error message is the value passed to the `error` function. In any case, Lua tries to add some information about the location where the error happened:

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

    local status, err = pcall(function () error("my error") end)
    print(err)
     --> stdin:1: my error
```

The location information gives the file name (`stdin`, in the example) plus the line number (1, in the example).

The `error` function has an additional second parameter, which gives the *level* where it should report the error; with it, you can blame someone else for the error. For instance, suppose you write a function and its first task is to check whether it was called correctly:

```
    function foo (str)
      if type(str) ~= "string" then
        error("string expected")
      end
      ...
    end
```

Then, someone calls your function with a wrong argument:

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

Lua points its finger to your function---after all, it was `foo` that called `error`---and not to the real culprit, the caller. To correct that, you inform `error` that the error you are reporting occurred on level 2 in the calling hierarchy (level 1 is your own function):

```
function foo (str)
  if type(str) ~= "string" then
    error("string expected", 2)
  end
  ...
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 went from it to the error point). Consequently, if we want a traceback, we must build it before `pcall` returns. To do that, Lua provides the `xpcall` function. Besides the function to be called, it receives a second argument, an *error handler function*. In case of errors, Lua calls that error 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 error handlers are `debug.debug`, which gives you a Lua prompt so that you can inspect by yourself what was going on when the error happened (later we will see more about that, when we discuss the debug library); 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. You also can call `debug.traceback` at any moment to get a traceback of the current execution:

```
print(debug.traceback())
```

9 - Coroutines

A *coroutine* is similar to a thread (in the sense of multithreading): a line of execution, with its own stack, its own local variables, and its own instruction pointer; but sharing global variables and mostly anything else with other coroutines. The main difference between threads and coroutines is that, conceptually (or literally, in a multiprocessor machine), a program with threads runs several threads concurrently. Coroutines, on the other hand, are collaborative: A program with coroutines is, at any given time, running only one of its coroutines and this running coroutine only suspends its execution when it explicitly requests to be suspended.

Coroutine is a powerful concept. As such, several of its main uses are complex. Do not worry if you do not understand some of the examples in this chapter on your first reading. You can read the rest of the book and come back here later. But please come back. It will be time well spent.

9.1 - Coroutine Basics

Lua offers all its coroutine functions packed in the `coroutine` table. The `create` function creates new coroutines. It has a single argument, a function with the code that the coroutine will run. It returns a value of type `thread`, which represents the new coroutine. Quite often, the argument to `create` is an anonymous function, like here:

```
co = coroutine.create(function ()
       print("hi")
     end)

print(co)    --> thread: 0x8071d98
```

A coroutine can be in one of three different states: suspended, running, and dead. When we create a coroutine, it starts in the suspended state. That means that a coroutine does not run its body automatically when we create it. We can check the state of a coroutine with the `status` function:

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

The function `coroutine.resume` (re)starts the execution of a coroutine, changing its state from suspended to running:

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

In this example, the coroutine body simply prints `"hi"` and terminates, leaving the coroutine in the dead state, from which it cannot return:

```
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 `yield` function, which allows a running coroutine to suspend its 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, 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 again:

```
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 finished the loop and then returned, so the coroutine is dead now. 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. 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.

A useful facility in Lua is that a pair resume-yield can exchange data between them. The first `resume`, which has no corresponding `yield` waiting for it, passes its extra arguments as arguments to the coroutine main function:

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

A call to `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, `yield` returns any extra arguments passed to the corresponding `resume`:

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

For those that already know something about coroutines, it is important to clarify some concepts before we go on. Lua offers what I call *asymmetric coroutines*. That 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 any coroutine to another.

Some people call asymmetric coroutine *semi-coroutines* (because they are not symmetrical, they are not really *co*). However, other people use the same term *semi-coroutine* to denote a restricted implementation of coroutines, where a coroutine can only suspend its execution when it is not inside any auxiliary 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 several interesting constructions that we can write with true coroutines. Lua offers true, asymmetric coroutines. Those that prefer symmetric coroutines can implement them on top of the asymmetric facilities of Lua. It is an easy task. (Basically, each transfer does a yield followed by a resume.)

9.2 - Pipes and Filters

One of the most paradigmatic examples of coroutines is in 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). Typically, these two functions look like this:

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

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

(In that implementation, both the producer and the consumer run forever. It is an easy task to change them to stop when there is no more data to be handled.) The problem here is how to match `send` with `receive`. It is a typical case of a 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, because a resume-yield pair turns upside-down the typical relationship between caller and 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. 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 be a coroutine:

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

In this design, the program starts 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 restarts it again. Therefore, we have what we call a *consumer-driven* design.

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 complete code would be like this:

```
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 ()
        local line = 1
        while true do
          local x = receive(prod)   -- get new value
          x = string.format("%5d %s", line, x)
          send(x)      -- send it to consumer
          line = line + 1
        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
```

The final bit simply creates the components it needs, connects them, and starts the final consumer:

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

Or better yet:

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

If you thought about Unix pipes after reading the previous example, you are not alone. After all, coroutines are a kind of (non-preemptive) multithreading. While in 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 the same cost of a function call), so the writer and the reader can go hand in hand.

9.3 - Coroutines as Iterators

We can see loop iterators as a quite specific 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. Actually, coroutines provide a powerful tool for this task. Again, the key feature is their ability to turn upside-down the relationship between caller and callee. With this feature, we can write iterators without worrying about how to keep state between successive calls to the iterator.

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 iterator, but it is not so difficult to write a recursive function that generates all those 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 as follows:

```
function permgen (a, n)
  if n == 0 then
    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 see it working, we should define an appropriate `printResult` function and call `permgen` with proper arguments:

```
function printResult (a)
  for i,v in ipairs(a) do
    io.write(v, " ")
  end
  io.write("\n")
end

permgen ({1,2,3,4}, 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)
  if n == 0 then
    coroutine.yield(a)
  else
  ...
```

Then, we define a factory that arranges for the generator to run inside a coroutine, and then create the iterator function. The iterator simply resumes the coroutine to produce the next permutation:

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

With that machinery in place, it is trivial to iterate over all permutations of an array with a **for** statement:

```
for p in perm{"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 `perm` function 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 errors. Using `wrap`, we can write `perm` as follows:

```
    function perm (a)
      local n = table.getn(a)
      return coroutine.wrap(function () permgen(a, n) 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 errors.

9.4 - Non-Preemptive Multithreading

As we saw earlier, coroutines are 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 "real" multithreading, coroutines are non preemptive. While a coroutine is running, it cannot be stopped from the outside. It only suspends execution 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. You do not need to be paranoid about synchronization bugs, because all synchronization among threads is explicit in the program. You only have to ensure that a coroutine only yields 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 most applications, this is an unacceptable behavior, which leads many programmers to disregard coroutines as a real alternative to conventional multithreading. As we will see here, that 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. Of course, to download several remote files, we must know how to download one remote file. In this example, we will use the *LuaSocket* library, developed by Diego Nehab. 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:

```
    require "luasocket"
```

Then, we define the host and the file we want to download. In this example, we will download the HTML 3.2 Reference Specification from the World Wide Web Consortium site:

```
    host = "www.w3.org"
```

```
file = "/TR/REC-html32.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))
```

The operation returns a connection object, which we use to send the file request:

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

The `receive` method always returns a string with what it read plus another string with the status of the operation. When the host closes the connection we break the receive loop.

Finally, 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 only start reading a file 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 simultaneously. Then, while a connection has no data available, the program can read from another connection. Clearly, coroutines offer a convenient way to structure those 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, let us first rewrite the previous download code as a function:

```
function download (host, file)
  local c = assert(socket.connect(host, 80))
  local count = 0     -- counts number of bytes read
  c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
  while true do
    local s, status = receive(c)
    count = count + string.len(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 only counts the file size, instead of writing the file to the standard output. (With several threads reading several files, the output would intermix 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)
  return connection:receive(2^10)
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:timeout(0)    -- do not block
  local s, status = connection:receive(2^10)
  if status == "timeout" then
    coroutine.yield(connection)
  end
```

```
    return s, status
  end
```

The call to `timeout(0)` makes any operation over the connection a non-blocking operation. When the operation 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. (Later we will see another version where the dispatcher needs the timed-out connection.) Notice that, even in case of a timeout, the connection returns what it read until the timeout, so `receive` always returns `s` to its caller.

The next function ensures that each download runs in an individual thread:

```
  threads = {}    -- list of all live threads
  function get (host, file)
    -- create coroutine
    local co = coroutine.create(function ()
      download(host, file)
    end)
    -- insert it in the list
    table.insert(threads, co)
  end
```

The table `threads` keeps a list of all live threads, for the dispatcher.

The dispatcher is simple. It is mainly a loop that goes through all threads, calling one by one. It must also remove from the list the threads that finish their tasks. It stops the loop when there are no more threads to run:

```
  function dispatcher ()
    while true do
      local n = table.getn(threads)
      if n == 0 then break end   -- no more threads to run
      for i=1,n do
        local status, res = coroutine.resume(threads[i])
        if not res then    -- thread finished its task?
          table.remove(threads, i)
          break
        end
      end
    end
  end
```

Finally, the main program creates the threads it needs and calls the dispatcher. For instance, to download four documents from the W3C site, the main program could be like this:

```
  host = "www.w3.org"

  get(host, "/TR/html401/html40.txt")
  get(host,"/TR/2002/REC-xhtml1-20020801/xhtml1.pdf")
  get(host,"/TR/REC-html32.html")
  get(host,
      "/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt")

  dispatcher()   -- main loop
```

My machine takes six seconds to download those four files using coroutines. With the sequential implementation, it takes more than twice that time (15 seconds).

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 almost 30 times more CPU than the sequential solution.

To avoid this behavior, we can use the `select` function 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 only have to change the dispatcher. The new version is like this:

```
function dispatcher ()
  while true do
    local n = table.getn(threads)
    if n == 0 then break end   -- no more threads to run
    local connections = {}
    for i=1,n do
      local status, res = coroutine.resume(threads[i])
      if not res then    -- thread finished its task?
        table.remove(threads, i)
        break
      else    -- timeout
        table.insert(connections, res)
      end
    end
    if table.getn(connections) == n then
      socket.select(connections)
    end
  end
end
```

Along the inner loop, this new dispatcher collects the timed-out connections in table `connections`. Remember that `receive` passes such connections to `yield`; thus `resume` returns them. When all connections time out, the dispatcher calls `select` to wait for any of those connections to change status. This final implementation runs as fast as the first implementation with coroutines. Moreover, as it does no busy waits, it uses just a little more CPU than the sequential implementation.

10 - Complete Examples

To end this introduction about the language, we show two complete programs that illustrate different facilities of Lua. The first example is a real program from the Lua site; it illustrates the use of Lua as a data description language. The second example is an implementation of the *Markov chain algorithm*, described by Kernighan & Pike in their book *The Practice of Programming* (Addison-Wesley, 1999).

10.1 - Data Description

The Lua site keeps a database containing a sample of projects around the world that use Lua. We represent each entry in the database by a constructor in an auto-documented way, as the following example shows:

```
entry{
  title = "Tecgraf",
  org = "Computer Graphics Technology Group, PUC-Rio",
  url = "http://www.tecgraf.puc-rio.br/",
  contact = "Waldemar Celes",
  description = [[
    TeCGraf is the result of a partnership between PUC-Rio,
    the Pontifical Catholic University of Rio de Janeiro,
    and <A HREF="http://www.petrobras.com.br/">PETROBRAS</A>,
    the Brazilian Oil Company.
    TeCGraf is Lua's birthplace,
    and the language has been used there since 1993.
    Currently, more than thirty programmers in TeCGraf use
    Lua regularly; they have written more than two hundred
    thousand lines of code, distributed among dozens of
    final products.]]
  }
```

The interesting thing about this representation is that a file with a sequence of such entries is a Lua program, which does a sequence of calls to a function `entry`, using the tables as the call arguments.

Our goal is to write a program that shows that data in HTML, so that it becomes the web page `http://www.lua.org/uses.html`. Because there are many projects, the final page first shows a list of all project titles, and then shows the details of each project. The result of the program is something like this:

```
<HTML>
<HEAD><TITLE>Projects using Lua</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
Here are brief descriptions of some projects around the
world that use <A HREF="home.html">Lua</A>.
<BR>
<UL>
<LI><A HREF="#1">TeCGraf</A>
<LI> ...
</UL>

<H3>
<A NAME="1" HREF="http://www.tecgraf.puc-rio.br/">TeCGraf</A>
<BR>
<SMALL><EM>Computer Graphics Technology Group,
           PUC-Rio</EM></SMALL>
</H3>

    TeCGraf is the result of a partnership between
    ...
    distributed among dozens of final products.<P>
Contact: Waldemar Celes

<A NAME="2"></A><HR>
...

</BODY></HTML>
```

To read the data, all the program has to do is to give a proper definition for `entry`, and then run the data file as a program (with `dofile`). Note that we have to traverse all the entries twice, first for the title list, and again for the project descriptions. A first approach would be to collect all entries in an array. However, because Lua compiles so fast, there is a second attractive solution: run the data file twice, each time with a different definition for `entry`. We follow this approach in the next program.

First, we define an auxiliary function for writing formatted text (we already saw this function in Section 5.2):

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

The `BEGIN` function simply writes the page header, which is always the same:

```
function BEGIN()
  io.write([[
    <HTML>
    <HEAD><TITLE>Projects using Lua</TITLE></HEAD>
    <BODY BGCOLOR="#FFFFFF">
    Here are brief descriptions of some projects around the
    world that use <A HREF="home.html">Lua</A>.
    <BR>
  ]])
end
```

The first definition for `entry` writes each title project as a list item. The argument `o` will be the table describing the project:

```
function entry0 (o)
  N=N + 1
  local title = o.title or '(no title)'
  fwrite('<LI><A HREF="#%d">%s</A>\n', N, title)
end
```

If `o.title` is **nil** (that is, the field was not provided), the function uses a fixed string `"(no title)"`.

The second definition writes all useful data about a project. It is a little more complex, because all items are optional.

```
function entry1 (o)
  N=N + 1
  local title = o.title or o.org or 'org'
  fwrite('<HR>\n<H3>\n')
  local href = ''

  if o.url then
    href = string.format(' HREF="%s"', o.url)
  end
  fwrite('<A NAME="%d"%s>%s</A>\n', N, href, title)

  if o.title and o.org then
    fwrite('<BR>\n<SMALL><EM>%s</EM></SMALL>', o.org)
  end
  fwrite('\n</H3>\n')

  if o.description then
    fwrite('%s', string.gsub(o.description,
                             '\n\n\n*', '<P>\n'))
    fwrite('<P>\n')
  end

  if o.email then
    fwrite('Contact: <A HREF="mailto:%s">%s</A>\n',
           o.email, o.contact or o.email)
  elseif o.contact then
    fwrite('Contact: %s\n', o.contact)
  end
end
```

(To avoid conflict with HTML, which uses double quotes, we used only single quotes in this program.) The last function closes the page:

```
function END()
  fwrite('</BODY></HTML>\n')
```

```
end
```

Finally, the main program starts the page, runs the data file with the first definition for `entry` (`entry0`) to create the list of titles, then runs the data file again with the second definition for `entry`, and closes the page:

```
BEGIN()

N = 0
entry = entry0
fwrite('<UL>\n')
dofile('db.lua')
fwrite('</UL>\n')

N = 0
entry = entry1
dofile('db.lua')

END()
```

10.2 - Markov Chain Algorithm

Our second example is an implementation of the *Markov chain algorithm*. The program generates random text, based on what words may follow a sequence of *n* previous words in a base text. For this implementation, we will use *n=2*.

The first part of the program reads the base text and builds a table that, for each prefix of two words, gives a list with the words that follow that prefix in the text. After building the table, the program uses the table to generate random text, wherein each word follows two previous words with the same probability of the base text. As a result, we have text that is very, but not quite, random. For instance, when applied over this book, the output of the program has pieces like "*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.*"

We will code each prefix by its two words concatenated with spaces in between:

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

We use the string `NOWORD` (`"\n"`) to initialize the prefix words and to mark the end of the text. For instance, for the following text

```
the more we try the more we do
```

the table of following words would be

```
{  ["\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 global variable `statetab`. To insert a new word in a prefix list of this table, we use the following function:

```
function insert (index, value)
  if not statetab[index] then
    statetab[index] = {value}
  else
    table.insert(statetab[index], 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 uses the predefined function `table.insert` to insert 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. For each prefix, we keep a list of all words that follow it.

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 randomly a next word from the list of valid next words, prints that word, and updates `w1` and `w2`. Next we show the complete program.

```
-- Markov Chain Program in Lua

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 s, e = string.find(line, "%w+", pos)
      if s then      -- found a word?
        pos = e + 1  -- update next position
        return string.sub(line, s, e)   -- 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 (index, value)
  if not statetab[index] then
    statetab[index] = {n=0}
  end
  table.insert(statetab[index], value)
end

local N  = 2
local MAXGEN = 10000
local NOWORD = "\n"

-- build table
statetab = {}
local w1, w2 = NOWORD, NOWORD
for w in allwords() do
```

```
    insert(prefix(w1, w2), w)
    w1 = w2; w2 = w;
  end
  insert(prefix(w1, w2), NOWORD)

  -- generate text
  w1 = NOWORD; w2 = NOWORD     -- reinitialize
  for i=1,MAXGEN do
    local list = statetab[prefix(w1, w2)]
    -- choose a random item from list
    local r = math.random(table.getn(list))
    local nextword = list[r]
    if nextword == NOWORD then return end
    io.write(nextword, " ")
    w1 = w2; w2 = nextword
  end
```

11 - Data Structures

Tables in Lua are not a data structure; they are *the* data structure. All structures that other languages offer---arrays, records, lists, queues, sets---are represented with tables in Lua. More to the point, tables implement all these structures efficiently.

In 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 that), tables are more powerful than arrays and lists; many algorithms are simplified to the point of triviality with the use of tables. For instance, you 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 show how you can implement typical data structures with tables and will provide some examples of their use. 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 our chapters about the language, but I will repeat it here for completeness.

11.1 - Arrays

We implement arrays in Lua simply by indexing tables with integers. Therefore, arrays do not have a fixed size, but grow as we need. Usually, when we initialize the array we define its size indirectly. For instance, after the following code

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

any attempt to access a field outside the range 1-1000 will return **nil**, instead of zero.

You can start an array at index 0, 1, or any other value:

```
  -- creates 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 1. The Lua libraries adhere to this convention; so, if your arrays also start with 1, you will be able to use their functions directly.

We can use constructors 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 you need (well, up to a few million elements).

11.2 - Matrices and Multi-Dimensional Arrays

There are two main ways to represent matrices in Lua. The first one is to use an array of arrays, that is, a table wherein each element is another table. For instance, you can create a matrix of zeros with dimensions `N` by `M` with the following code:

```
    mt = {}          -- create the matrix
    for i=1,N do
      mt[i] = {}     -- create a new row
      for j=1,M do
        mt[i][j] = 0
      end
    end
```

Because tables are objects in Lua, you have to create each row explicitly to create a matrix. On the one hand, this is certainly more verbose than simply declaring a matrix, as you do in C or Pascal. On the other hand, that gives you more flexibility. For instance, you can create a triangular matrix changing the line

```
      for j=1,M do
```

in the previous example to

```
      for j=1,i do
```

With that code, the triangular matrix uses only half the memory of the original one.

The second way to represent a matrix in Lua is by composing the two indices into a single one. If the two indices are integers, you can multiply the first one by a constant and then add the second index. With this approach, the following code would create our matrix of zeros with dimensions `N` by `M`:

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

If the indices are strings, you can create a single index concatenating both indices with a character in between to separate them. For instance, you can index a matrix `m` with string indices `s` and `t` with the code `m[s..':'..t]`, provided that both `s` and `t` do not contain colons (otherwise, pairs like (`"a:"`, `"b"`) and (`"a"`, `":b"`) would collapse into a single index `"a::b"`). When in doubt, you can use a control character like `` `\0´ `` to separate the indices.

Quite often, applications use a *sparse matrix*, a matrix wherein most elements are 0 or nil. For instance, you can represent a graph by its adjacency matrix, which has the value `x` in position `m,n` only when the nodes `m` and `n` are connected with cost `x`; when those 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, you will need a matrix with a hundred million entries (a square matrix with 10,000 columns and 10,000 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 400 MB of memory, but you do not need those techniques when programming in Lua. Because arrays are represented by tables, they are naturally sparse. With our first representation (tables of tables), you will need ten thousand tables, each one with about five elements, with a grand total of fifty thousand entries. With the second representation, you will have a single table, with fifty thousand entries in it. Whatever the representation, you only need space for the non-nil elements.

11.3 - Linked Lists

Because tables are dynamic entities, it is easy to implement linked lists in Lua. Each node is represented by a table and links are simply table fields that contain references to other tables. For instance, to implement a basic list, where each node has two fields, `next` and `value`, we need a variable to be the list root:

```
list = nil
```

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

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

To traverse the list, we write:

```
local l = list
while l do
  print(l.value)
  l = l.next
end
```

Other kinds of lists, such as double-linked lists or circular lists, are also implemented easily. However, you seldom need those structures in Lua, because usually there is a simpler way to represent your data without using lists. For instance, we can represent a stack with an (unbounded) array, with a field `n` pointing to the top.

11.4 - Queues and Double Queues

Although we can implement queues trivially using `insert` and `remove` (from the `table` library), this implementation can be too slow for large structures. A more efficient implementation uses two indices, one for the first and another for the last element:

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

To avoid polluting the global space, we will define all list operations inside a table, properly called `List`. Therefore, we rewrite our last example like this:

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

Now, we can insert or remove an element at both ends in constant time:

```
function List.pushleft (list, value)
  local first = list.first - 1
  list.first = first
  list[first] = value
end

function List.pushright (list, value)
  local last = list.last + 1
  list.last = last
  list[last] = value
end

function List.popleft (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 List.popright (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 you use this structure in a strict queue discipline, calling only `pushright` and `popleft`, both `first` and `last` will increase continually. However, because we represent arrays in Lua with tables, you can index them either from 1 to 20 or from 16,777,216 to 16,777,236. Moreover, because Lua uses double precision to represent numbers, your program can run for two hundred years, doing one million insertions per second, before it has problems with overflows.

11.5 - Sets and Bags

Suppose you want to list all identifiers used in a program source; somehow you need to filter the reserved words out of your listing. Some C programmers could be tempted to represent the set of reserved words as an array of strings, and then to 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 or a hash table 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, you just index the table and test whether the result is **nil** or not. In our example, we could write the next code:

```
reserved = {
  ["while"] = true,     ["end"] = true,
  ["function"] = true,  ["local"] = true,
}

for w in allwords() do
  if reserved[w] then
    -- `w' is a reserved word
    ...
```

(Because **while** is a reserved word in Lua, we cannot use it as an identifier. Therefore, we cannot write `while = 1`; instead, we use the `["while"] = 1` notation.)

You 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", }
```

11.6 - String Buffers

Suppose you are building a string piecemeal, for instance reading a file line by line. Your typical code would look like this:

```
-- WARNING: bad code ahead!!
local buff = ""
for line in io.lines() do
buff = buff .. line .. "\n"
end
```

Despite its innocent look, that code in Lua can cause a huge performance penalty for large files: For instance, it takes almost a minute to read a 350 KB file. (That is why Lua provides the `io.read("*all")` option, which reads the whole file in 0.02 seconds.)

Why is that? Lua uses a true garbage-collection algorithm; when it detects that the program is using too much memory, it goes through all its data structures and frees those structures that are not in use anymore (the garbage). Usually this algorithm has a good performance (it is not by chance that Lua is so fast), but the above loop takes the worst of the algorithm.

To understand what happens, let us assume that we are in the middle of the read loop; `buff` is already a string with 50 KB and each line has 20 bytes. When Lua concatenates `buff..line.."\n"`, it creates a new string with 50,020 bytes and copies 50 KB from `buff` into this new string. That is, for each new line, Lua moves 50 KB of memory, and growing. After reading 100 new lines (only 2 KB), Lua has already moved more than 5 MB of memory. To make things worse, after the assignment

```
    buff = buff .. line .. "\n"
```

the old string is now garbage. After two loop cycles, there are two old strings making a total of more than 100 KB of garbage. So, Lua decides, quite correctly, that it is a good time to run its garbage collector and so it frees those 100 KB. The problem is that this will happen every two cycles and so Lua will run its garbage collector two thousand times before reading the whole file. Even with all this work, its memory usage will be approximately three times the file size.

This problem is not peculiar to Lua: Other languages with true garbage collection, and where strings are immutable objects, present a similar behavior, Java being the most famous example. (Java offers the structure `StringBuffer` to ameliorate the problem.)

Before we continue, we should remark that, despite all I said, that situation is not a common problem. For small strings, the above loop is OK. To read a whole file, we use the `"*all"` option, which reads it at once. However, sometimes there are no simple solutions. Then, the only solution is a more efficient algorithm. Here we show one.

Our original loop took a linear approach to the problem, concatenating small strings one by one into the accumulator. This new algorithm avoids this, using a binary approach instead. It concatenates several small strings among them and, occasionally, it concatenates the resulting large strings into larger ones. The heart of the algorithm is a stack that keeps the large strings already created in its bottom, while small strings enter through the top. The main invariant of this stack is similar to that of the popular (among programmers, at least) *Tower of Hanoi*: A string in the stack can never sit over a shorter string. Whenever a new string is pushed over a shorter one, then (and only then) the algorithm concatenates both. This concatenation creates a larger string, which now may be larger than its neighbor in the previous floor. If that happens, they are joined too. Those concatenations go down the stack until the loop reaches a larger string or the stack bottom.

```
function newStack ()
  return {""}   -- starts with an empty string
end

function addString (stack, s)
  table.insert(stack, s)    -- push 's' into the the stack
  for i=table.getn(stack)-1, 1, -1 do
    if string.len(stack[i]) > string.len(stack[i+1]) then
      break
    end
    stack[i] = stack[i] .. table.remove(stack)
  end
end
```

To get the final contents of the buffer, we just need to concatenate all strings down to the bottom. The `table.concat` function does exactly that: It concatenates all strings of a list.

Using this new data structure, we can rewrite our program as follows:

```
local s = newStack()
for line in io.lines() do
  addString(s, line .. "\n")
end
s = toString(s)
```

This new program reduces our original time to read a 350 KB file from 40 seconds to 0.5 seconds. The call `io.read("*all")` is still faster, finishing the job in 0.02 seconds.

Actually, when we call `io.read("*all")`, `io.read` uses exactly the data structure that we presented here, but implemented in C. Several other functions in the Lua libraries also use this C implementation. One of these functions is `table.concat`. With `concat`, we can simply collect all strings in a table and then concatenate all of them at once. Because `concat` uses the C implementation, it is efficient even for huge strings.

The `concat` function 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
  table.insert(t, line)
end
s = table.concat(t, "\n") .. "\n"
```

(The `io.lines` iterator returns each line without the newline.) `concat` inserts the separator between the strings, but not after the last one, so we have to add the last newline. This last concatenation duplicates the resulting string, which can be quite big. There is no option to make `concat` insert this extra separator, but we can deceive it, inserting an extra empty string in `t`:

```
table.insert(t, "")
s = table.concat(t, "\n")
```

The extra newline that `concat` adds before this empty string is at the end of the resulting string, as we wanted.

12 - Data Files and Persistence

When dealing with data files, it is usually much easier to write the data than to read them 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 kinds of data that a correct file may contain, a robust program should also handle bad files gracefully. Because of that, coding robust input routines is always difficult.

As we saw in the example of Section 10.1, 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 runs, builds the data into the program. With table constructors, these chunks can look remarkably like a plain data file.

As usual, let us see an example to make things clear. If our data file is in a predefined format, such as CSV (Comma-Separated Values), we have little choice. (In Chapter 20, we will see how to read CSV in Lua.) However, if we are going to create the file for later use, we can use Lua constructors as our format, instead of CSV. In this format, we represent each data record as a Lua constructor. Instead of writing something like

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

in our data file, we write

```
Entry{"Donald E. Knuth",
      "Literate Programming",
      "CSLI",
      1992}

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

Remember that `Entry{...}` is the same as `Entry({...})`, that is, a call to function `Entry` with a table as its single argument. Therefore, this previous piece of data is a Lua program. To read this 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 (b) 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. (The author's name is the first field in each entry; so, if `b` is an entry value, `b[1]` is the author.)

```
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
```

Notice the event-driven approach in these program fragments: The `Entry` function 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:

```
Entry{
  author = "Donald E. Knuth",
  title = "Literate Programming",
  publisher = "CSLI",
  year = 1992
}

Entry{
  author = "Jon Bentley",
  title = "More Programming Pearls",
  publisher = "Addison-Wesley",
  year = 1990
}
```

(If this format reminds you of BibTeX, it is not a coincidence. BibTeX was one of the inspirations for the constructor syntax in Lua.) 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 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

```
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 only have to change `Entry`:

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

Lua not only runs fast, but it also compiles fast. For instance, the above program for listing authors runs in less than one second for 2 MB of data. Again, this is not by chance. Data description has been one of the main applications of Lua since its creation and we took great care to make its compiler fast for large chunks.

12.1 - 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(o)
  else ...
end
```

For a string value, a naive approach would be something like

```
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. Here, you may be tempted to solve this problem changing quotes:

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

Do not do that! Double square brackets are intended for hand-written strings, not for automatically generated ones. If a malicious user manages to direct your program to save something like `" ]]..os.execute('rm *')..[[ "` (for instance, she can supply that string as her address), your final chunk will be

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

You will have a bad surprise trying to load this "data".

To quote an arbitrary string in a secure way, the `format` function, from the standard `string` library, offers the option `"%q"`. It surrounds the string with double quotes and properly escapes double quotes, newlines, and some other characters inside the string. Using this feature, our `serialize` function now looks like this:

```
function serialize (o)
  if type(o) == "number" then
    io.write(o)
  elseif type(o) == "string" then
    io.write(string.format("%q", o))
  else ...
end
```

12.1.1 - Saving Tables without Cycles

Our next (and harder) task is to save tables. There are several ways to do that, according to what restrictions we assume about the table structure. No single algorithm is appropriate for all cases. Simple tables not only need simpler algorithms, but the resulting files can be more aesthetic, too.

Our first attempt is as follows:

```
function serialize (o)
  if type(o) == "number" then
    io.write(o)
  elseif type(o) == "string" then
    io.write(string.format("%q", o))
  elseif type(o) == "table" then
    io.write("{\n")
    for k,v in pairs(o) do
      io.write("  ", k, " = ")
      serialize(v)
      io.write(",\n")
    end
    io.write("}\n")
  else
```

```
      error("cannot serialize a " .. type(o))
    end
  end
```

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 sub-tables and no cycles). A small aesthetic improvement would be to indent occasional nested tables; you can try it as an exercise. (Hint: Add an extra parameter to `serialize` with the indentation string.)

The previous function assumes that all keys in a table are valid identifiers. If a table has numeric keys, or string keys which are not syntactic valid Lua identifiers, we are in trouble. A simple way to solve this difficulty is to change the line

```
        io.write("  ", k, " = ")
```

to

```
        io.write("  [")
        serialize(k)
        io.write("] = ")
```

With this change, we improve the robustness of our function, at the cost of the aesthetics of the resulting file. Compare:

```
  -- result of serialize{a=12, b='Lua', key='another "one"'}
  -- first version
  {
    a = 12,
    b = "Lua",
    key = "another \"one\"",
  }

  -- second version
  {
    ["a"] = 12,
    ["b"] = "Lua",
    ["key"] = "another \"one\"",
  }
```

We can improve this result by testing for each case whether it needs the square brackets; again, we will leave this improvement as an exercise.

12.1.2 - Saving Tables with Cycles

To handle tables with generic topology (i.e., with cycles and shared sub-tables) we need a different approach. Constructors cannot represent 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 tables as *indices* and their names as the associated values.

We will keep the restriction that the tables we want to save have only strings or numbers as keys. The following function serializes these basic types, returning the result:

```
  function basicSerialize (o)
    if type(o) == "number" then
      return tostring(o)
    else   -- assume it is a string
      return string.format("%q", o)
    end
  end
```

The next function does the hard work. The `saved` parameter is the table that keeps track of tables already saved:

```
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
        local fieldname = string.format("%s[%s]", name,
                                        basicSerialize(k))
        save(fieldname, v, saved)
      end
    end
  else
    error("cannot save a " .. type(value))
  end
end
```

As an example, if we build a table like

```
a = {x=1, y=2; {3,4,5}}
a[2] = a    -- cycle
a.z = a[1]  -- shared sub-table
```

then 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 previous 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` using the same `saved` table. For instance, if we create the following two tables,

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

and save them as follows,

```
save('a', a)
save('b', b)
```

the result will not have common parts:

```
a = {}
a[1] = {}
a[1][1] = "one"
a[1][2] = "two"
a[2] = 3
b = {}
b["k"] = {}
b["k"][1] = "one"
b["k"][2] = "two"
```

However, if we use the same `saved` table for each call to `save`,

```
local t = {}
save('a', a, t)
save('b', b, t)
```

then the result will share common parts:

```
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 handle functions (by building a table that associates each function to its name) etc. Lua gives you the power; you build the mechanisms.

13 - Metatables and Metamethods

Usually, tables in Lua have a quite predictable set of operations. We can add key-value pairs, we can check the value associated with a key, we can traverse all key-value pairs, and that is all. We cannot add tables, we cannot compare tables, and we cannot call a table.

Metatables allow us to change the behavior of a table. 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 that 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.

Each table in Lua may have its own *metatable*. (As we will see later, userdata also can have metatables.) Lua always create new tables without metatables:

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

We can use `setmetatable` to set or change the metatable of any table:

```
t1 = {}
setmetatable(t, t1)
assert(getmetatable(t) == t1)
```

Any table can be the metatable of any other table; a group of related tables may 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.

13.1 - Arithmetic Metamethods

In this section, we will introduce a simple example to explain how to use metatables. Suppose we are using tables to represent sets, with functions to compute the union of two sets, intersection, and the like. As we did with lists, we store these functions inside a table and we define a constructor to create new sets:

```
Set = {}

function Set.new (t)
  local set = {}
  for _, l in ipairs(t) do set[l] = 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
```

To help checking our examples, we also define a function to print sets:

```
function Set.tostring (set)
  local s = "{"
  local sep = ""
  for e in pairs(set) do
    s = s .. sep .. e
    sep = ", "
  end
  return s .. "}"
end

function Set.print (s)
  print(Set.tostring(s))
end
```

Now, we want to make the addition operator (`+´) compute the union of two sets. For that, we will arrange that all tables representing sets share a metatable and this metatable will define how they react to the addition operator. Our first step is to create a regular table that we will use as the metatable for sets. To avoid polluting our namespace, we will store it in the `Set` table:

```
Set.mt = {}    -- metatable for sets
```

The next step is to modify the `Set.new` function, 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 (t)   -- 2nd version
```

```
  local set = {}
  setmetatable(set, Set.mt)
  for _, l in ipairs(t) do set[l] = 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: 00672B60
print(getmetatable(s2))          --> table: 00672B60
```

Finally, we add to the metatable the so-called metamethod, a field `__add` that describes how to perform the union:

```
Set.mt.__add = Set.union
```

Whenever Lua tries to add two sets, it will call this function, with the two operands as arguments.

With the metamethod in place, we can use the addition operator to do set unions:

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

Similarly, we may use the multiplication operator to perform set intersection:

```
Set.mt.__mul = Set.intersection

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

For each arithmetic operator there is a corresponding field name in a metatable. Besides `__add` and `__mul`, there are `__sub` (for subtraction), `__div` (for division), `__unm` (for negation), and `__pow` (for exponentiation). We may define also the field `__concat`, to define a behavior for the concatenation operator.

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
```

To choose a metamethod, Lua does the following: (1) If the first value has a metatable with an `__add` field, Lua uses this value as the metamethod, independently of the second value; (2) otherwise, if the second value has a metatable with an `__add` field, Lua uses this value as the metamethod; (3) otherwise, Lua raises an error. Therefore, the last example will call `Set.union`, as will the expressions `10 + s` and `"hy" + s`.

Lua does not care about those mixed types, but our implementation does. If we run the `s = s + 8` example, the error we get will be inside `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:

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

```
    ...  -- same as before
```

13.2 - Relational Metamethods

Metatables also allow us to give meaning to the relational operators, through the metamethods `__eq` (*equality*), `__lt` (*less than*), and `__le` (*less or equal*). There are no separate metamethods for the other three relational operators, as Lua translates `a ~= b` to `not (a == b)`, `a > b` to `b < a`, and `a >= b` to `b <= a`.

(Big parentheses: Until Lua 4.0, all order operators were translated 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, floating-point numbers are not totally ordered in most machines, because of the value *Not a Number* (*NaN*). According to the IEEE 754 standard, currently adopted by virtually every hardware, NaN represents undefined values, such as the result of `0/0`. The standard specifies that any comparison that involves NaN should result in false. That means that `NaN <= x` is always false, but `x < NaN` is also false. That implies 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 that meaning, again it is possible that both `a <= b` and `b < a` are false; therefore, we need separate implementations for `__le` (*less or equal*) and `__lt` (*less than*):

```
    Set.mt.__le = function (a,b)    -- set containment
      for k in pairs(a) do
        if not b[k] then return false end
      end
      return true
    end

    Set.mt.__lt = function (a,b)
      return a <= b and not (b <= a)
    end
```

Finally, we can define set equality through set containment:

```
    Set.mt.__eq = function (a,b)
      return a <= b and b <= a
    end
```

After those definitions, we are now 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
```

Unlike arithmetic metamethods, relational metamethods do not support mixed types. Their behavior for mixed types mimics the common behavior of these operators in Lua. If you try to compare a string with a number for order, Lua raises an error. Similarly, if you try to compare two objects with different metamethods for order, Lua raises an error.

An equality comparison never raises an error, but if two objects have different metamethods, the equality operation results in false, without even calling any metamethod. Again, this behavior mimics the common behavior of Lua, which always classifies strings as different from numbers, regardless of their values. Lua calls the equality metamethod only when the two objects being compared share this metamethod.

13.3 - Library-Defined Metamethods

It is a common practice for some libraries to define their own fields in metatables. So far, all the metamethods we have seen are for the Lua core. It is the virtual machine that detects that the values involved in an operation have metatables and that these metatables define metamethods for that operation. However, because the metatable is a regular table, anyone can use it.

The `tostring` function provides a typical example. As we saw earlier, `tostring` represents tables in a rather simple format:

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

(Note that `print` always calls `tostring` to format its output.) However, when formatting an object, `tostring` first checks whether the object has a metatable with a `__tostring` field. If this is the case, `tostring` calls the corresponding value (which must be a function) 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 set metatable:

```
    Set.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 `setmetatable/getmetatable` functions use a metafield also, in this case to protect metatables. Suppose you want to protect your sets, so that users can neither see nor change their metatables. If you set a `__metatable` field in the metatable, `getmetatable` will return the value of this field, whereas `setmetatable` will raise an error:

```
    Set.mt.__metatable = "not your business"

    s1 = Set.new{}
    print(getmetatable(s1))     --> not your business
    setmetatable(s1, {})
      stdin:1: cannot change protected metatable
```

13.4 - Table-Access Metamethods

The metamethods for arithmetic and relational operators all define behavior for otherwise erroneous situations. They do not change the normal behavior of the language. But Lua also offers a way to change the behavior of tables for two normal situations, the query and modification of absent fields in a table.

13.4.1 - The `__index` Metamethod

I said 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 access triggers 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 and a constructor function, which creates new windows sharing a metatable:

```
-- create a namespace
Window = {}
-- create the prototype with default values
Window.prototype = {x=0, y=0, width=100, height=100, }
-- create a metatable
Window.mt = {}
-- declare the constructor function
function Window.new (o)
  setmetatable(o, Window.mt)
  return o
end
```

Now, we define the `__index` metamethod:

```
Window.mt.__index = function (table, key)
  return Window.prototype[key]
end
```

After that code, we create a new window and query it for an absent field:

```
w = Window.new{x=10, y=20}
print(w.width)    --> 100
```

When Lua detects that `w` does not have the requested field, but has a metatable with an `__index` field, 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 the name, 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. When it is a table, Lua redoes the access in that table. Therefore, in our previous example, we could declare `__index` simply as

```
Window.mt.__index = Window.prototype
```

Now, when Lua looks for the metatable's `__index` field, it finds the value of `Window.prototype`, which is a table. Consequently, Lua repeats the access in this table, that is, it executes the equivalent of

```
Window.prototype["width"]
```

which gives the desired result.

The use of a table as an `__index` metamethod provides a cheap 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 those forms of inheritance in Chapter 16.

When we want to access a table without invoking its `__index` metamethod, we use the `rawget` function. The call `rawget(t,i)` does a *raw* access to table `t`. Doing a raw access will not speed up your code (the overhead of a function call kills any gain you could have), but sometimes you need it, as we will see later.

13.4.2 - The `__newindex` Metamethod

The `__newindex` metamethod does for table updates what `__index` does for table accesses. When you 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 that table, instead of in the original one. Moreover, there is a raw function that allows you to bypass the metamethod: The call `rawset(t, k, v)` sets the value `v` in key `k` of table `t` without invoking any metamethod.

The combined use of `__index` and `__newindex` metamethods allows several powerful constructs in Lua, from read-only tables to tables with default values to inheritance for object-oriented programming. In the rest of this chapter we see some of these uses. Object-oriented programming has its own chapter.

13.4.3 - 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
```

Now, whenever we access an absent field in `tab`, its `__index` metamethod is called and returns zero, which is the value of `d` for that metamethod.

The `setDefault` function creates a new metatable for each table that needs a default value. This may be expensive if we have many tables that need default values. However, the metatable has the default value `d` wired into itself, so the function cannot use a single metatable for all tables. To allow the use of a single metatable for tables with different default values, 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
```

If we are worried about name clashes, it is easy to ensure the uniqueness of this special key. All we need is to create a new table and use it 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 to associating each table with its default value is to use 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 17.

Another alternative is to *memoize* 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 17.

13.4.4 - Tracking Table Accesses

Both `__index` and `__newindex` are relevant only when the index does not exist in the table. The only way to catch all accesses to a table is to keep it empty. So, 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 `__index` and `__newindex` metamethods, which track all accesses and redirect them to the original table. Suppose that `t` is the original table we want to track. We can write something like this:

```
t = {}   -- original table (created somewhere)

-- keep a private access to original table
local _t = t

-- create proxy
t = {}

-- create metatable
local mt = {
  __index = function (t,k)
    print("*access to element " .. tostring(k))
    return _t[k]   -- access the original table
  end,

  __newindex = function (t,k,v)
    print("*update of element " .. tostring(k) ..
                         " to " .. tostring(v))
    _t[k] = v   -- update original table
  end
}
setmetatable(t, mt)
```

This code tracks every access to `t`:

```
> t[2] = 'hello'
*update of element 2 to hello
> print(t[2])
*access to element 2
hello
```

(Notice that, unfortunately, this scheme does not allow us to traverse tables. The `pairs` function will operate on the proxy, not on the original table.)

If we want to monitor several tables, we do not need a different metatable for each one. Instead, we can somehow associate each proxy to its original table and share a common metatable for all proxies. A simple way to associate proxies to tables is to keep the original table in a proxy's field, as long as we can be sure that this field will not be used for other means. A simple way to ensure that is to create a private key that nobody else can access. Putting these ideas together results in the following code:

```
-- create private index
local index = {}

-- create metatable
local mt = {
  __index = function (t,k)
    print("*access to element " .. tostring(k))
    return t[index][k]   -- access the original table
  end,

  __newindex = function (t,k,v)
    print("*update of element " .. tostring(k) ..
                         " to " .. tostring(v))
    t[index][k] = v   -- update original table
  end
}

function track (t)
  local proxy = {}
  proxy[index] = t
  setmetatable(proxy, mt)
  return proxy
end
```

Now, whenever we want to monitor a table `t`, all we have to do is `t = track(t)`.

13.4.5 - 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 quite more efficient to redirect all queries to the original table. This use, however, 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
```

(Remember that the second argument to `error`, 2, directs the error message to where the update was attempted.) 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
```

14 - The Environment

Lua keeps all its global variables in a regular table, called the *environment*. (To be more precise, Lua keeps its "global" variables in several environments, but we will ignore this multiplicity for a while.) One advantage of this structure is that it simplifies the internal implementation of Lua, because there is no need for a different data structure for global variables. The other (actually the main) advantage is that we can manipulate this table as any other table. To facilitate such manipulations, Lua stores the environment itself in a global variable `_G`. (Yes, `_G._G` is equal to `_G`.) For instance, the following code prints the names of all global variables defined in the current environment:

```
for n in pairs(_G) do print(n) end
```

In this chapter, we will see several useful techniques to manipulate the environment.

14.1 - Accessing Global Variables with Dynamic Names

Usually, assignment is enough for getting and setting global variables. However, often 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 somehow computed at run time. To get the value of this variable, many programmers are tempted to write something like

```
loadstring("value = " .. varname)()
```

or

```
value = loadstring("return " .. varname)()
```

If `varname` is `x`, for instance, the concatenation will result in `"return x"` (or `"value = x"`, with the first form), which when run achieves the desired result. However, such codes involve the creation and compilation of a new chunk and lots of extra work. You 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, you can simply index it with the desired key (the variable name).

In a similar way, you can assign to a global variable whose name is computed dynamically, writing `_G[varname] = value`. Beware, however: Some programmers get a little excited with these functions and end up writing code like `_G["a"] = _G["var1"]`, which is just a complicated way to write `a = var1`.

A generalization of the previous problem is to allow fields in a dynamic name, such as `"io.read"` or `"a.b.c.d"`. We solve this problem with 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.gfind(f, "[%w_]+") do
    v = v[w]
  end
  return v
end
```

We rely on `gfind`, from the `string` library, to iterate over all words in `f` (where "word" is a sequence of one or more alphanumeric characters and underscores).

The corresponding function to set fields is a little more complex. An assignment like

```
a.b.c.d.e = v
```

is exactly equivalent to

```
local temp = a.b.c.d
temp.e = v
```

That is, we must retrieve up to the last name; we must handle the last field separately. The new `setfield` function also creates intermediate tables in a path when they do not exist:

```
function setfield (f, v)
  local t = _G    -- start with the table of globals
  for w, d in string.gfind(f, "([%w_]+)(.?)") do
    if d == "." then      -- not last field?
      t[w] = t[w] or {}   -- create table if absent
      t = t[w]            -- get the table
    else                  -- last field
      t[w] = v            -- do the assignment
    end
  end
end
```

This new pattern captures the field name in variable `w` and an optional following dot in variable `d`. If a field name is not followed by a dot then it is the last name. (We will discuss pattern matching at great length in Chapter 20.)

With the previous functions, the call

```
setfield("t.x.y", 10)
```

creates a global table `t`, another table `t.x`, and assigns 10 to `t.x.y`:

```
print(t.x.y)     --> 10
print(getfield("t.x.y"))   --> 10
```

14.2 - Declaring Global Variables

Global variables in Lua do not need declarations. Although this is handy for small programs, in larger programs a simple typo can cause bugs that are difficult to find. However, we can change that behavior if we like. Because Lua keeps its global variables in a regular table, we can use metatables to change its behavior when accessing global variables.

A first approach is as follows:

```
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 that code, any attempt to access a non-existent global variable will trigger an error:

```
> a = 1
stdin:1: attempt to write to undeclared variable a
```

But how do we declare new variables? With `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**. Notice that you should define this function before installing the access control, otherwise you get an error: After all, you are trying to create a new global, `declare`. With that function in place, you have complete control over your global variables:

```
> a = 1
stdin:1: attempt to write to undeclared variable a
> declare"a"
> a = 1           -- OK
```

But now, to test whether a variable exists, we cannot simply compare it to **nil**; if it is **nil**, the access will throw an error. Instead, we use `rawget`, which avoids the metamethod:

```
if rawget(_G, var) == nil then
  -- `var' is undeclared
  ...
end
```

It is not difficult to change that control to allow global variables with **nil** value. All we need is an auxiliary table that keeps the names of declared variables. Whenever a metamethod is called, it checks in that table whether the variable is undeclared or not. The code may be like this:

```
local declaredNames = {}
function declare (name, initval)
  rawset(_G, name, initval)
  declaredNames[name] = true
end
setmetatable(_G, {
  __newindex = function (t, n, v)
    if not declaredNames[n] then
      error("attempt to write to undeclared var. "..n, 2)
    else
      rawset(t, n, v)   -- do the actual set
    end
  end,
  __index = function (_, n)
    if not declaredNames[n] then
      error("attempt to read undeclared var. "..n, 2)
    else
      return nil
    end
  end,
})
```

For both solutions, the overhead is negligible. With the first solution, the metamethods are never called during normal operation. In the second, they may be called, but only when the program accesses a variable holding a **nil**.

14.3 - Non-Global Environments

One of the problems with the environment is that it is global. Any modification you do on it affects all parts of your program. For instance, when you install a metatable to control global access, your whole program must follow the guidelines. If you want to use a library that uses global variables without declaring them, you are in bad luck.

Lua 5.0 ameliorates this problem by allowing each function to have its own environment. That may sound strange at first; after all, the goal of a table of global variables is to be global. However, in Section 15.4 we will see that this facility allows several interesting constructions, where global values are still available everywhere.

You can change the environment of a function with the `setfenv` function (*set function environment*). It receives the function and the new environment. Instead of the function itself, you can also give a number, meaning the active function at that given stack level. Number 1 means the current function, number 2 means the function calling the current function (which is handy to write auxiliary functions that change the environment of their caller), and so on.

A naive first attempt to use `setfenv` fails miserably. The code

```
a = 1   -- create a global variable
-- change current environment to a new empty table
setfenv(1, {})
print(a)
```

results in

```
stdin:5: attempt to call global `print' (a nil value)
```

(You must run that code in a single chunk. If you enter it line by line in interactive mode, each line is a different function and the call to `setfenv` only affects its own line.) Once you change your environment, all global accesses will use this new table. If it is empty, you lost all your global variables, even `_G`. So, you should first populate it with some useful values, such as the old environment:

```
a = 1   -- create a global variable
-- change current environment
setfenv(1, {_G = _G})
_G.print(a)      --> nil
_G.print(_G.a)   --> 1
```

Now, when you access the "global" `_G`, its value is the old environment, wherein you will find the field `print`.

You can populate your new environment using inheritance also:

```
a = 1
local newgt = {}        -- create new environment
setmetatable(newgt, {__index = _G})
setfenv(1, newgt)    -- set it
print(a)          --> 1
```

In this code, the new environment inherits both `print` and `a` from the old one. Nevertheless, any assignment goes to the new table. There is no danger of changing a really global variable by mistake, although you still can change them through `_G`:

```
-- continuing previous code
```

```
a = 10
print(a)       --> 10
print(_G.a)    --> 1
_G.a = 20
print(_G.a)    --> 20
```

When you create a new function, it inherits its environment from the function creating it. Therefore, if a chunk changes its own environment, all functions it defines afterward will share this same environment. This is a useful mechanism for creating namespaces, as we will see in the next chapter.

15 - Packages

Many languages provide mechanisms to organize their space of global names, such as *modules* in Modula, *packages* in Java and Perl, or *namespaces* in C++. Each of these mechanisms has different rules regarding the use of elements declared inside a package, visibility, and other details. Nevertheless, all of them provide a basic mechanism to avoid collision among names defined in different libraries. Each library creates its own namespace and names defined inside this namespace do not interfere with names in other namespaces.

Lua does not provide any explicit mechanism for packages. However, we can implement them easily with the basic mechanisms that the language provides. The main idea is to represent each package by a table, as the basic libraries do.

An obvious benefit of using tables to implement packages is that we can manipulate packages like any other table and use the whole power of Lua to create extra facilities. In most languages, packages are not first-class values (that is, they cannot be stored in variables, passed as arguments to functions, etc.), so these languages need special mechanisms for each extra trick you may do with a package.

In Lua, although we always represent packages as tables, there are several different methods to write a package. In this chapter, we cover some of these methods.

15.1 - The Basic Approach

A simple way to define a package is to write the package name as a prefix for each object in the package. For instance, suppose we are writing a library to manipulate complex numbers. We represent each complex number as a table, with fields `r` (real part) and `i` (imaginary part). We declare all our new operations in another table, which acts as a new package:

```
complex = {}

function complex.new (r, i) return {r=r, i=i} end

-- defines a constant `i'
complex.i = complex.new(0, 1)

function complex.add (c1, c2)
  return complex.new(c1.r + c2.r, c1.i + c2.i)
end

function complex.sub (c1, c2)
  return complex.new(c1.r - c2.r, c1.i - c2.i)
end
```

```
function complex.mul (c1, c2)
  return complex.new(c1.r*c2.r - c1.i*c2.i,
                     c1.r*c2.i + c1.i*c2.r)
end

function complex.inv (c)
  local n = c.r^2 + c.i^2
  return complex.new(c.r/n, -c.i/n)
end

return complex
```

This library defines one single global name, `complex`. All other definitions go inside this table.

With this definition, we can use any complex operation qualifying the operation name, like this:

```
c = complex.add(complex.i, complex.new(10, 20))
```

This use of tables for packages does not provide exactly the same functionality as provided by real packages. First, we must explicitly put the package name in every function definition. Second, a function that calls another function inside the same package must qualify the name of the called function. We can ameliorate those problems using a fixed local name for the package (`P`, for instance), and then assigning this local to the final name of the package. Following this guideline, we would write our previous definition like this:

```
local P = {}
complex = P           -- package name

P.i = {r=0, i=1}
function P.new (r, i) return {r=r, i=i} end

function P.add (c1, c2)
  return P.new(c1.r + c2.r, c1.i + c2.i)
end

   ...
```

Whenever a function calls another function inside the same package (or whenever it calls itself recursively), it still needs to prefix the name. At least, the connection between the two functions does not depend on the package name anymore. Moreover, there is only one place in the whole package where we write the package name.

Maybe you noticed that the last statement in the package was

```
return complex
```

This return is not necessary, because the package is already assigned to a global variable (`complex`). Nevertheless, we consider a good practice that a package returns itself when it opens. The extra return costs nothing, and allows alternative ways to handle the package.

15.2 - Privacy

Sometimes, a package *exports* all its names; that is, any client of the package can use them. Usually, however, it is useful to have private names in a package, that is, names that only the package itself can use. A convenient way to do that in Lua is to define those private names as local variables. For instance, let us add to our example a private function that checks whether a value is a valid complex number. Our example now looks like this:

```
local P = {}
complex = P

local function checkComplex (c)
  if not ((type(c) == "table") and
     tonumber(c.r) and tonumber(c.i)) then
    error("bad complex number", 3)
  end
end

function P.add (c1, c2)
  checkComplex(c1);
  checkComplex(c2);
  return P.new(c1.r + c2.r, c1.i + c2.i)
end

  ...

return P
```

What are the pros and cons of this approach? All names in a package live in a separate namespace. Each entity in a package is clearly marked as public or private. Moreover, we have real privacy: Private entities are inaccessible outside the package. A drawback of this approach is its verbosity when accessing other public entities inside the same package, as every access still needs the prefix `P`. A bigger problem is that we have to change the calls whenever we change the status of a function from private to public (or from public to private).

There is an interesting solution to both problems at once. We can declare all functions in our package as local and later put them in the final table to be exported. Following this approach, our `complex` package would be like this:

```
local function checkComplex (c)
  if not ((type(c) == "table")
     and tonumber(c.r) and tonumber(c.i)) then
    error("bad complex number", 3)
  end
end

local function new (r, i) return {r=r, i=i} end

local function add (c1, c2)
  checkComplex(c1);
  checkComplex(c2);
  return new(c1.r + c2.r, c1.i + c2.i)
end

  ...

complex = {
  new = new,
  add = add,
  sub = sub,
  mul = mul,
  div = div,
}
```

Now we do not need to prefix any calls, so that calls to exported and private functions are equal. There is a simple list at the end of the package that defines explicitly which names to export. Most people find more natural to have this list at the beginning of the package, but we cannot put the list at the top, because we must define the local functions first.

15.3 - Packages and Files

Typically, when we write a package, we put all its code in a single file. Then, to open or import a package (that is, to make it available) we just execute that file. For instance, if we have a file `complex.lua` with the definition of our complex package, the command `require "complex"` will open the package. Remember that `require` avoids loading the same package multiple times.

A recurring issue is the relationship between the file name and the package name. Of course, it is a good idea to relate them, because `require` works with files, not with packages. One solution is to name the file after the package, followed by some known extension. Lua does not fix any extension; it is up to your path to do that. For instance, if your path includes a component like `"/usr/local/lualibs/?.lua"`, than the package `complex` may live in a `complex.lua` file.

Some people prefer the reverse, to name the package after the file name, dynamically. That is, if you rename the file, the package is renamed, too. This solution gives you more flexibility. For instance, if you get two different packages with the same name, you do not have to change any of them, just rename one file. To implement this naming scheme in Lua, we use the `_REQUIREDNAME` variable. Remember that, when `require` loads a file, it defines that variable with the virtual file name. So, you can write something like the following in your package:

```
local P = {}   -- package
if _REQUIREDNAME == nil then
  complex = P
else
  _G[_REQUIREDNAME] = P
end
```

The test allows us to use the package without `require`. If `_REQUIREDNAME` is not defined, we use a fixed name for the package (`complex`, in the example). Otherwise, the package registers itself with the virtual file name, whatever it is. If a user puts the library in file `cpx.lua` and runs `require"cpx"`, the package loads itself in table `cpx`. If another user moves the library to file `cpx_v1.lua` and runs `require"cpx_v1"`, the package loads itself in table `cpx_v1`.

15.4 - Using the Global Table

One drawback of all these methods to create packages is that they call for special attention from the programmer. It is all too easy to forget a **local** in a declaration, for instance. Metamethods in the table of global variables offer some interesting alternative techniques for creating packages. The common part in all these techniques is the use of an exclusive environment for the package. This is easily done: If we change the environment of the package's main chunk, all functions it creates will share this new environment.

The simplest technique does little more than that. Once the package has an exclusive environment, not only all its functions share this table, but also all its global variables go to this table. Therefore, we can declare all public functions as global variables and they will go to a separate table automatically. All the package has to do is to register this table as the package name. The next code fragment illustrates this technique for the `complex` library:

```
local P = {}
complex = P
```

```
setfenv(1, P)
```

Now, when we declare function `add`, it goes to `complex.add`:

```
function add (c1, c2)
  return new(c1.r + c2.r, c1.i + c2.i)
end
```

Moreover, we can call other functions from this package without any prefix. For instance, `add` gets `new` from its environment, that is, it gets `complex.new`.

This method offers a good support for packages, with little extra work on the programmer. It needs no prefixes at all. There is no difference between calling an exported and a private function. If the programmer forgets a `local`, she does not pollute the global namespace; instead, only a private function becomes public. Moreover, we can use it together with the techniques from the previous section for package names:

```
local P = {}   -- package
if _REQUIREDNAME == nil then
  complex = P
else
  _G[_REQUIREDNAME] = P
end
setfenv(1, P)
```

What is missing, of course, is access to other packages. Once we make the empty table `P` our environment, we lose access to all previous global variables. There are several solutions to this, each with its pros and cons.

The simplest solution is inheritance, as we saw earlier:

```
local P = {}   -- package
setmetatable(P, {__index = _G})
setfenv(1, P)
```

(You must call `setmetatable` before calling `setfenv`; can you tell why?) With this construction, the package has direct access to any global identifier, but it pays a small overhead for each access. A funny consequence of this solution is that, conceptually, your package now contains all global variables. For instance, someone using your package may call the standard sine function writing `complex.math.sin(x)`. (Perl's package system has this peculiarity, too.)

Another quick method of accessing other packages is to declare a local that holds the old environment:

```
local P = {}
pack = P
local _G = _G
setfenv(1, P)
```

Now you must prefix any access to external names with `_G.`, but you get faster access, because there is no metamethod involved. Unlike inheritance, this method gives you write access to the old environment; whether this is good or bad is debatable, but sometimes you may need this flexibility.

A more disciplined approach is to declare as locals only the functions you need, or at most the packages you need:

```
local P = {}
pack = P

-- Import Section:
-- declare everything this package needs from outside
```

```
local sqrt = math.sqrt
local io = io

-- no more external access after this point
setfenv(1, P)
```

This technique demands more work, but it documents your package dependencies better. It also results in faster code than the previous schemes.

15.5 - Other Facilities

As I said earlier, the use of tables to implement packages allows us to use the whole power of Lua to manipulate them. There are unlimited possibilities. Here I will give only a few suggestions.

We do not need to define all public items of a package together. For instance, we can add a new item to our `complex` package in a separate chunk:

```
function complex.div (c1, c2)
  return complex.mul(c1, complex.inv(c2))
end
```

(But notice that the private part is restricted to one file, which I think is a good thing.) Conversely, we can define more than one package in the same file. All we have to do is to enclose each one inside a **do** block, so that its local variables are restricted to that block.

Outside the package, if we are going to use some operations often, we can give them local names:

```
local add, i = complex.add, complex.i

c1 = add(complex.new(10, 20), i)
```

Or else, if we do not want to write the package name over and over, we can give a shorter local name to the package itself:

```
local C = complex
c1 = C.add(C.new(10, 20), C.i)
```

It is easy to write a function that unpacks a package, putting all its names into the global namespace:

```
function openpackage (ns)
  for n,v in pairs(ns) do _G[n] = v end
end

openpackage(complex)
c1 = mul(new(10, 20), i)
```

If you are afraid of name clashes when opening a package, you can check the name before the assignment:

```
function openpackage (ns)
  for n,v in pairs(ns) do
    if _G[n] ~= nil then
      error("name clash: " .. n .. " is already defined")
    end
    _G[n] = v
  end
end
```

Because packages themselves are tables, we can even nest packages; that is, we can create a package inside another one. However, this facility is seldom necessary.

Another interesting facility is *autoload*, which only loads a function if the function is actually used by the program. When we load an autoload package, it creates an empty table to represent the package and sets the `__index` metamethod of the table to do the autoload. Then, when we call any function that is not yet loaded, the `__index` metamethod is invoked to load it. Subsequent calls find the function already loaded; therefore, they do not activate the metamethod.

A simple way to implement autoload can be as follows. Each function is defined in an auxiliary file. (There can be more than one function in each file.) Each of these files defines its functions in a standard way, for instance like here:

```
function pack1.foo ()
  ...
end

function pack1.goo ()
  ...
end
```

However, the file does not create the package, because the package already exists when the function is loaded.

In the main package we define an auxiliary table that describes where we can find each function:

```
local location = {
  foo = "/usr/local/lua/lib/pack1_1.lua",
  goo = "/usr/local/lua/lib/pack1_1.lua",
  foo1 = "/usr/local/lua/lib/pack1_2.lua",
  goo1 = "/usr/local/lua/lib/pack1_3.lua",
}
```

Then we create the package and define its metamethod:

```
pack1 = {}

setmetatable(pack1, {__index = function (t, funcname)
  local file = location[funcname]
  if not file then
    error("package pack1 does not define " .. funcname)
  end
  assert(loadfile(file))()      -- load and run definition
  return t[funcname]            -- return the function
end})

return pack1
```

After loading this package, the first time the program executes `pack1.foo()` it will invoke that `__index` metamethod, which is quite simple. It checks that the function has a corresponding file and loads that file. The only subtlety is that it must not only load the file, but also return the function as the result of the access.

Because the entire system is written in Lua, it is easy to change its behavior. For instance, the functions may be defined in C, with the metamethod using `loadlib` to load them. Or we can set a metamethod in the global table to autoload entire packages. The possibilities are endless.

16 - 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 *selfness*) 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, but it is always the same object. 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:

```
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 `Account` object. Then, we can call it as

```
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 bad 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 name of this object, `withdraw` does not work any more:

```
a = Account; Account = nil
a.withdraw(100.00)    -- ERROR!
```

Such behavior violates the previous principle that objects have independent life cycles.

A more flexible approach is to operate on the *receiver* of the operation. For that, we would have to define our method with an extra parameter, which tells the method on which object it has to operate. 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 on which object it has to operate:

```
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 partly 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 can also hide this parameter, using the *colon operator*. We can rewrite the previous method definition as

```
function Account:withdraw (v)
  self.balance = self.balance - v
end
```

and the method call as

```
a:withdraw(100.00)
```

The effect of the colon is to add an extra hidden parameter in a method definition and to add an extra argument in a method call. 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)
```

Now our objects have an identity, a state, and operations over 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?

16.1 - Classes

A *class* works as a mold for the creation of objects. Several OO languages offer the concept of class. In such languages, each object is an instance of a specific class. Lua does not have the concept of class; each object defines its own behavior and has a shape of its own. Nevertheless, it is not difficult to emulate classes in Lua, following the lead from prototype-based languages, such as Self and NewtonScript. In those 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, it is trivial to implement prototypes, using the idea of inheritance that we saw in the previous chapter. More specifically, if we have two objects `a` and `b`, all we have to do to make `b` a prototype for `a` is

```
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 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. Note a small optimization, that we do not need to create an extra table to be the metatable of the account objects; we can use the `Account` table itself for that purpose:

```
function Account:new (o)
  o = o or {}   -- create object if user does not provide one
  setmetatable(o, self)
  self.__index = self
  return o
```

```
    end
```

(When we call `Account:new`, `self` is equal to `Account`; so we could have used `Account` directly, instead of `self`. However, the use of `self` will fit nicely when we introduce class inheritance, in the next section.) After that code, what happens when we create a new account and call a method on it?

```
    a = Account:new{balance = 0}
    a:deposit(100.00)
```

When we create this new account, `a` will have `Account` (the *self* in the call `Account:new`) as its metatable. Then, 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 table `a`; so, it looks into the metatable's `__index` entry. The situation now is more or less like this:

```
    getmetatable(a).__index.deposit(a, 100.00)
```

The metatable of `a` is `Account` and `Account.__index` is also `Account` (because the new method did `self.__index = self`). Therefore, we can rewrite the previous expression as

```
    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 `deposit` function from `Account`. By the same mechanism, it can inherit all fields from `Account`.

The inheritance works not only for methods, but also for other fields that are absent in the new account. Therefore, a class provides not only methods, but also 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

```
    b.balance = b.balance + v
```

(because `self` is `b`). The expression `b.balance` evaluates to zero and an initial deposit is assigned to `b.balance`. The next time we ask for this value, the index metamethod is not invoked (because now `b` has its own `balance` field).

16.2 - Inheritance

Because classes are objects, they can get methods from other classes, too. That makes inheritance (in the usual object-oriented meaning) quite easy to implement in Lua.

Let us assume we have a base class like `Account`:

```
    Account = {balance = 0}

    function Account:new (o)
      o = o or {}
      setmetatable(o, self)
      self.__index = self
      return o
    end
```

```
function Account:deposit (v)
  self.balance = self.balance + v
end

function Account:withdraw (v)
  if v > self.balance then error"insufficient funds" end
  self.balance = self.balance - v
end
```

From that class, we want to derive a subclass `SpecialAccount`, which 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 nice thing happens now:

```
s = SpecialAccount:new{limit=1000.00}
```

`SpecialAccount` inherits `new` from `Account` like any other method. This time, however, when `new` executes, the `self` parameter will refer to `SpecialAccount`. Therefore, the metatable of `s` will be `SpecialAccount`, whose value at index __`index` is also `SpecialAccount`. So, `s` inherits from `SpecialAccount`, which inherits from `Account`. 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` and there it finds the original implementation for a deposit.

What makes a `SpecialAccount` special is that it 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. Because `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 OO in Lua is that you do not need to create a new class to specify a new behavior. If only a single object needs a specific behavior, you can implement that directly in the object. For instance, if the account `s` represents some special client whose limit is always 10% of her balance, you can modify only this single account:

```
function s:getLimit ()
  return self.balance * 0.10
end
```

After that declaration, the call `s:withdraw(200.00)` runs the `withdraw` method from

`SpecialAccount`, but when that method calls `self:getLimit`, it is this last definition that it invokes.

16.3 - Multiple Inheritance

Because objects are not primitive in Lua, there are several ways to do object-oriented programming in Lua. The method we saw previously, using the index metamethod, is probably the best combination of simplicity, performance, and flexibility. Nevertheless, there are other implementations, which may be more appropriate to some particular cases. Here we will see an alternative implementation that allows multiple inheritance in Lua.

The key for this implementation is the use of a function for the metafield `__index`. Remember that, when a table's metatable has a function in the field `__index`, Lua will call that 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 may have more than one superclass. Therefore, we cannot use a class method to create subclasses. Instead, we will define a specific function for that purpose, `createClass`, which has as arguments the superclasses of the new class. 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 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 classes and instances. Particularly, a class cannot be the metatable for its instances and its own metatable at the same time. In the following implementation, we keep a class as the metatable for its instances and create another table to be the class' metatable.

```
-- look up for `k' in list of tables `plist'
local function search (k, plist)
  for i=1, table.getn(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

  -- class will search for each method in the list of its
  -- parents (`arg' is the list of parents)
  setmetatable(c, {__index = function (t, k)
    return search(k, arg)
  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 new class
  return c
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 what happens in the last statement. Lua cannot find the field `"getname"` in `account`. So, it looks for the field `__index` of `account`'s metatable, which is `NamedAccount`. 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 into `Account`, without success, and then into `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, arg)
    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 system is running, because these changes do not propagate down the hierarchy chain.

16.4 - Privacy

Many people consider privacy to be an integral part of an object-oriented language; the state of each object should be its own internal affair. In some OO languages, such as C++ and Java, you can control whether an object field (also called an *instance variable*) or a method is visible outside the object. Other languages, such as Smalltalk, make all variables private and all methods public. The first OO language, Simula, did not offer any kind of protection.

The main design for 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. But this also reflects some basic design decisions behind Lua. Lua is not intended for building huge programs, where many programmers are involved for long periods. Quite the opposite, Lua aims at small to medium programs, usually part of a larger system, typically developed by one or a few programmers, or even by non programmers. Therefore, Lua avoids too much redundancy and artificial restrictions. If you do not want to access something inside an object, just *do not do it*.

Nevertheless, another aim of Lua is to be flexible, offering to the programmer meta-mechanisms through which she can 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, so as to have access control. Although this implementation is not used frequently, it is instructive to know about it, both because it explores some interesting corners of Lua and because it can be a good solution for other problems.

The basic idea of this alternative design is to represent each object through two tables: one for its state; another for its operations, or its *interface*. The object itself is accessed through the second table, that is, through the operations that compose its interface. To avoid unauthorized access, the table that represents 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 closures (that is, instances of nested functions) for each of 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 those 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. The methods are called just like any other function:

```
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 `newAccount` returns, there is no way to gain direct access to that table. We can only access it 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 that 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 this computation. We can implement this 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 + self.extra()
  end

  ...
```

Again, there is no way for any user to access the `extra` function directly.

16.5 - The Single-Method Approach

A particular case of the previous approach for OO 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 Section 7.1, where we saw how to construct iterator functions that keep state as closures. An iterator that keeps state 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 possible implementation for such 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 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.

17 - Weak Tables

Lua does automatic memory management. A program only creates objects (tables, functions, etc.); there is no function to delete objects. Lua automatically deletes objects that become garbage, using *garbage collection*. That frees you from most of the burden of memory management and, more important, frees you from most of the bugs related to that activity, such as dangling pointers and memory leaks.

Unlike some other collectors, Lua's garbage collector has no problems with cycles. You do not need to take any special action when using cyclic data structures; they are collected like any other data. Nevertheless, sometimes even the smarter collector needs your help. No garbage collector allows you to forget all worries about memory management.

A garbage collector can collect only what it can be sure is garbage; it cannot know what you consider garbage. A typical example is a stack, implemented with an array and an index to the top. You know that the valid part of the array goes only up to the top, but Lua does not. If you pop an element by simply decrementing the top, the object left in the array is not garbage for Lua. Similarly, any object stored in a global variable is not garbage for Lua, even if your program will never use it again. In both cases, it is up to you (i.e., your program) to assign **nil** to these positions so that they do not lock an otherwise free object.

However, simply cleaning your references is not always enough. Some constructions need extra collaboration between you and the collector. A typical example happens when you want to keep a collection of all live objects of some kind (e.g., files) in your program. That seems a simple task: All you have to do is to insert each new object into the collection. However, once the object is inside the collection, it will never be collected! Even if no one else points to it, the collection does. Lua cannot know that this reference should not prevent the reclamation of the object, unless you tell Lua about that.

Weak tables are the mechanism that you 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 object is collected and somehow these weak references are deleted. Lua implements weak references as weak tables: A *weak table* is a table where all references are weak. That means that, if an object is only held inside weak tables, Lua will collect the object eventually.

Tables have keys and values and both may 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 to which they refer. In a weak table, keys and values may be weak. That means that there are three kinds of weak tables: tables with weak keys, tables with weak values, and fully weak tables, where both keys and values are weak. Irrespective of the table kind, 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 the string contains the letter `` `k´ `` (lower case), the keys in the table are weak; if the string contains the letter `` `v´ `` (lower case), the values in the table are weak. The following example, although artificial, illustrates the basic behavior of weak tables:

```
    a = {}
    b = {}
```

```
setmetatable(a, b)
b.__mode = "k"         -- 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 first key. When the collector runs, there is no other reference to the first key, so it is collected and the corresponding entry in the table is removed. The second key, however, is still anchored in variable `key`, so it is not collected.

Notice that only objects can be collected from a weak table. Values, such as numbers and booleans, are not collectible. For instance, if we insert a numeric key in table `a` (from our previous example), it will never be removed by the collector. Of course, if the value corresponding to a numeric key is collected, then the whole entry is removed from the weak 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 functions, are created explicitly. For instance, whenever Lua evaluates `{}`, it creates a new table. Whenever it evaluates `function () ... end`, it creates a new function (a closure, actually). 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 that string before running the program? It does not matter: These are implementation details. Thus, from the programmer's point of view, strings are values, not objects. Therefore, like a number or a boolean, a string is not removed from weak tables (unless its associated value is collected).

17.1 - Memoize Functions

A common programming technique is to trade space for time. You can speed up some functions by *memoizing* their results so that, later, when you call the function with the same arguments, it can reuse the result.

Imagine a generic server that receives requests containing strings with Lua code. Each time it gets a request, it runs `loadstring` on the string, and then calls the resulting function. However, `loadstring` is an expensive function and some commands to the server may be quite frequent. Instead of calling `loadstring` over and over each time it receives a common command like `"closeconnection()"`, the server can *memoize* the results from `loadstring` using an auxiliary table. Before calling `loadstring`, the server checks in the table whether that string already has a translation. If it cannot find the string, then (and only then) the server calls `loadstring` and stores the result into the table. We can pack this behavior in a new function:

```
local results = {}
function mem_loadstring (s)
  if results[s] then      -- result available?
    return results[s]     -- reuse it
  else
    local res = loadstring(s)   -- compute new result
    results[s] = res            -- save for later reuse
    return res
  end
end
```

The savings with this scheme can be huge. However, it may also cause unsuspected wastes. 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 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 that table fully weak, if we want:

```
setmetatable(results, {__mode = "kv"})
```

The net result is exactly the same.

The memoize technique is also useful 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 the memoize technique, 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 = r .. "-" .. g .. "-" .. b
  if results[key] then return results[key]
  else
    local newcolor = {red = r, green = g, blue = b}
    results[key] = newcolor
    return newcolor
  end
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. Note that the same color may be represented by different tables at different times, because from time to time a garbage-collector cycle 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 survives long enough to be reused by the new color.

17.2 - 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 object (typically a table) and use it as 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 associate attributes to objects. Of course, an external table provides an ideal way to associate attributes to objects (it is not by chance that tables are sometimes called *associative arrays*). 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. If we use a regular table to associate functions to its names, none of those 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.

Lua itself uses this technique to keep the size of tables used as arrays. As we will see later, the table library offers a function to set the size of an array and another to get this size. When you set the size of an array, Lua stores this size in a private weak table, where the index is the array itself and the value is its size.

17.3 - Revisiting Tables with Default Values

In Section 13.4.3, we discussed how to implement tables with non-nil default values. We saw one particular technique and commented that two other techniques need weak tables so we postponed them. Now it is time to revisit the subject. As we will see, those two techniques for default values are actually particular applications of the two general techniques that we have seen here: object attributes and memoizing.

In the first solution, we use a weak table to associate to each table 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
```

If `defaults` had not 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 memoizing:

```
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     -- memoize
  end
  setmetatable(t, mt)
end
```

We use weak values, in this case, 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 words for each table with a default value (an entry in `defaults`). The second implementation needs a few dozen words for each distinct default value (a new table, a new closure, plus an entry in `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 use the first one.

18 - The Mathematical Library

In this chapter (and in the other chapters about the standard libraries), my purpose is not to give the complete specification of each function, but to show you what kind of functionality the library can provide. I may omit some subtle options or behaviors for clarity of exposition. The main idea is to spark your curiosity, which can then be satisfied by the reference manual.

The `math` library comprises a standard set of mathematical functions, such as trigonometric functions (`sin`, `cos`, `tan`, `asin`, `acos`, etc.), exponentiation and logarithms (`exp`, `log`, `log10`), rounding functions (`floor`, `ceil`), `max`, `min`, plus a variable `pi`. The mathematical library also defines the operator `` `^´ `` to work as the exponentiation operator.

All trigonometric functions work in radians. (Until Lua 4.0, they worked in degrees.) You can use the functions `deg` and `rad` to convert between degrees and radians. If you want to work in degrees, you can redefine the trigonometric functions:

```
    local sin, asin, ... = math.sin, math.asin, ...
    local deg, rad = math.deg, math.rad
    math.sin = function (x) return sin(rad(x)) end
    math.asin = function (x) return deg(asin(x)) end
    ...
```

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 an integer pseudo-random number *x* such that *1 <= x <= n*. For instance, you can simulate the result of a die with `random(6)`. Finally, we can call `random` with two integer arguments, *l* and *u*, to get a pseudo-random integer *x* such that *l <= x <= u*.

You can set a seed for the pseudo-random generator with the `randomseed` function; its only numeric argument is the seed. Usually, when a program starts, it initializes the generator with a fixed seed. That means that, every time you run your program, it generates the same sequence of pseudo-random numbers. For debugging, that is a nice property; but in a game, you will have the same scenario over and over. A common trick to solve this problem is to use the current time as a seed:

```
    math.randomseed(os.time())
```

(The `os.time` function returns a number that represents the current time, usually as the number of seconds since some epoch.)

19 - The Table Library

The `table` library comprises auxiliary functions to manipulate tables as arrays. One of its main roles is to give a reasonable meaning for the *size* of an array in Lua. It also provides functions to insert and remove elements from lists and to sort the elements of an array.

19.1 - Array Size

Frequently, in Lua, we assume that an array ends just before its first nil element. This convention has one drawback: We cannot have a **nil** inside an array. For several applications this restriction is not a hindrance, such as when all elements in the array have a fixed type. But sometimes we must allow **nil**s inside an array. In such cases, we need a method to keep an explicit size for an array.

The table library defines two functions to manipulate array sizes: `getn`, which returns the size of an array, and `setn`, which sets the size of an array. As we saw earlier, there are two methods to associate an attribute to a table: Either we store the attribute in a field of the table, or we use a separate (weak) table to do the association. Both methods have pros and cons; for that reason, the `table` library uses both.

Usually, a call `table.setn(t, n)` associates `t` with `n` in an internal (weak) table and a call `table.getn(t)` retrieves the value associated with `t` in that internal table. However, if the table `t` has a field `"n"` with a numeric value, `setn` updates this value and `getn` returns it. The `getn` function still has a last option: If it cannot get an array size with any of those options, it uses the naive approach: to traverse the array looking for its first nil element. So, you can always use `table.getn(t)` in an array and get a reasonable result. See the examples:

```
    print(table.getn{10,2,4})          --> 3
    print(table.getn{10,2,nil})        --> 2
    print(table.getn{10,2,nil; n=3})   --> 3
    print(table.getn{n=1000})          --> 1000

    a = {}
    print(table.getn(a))               --> 0
    table.setn(a, 10000)
    print(table.getn(a))               --> 10000

    a = {n=10}
    print(table.getn(a))               --> 10
    table.setn(a, 10000)
    print(table.getn(a))               --> 10000
```

By default, `setn` and `getn` use the internal table to store a size. This is the cleanest option, as it does not pollute the array with an extra element. However, the `n`-field option has some advantages too. The Lua core uses this option to set the size of the `arg` array, in functions with variable number of arguments; because the core cannot depend on a library, it cannot use `setn`. Another advantage of this option is that we can set the size of an array directly in its constructor, as we saw in the examples.

It is a good practice to use both `setn` and `getn` to manipulate array sizes, even when you know that the size is at field `n`. All functions from the `table` library (`sort`, `concat`, `insert`, etc.) follow this practice. Actually, the possibility of `setn` to change the value of the field `n` is provided only for compatibility with older versions of Lua. This behavior may change in future versions of the language. To play safe, do not assume this behavior. Always use `getn` to get a size set by `setn`.

19.2 - Insert and Remove

The `table` library provides functions to insert and to remove elements from arbitrary positions of a list. The `table.insert` function inserts an element in a given position of an array, moving up other elements to open space. Moreover, `insert` increments the size of the array (using `setn`). For instance, if `a` is the array `{10, 20, 30}`, after the call `table.insert(a, 1, 15)` `a` will be `{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 array (and, therefore, moves no elements). As an example, the following code reads the program input line by line, storing all lines in an array:

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

The `table.remove` function removes (and returns) an element from a given position in an array, moving down other elements to close space and decrementing the size of the array. When called without a position, it removes the last element of the array.

With those two functions, it is straightforward to implement stacks, queues, and double queues. We can initialize such structures as `a = {}`. A push operation is equivalent to `table.insert(a, x)`; a pop operation is equivalent to `table.remove(a)`. To insert at the other end of the structure we use `table.insert(a, 1, x)`; to remove from that end we use `table.remove(a, 1)`. 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 and this implementation is good enough for small arrays (up to some hundred elements, say).

19.3 - Sort

Another useful function on arrays is `table.sort`, which we have seen before. It receives the array to be sorted, plus an optional order function. This order function receives two arguments and must return true if the first argument should come first in the sorted array. If this function is not provided, `sort` uses the default less-than operation (corresponding to the `<´ operator).

A common mistake is to try to order the indices of a table. In a table, the indices form a set, and have no order whatsoever. If you want to order them, you have to copy them to an array and then sort the array. Let us see an example. Suppose that you read a source file and build a table that gives, for each function name, the line where that function is defined; something like this:

```
lines = {
  luaH_set = 10,
  luaH_get = 24,
  luaH_present = 48,
}
```

Now you want to print these function names in alphabetical order. If you traverse this table with `pairs`, the names appear in an arbitrary order. However, you cannot sort them directly, because these names are keys of the table. However, when you put these names into an array, then you can sort them. First, you must create an array with those names, then sort it, and finally print the result:

```
a = {}
for n in pairs(lines) do table.insert(a, n) end
table.sort(a)
for i,n in ipairs(a) do print(n) end
```

Note that, for Lua, arrays also have no order. But we know how to count, so we get ordered values as long as we access the array with ordered indices. That is why you should always traverse arrays with `ipairs`, rather than `pairs`. The first imposes the key order 1, 2, ..., whereas the latter uses the natural arbitrary order of the table.

As a more advanced solution, we can write an iterator that traverses a table following the order of its keys. An optional parameter `f` allows the specification of an alternative order. It first sorts the keys into an array, and then iterates on the array. At each step, it returns the key and value from the original table:

```
function pairsByKeys (t, f)
  local a = {}
  for n in pairs(t) do table.insert(a, n) end
  table.sort(a, f)
  local i = 0      -- iterator variable
  local iter = function ()   -- iterator function
    i = i + 1
    if a[i] == nil then return nil
    else return a[i], t[a[i]]
    end
  end
  return iter
end
```

With this function, it is easy to print those function names in alphabetical order. The loop

```
for name, line in pairsByKeys(lines) do
  print(name, line)
end
```

will print

```
luaH_get        24
luaH_present    48
luaH_set        10
```

20 - The String Library

The power of a raw Lua interpreter to manipulate strings is quite limited. A program can create string literals and concatenate them. But it cannot extract a substring, check its size, or examine its contents. The full power to manipulate strings in Lua comes from its string library.

Some functions in the string library are quite simple: `string.len(s)` returns the length of a string `s`. `string.rep(s, n)` returns the string `s` repeated `n` times. You can create a string with 1M bytes (for tests, for instance) with `string.rep("a", 2^20)`. `string.lower(s)` returns a copy of `s` with the upper-case letters converted to lower case; all other characters in the string are not changed (`string.upper` converts to upper case). As a typical use, if you want to sort an array of strings regardless of case, you may write something like

```
table.sort(a, function (a, b)
  return string.lower(a) < string.lower(b)
end)
```

Both `string.upper` and `string.lower` follow the current locale. Therefore, if you work with the European Latin-1 locale, the expression

```
    string.upper("ação")
```

results in `"AÇÃO"`.

The call `string.sub(s,i,j)` extracts a piece of the string `s`, from the `i`-th to the `j`-th character inclusive. In Lua, the first character of a string has index 1. You can also use negative indices, which count from the end of the string: The index *-1* refers to the last character in a string, *-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 (if you do not provide a third argument, it defaults to *-1*, so we could write the last call as `string.sub(s, j)`); and `string.sub(s, 2, -2)` returns a copy of the string `s` with the first and last characters removed:

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

Remember that strings in Lua are immutable. The `string.sub` function, like any other function in Lua, 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 to assume that the value of `s` will be modified. If you want to modify the value of a variable, you must assign the new value to the variable:

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

The `string.char` and `string.byte` functions 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 those characters. The function `string.byte(s, i)` returns the internal numeric representation of the `i`-th character of the string `s`; the second argument is optional, so that a call `string.byte(s)` returns the internal numeric representation of the first (or single) character of `s`. In the following examples, we assume that characters are represented in ASCII:

```
    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.

The function `string.format` is a powerful tool when formatting strings, typically for output. It returns a formatted version of its variable number of arguments following the description given by its first argument, the so-called *format* string. The format string has rules similar to those of the `printf` function of standard C: It is composed of regular text and *directives*, which control where and how each argument must be placed in the formatted string. A simple directive is the character `` `%´ `` plus a letter that tells how to format the argument: `` `d´ `` for a decimal number, `` `x´ `` for hexadecimal, `` `o´ `` for octal, `` `f´ `` for a floating-point number, `` `s´ `` for strings, plus other variants. Between the `` `%´ `` and the letter, a directive can include other options, which control the details of the format, such as the number of decimal digits of a floating-point number:

```
    print(string.format("pi = %.4f", PI))     --> pi = 3.1416
    d = 5; m = 11; y = 1990
```

```
print(string.format("%02d/%02d/%04d", d, m, y))
  --> 05/11/1990
tag, title = "h1", "a title"
print(string.format("<%s>%s</%s>", tag, title, tag))
  --> <h1>a title</h1>
```

In the first example, the `%.4f` means a floating-point number with four digits after the decimal point. In the second example, the `%02d` means a decimal number (`` `d' ``), with at least two digits and zero padding; the directive `%2d`, without the zero, would use blanks for padding. For a complete description of those directives, see the Lua reference manual. Or, better yet, see a C manual, as Lua calls the standard C libraries to do the hard work here.

20.1 - Pattern-Matching Functions

The most powerful functions in the string library are `string.find` (*string Find*), `string.gsub` (*Global Substitution*), and `string.gfind` (*Global Find*). They all are based on *patterns*.

Unlike several other scripting languages, Lua does not use POSIX regular expressions (regexp) for pattern matching. The main reason for this is size: A typical implementation of POSIX regexp takes more than 4,000 lines of code. This is bigger than all Lua standard libraries together. In comparison, the implementation of pattern matching in Lua has less than 500 lines. Of course, the 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 basic use of `string.find` is to search for a pattern inside a given string, called the *subject* string. The function returns the position where it found the pattern or **nil** if it could not find it. 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 `find` finds its pattern, it returns two values: the index where the match begins and the index where the match ends.

```
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, a `string.sub` of the values returned by `string.find` would return the part of the subject string that matched the pattern. (For simple patterns, this is the pattern itself.)

The `string.find` function has an optional third parameter: an index that tells where in the subject string to start the search. This parameter is useful when we want to process all the indices where a given pattern appears. We search for a new pattern repeatedly, each time starting after the position where we found the previous one. As an example, the following code makes a table with the positions of all newlines in a string:

```
local t = {}                    -- table to store the indices
local i = 0
while true do
  i = string.find(s, "\n", i+1)    -- find 'next' newline
  if i == nil then break end
```

```
      table.insert(t, i)
    end
```

We will see later a simpler way to write such loops, using the `string.gfind` iterator.

The `string.gsub` function has three 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", "perl", "tcl")
    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
```

The `string.gsub` function also returns as a second result the number of times it made the substitution. For instance, an easy way to count the number of spaces in a string is

```
    _, count = string.gsub(str, " ", " ")
```

(Remember, the _ is just a dummy variable name.)

20.2 - Patterns

You can make patterns more useful 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, you 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.sub(s, string.find(s, date)))   --> 30/05/1999
```

The following table lists all character classes:

`.`	all characters
`%a`	letters
`%c`	control characters
`%d`	digits
`%l`	lower case letters
`%p`	punctuation characters
`%s`	space characters
`%u`	upper case letters
`%w`	alphanumeric characters
`%x`	hexadecimal digits

`%z`	the character with representation 0

An upper case version of any of those 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. 4
```

(The `4` is not part of the result string. It is the second result of `gsub`, the total number of substitutions. Other examples that print the result of `gsub` will omit this count.)

Some characters, called *magic characters*, have special meanings when used in a pattern. The magic characters are

```
    ( ) . % + - * ? [ ^ $
```

The character `` `%´ `` works as an escape for those magic characters. So, '`%.`' matches a dot; '`%%`' matches the character `` `%´ `` itself. You can use the escape `` `%´ `` not only for the magic characters, but also for all other non-alphanumeric characters. When in doubt, play safe and put an escape.

For Lua, patterns are regular strings. They have no special treatment and follow the same rules as other strings. Only inside the functions are they interpreted as patterns and only then does the `` `%´ `` work as an escape. Therefore, if you need to put a quote inside a pattern, you must use the same techniques that you use to put a quote inside other strings; for instance, you can escape the quote with a `` `\´ ``, which is the escape character for Lua.

A *char-set* allows you to create your own character classes, combining different classes and single characters between square brackets. For instance, the char-set '`[%w_]`' matches both alphanumeric characters and underscores, the char-set '`[01]`' matches binary digits, and the char-set '`[%[%]]`' matches square brackets. To count the number of vowels in a text, you can write

```
    _, nvow = string.gsub(text, "[AEIOUaeiou]", "")
```

You can also include character ranges in a char-set, by writing the first and the last characters of the range separated by a hyphen. You will seldom need this facility, because most useful ranges are already predefined; for instance, '`[0-9]`' is simpler when written as '`%d`', '`[0-9a-fA-F]`' is the same as '`%x`'. However, if you need to find an octal digit, then you may prefer '`[0-7]`', instead of an explicit enumeration ('`[01234567]`'). You can get the complement of a char-set by starting it with `` `^´ ``: '`[^0-7]`' finds any character that is not an octal digit and '`[^\n]`' matches any character different from newline. But remember that you can negate simple classes with its upper case version: '`%S`' is simpler than '`[^%s]`'.

Character classes follow the current locale set for Lua. Therefore, the class '`[a-z]`' can be different from '`%l`'. In a proper locale, the latter form includes letters such as `` `ç´ `` and `` `ã´ ``. You should always use the latter form, unless you have a strong reason to do otherwise: It is simpler, more portable, and slightly more efficient.

You 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
`-`	also 0 or more repetitions
`?`	optional (0 or 1 occurrence)

The `` `+´ `` 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, or 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):

```
    i, j = string.find("the number 1298 is even", "%d+")
    print(i,j)    --> 12  15
```

The modifier `*´ is similar to `+´, 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 ( ), you use the pattern '%(%s*%)'. (The pattern '%s*' matches zero or more spaces. Parentheses have a special meaning in a pattern, so we must escape them with a `%´.) As another example, the pattern '[_%a][_%w]*' matches identifiers in a Lua program: a sequence that starts with a letter or an underscore, followed by zero or more underscores or alphanumeric characters.

Like `*´, the 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 `*´ or `-´, but usually they present rather different results. For instance, if you try to find an identifier with the pattern '[_%a][_%w]-', you will find only the first letter, because the '[_%w]-' will always match the empty sequence. On the other hand, suppose you want to find 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, "/%*.*%*/", "<COMMENT>"))
      --> int x; <COMMENT>
```

The pattern '.-', instead, will expand the least amount necessary to find the first "*/", so that you get your desired result:

```
    test = "int x; /* x */  int y; /* y */"
    print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
        --> int x; <COMMENT>  int y; <COMMENT>
```

The last modifier, `?´, matches an optional character. As an example, suppose we want to find an integer in a text, where the number may contain an optional sign. The pattern '[+-]?%d+' does the job, matching numerals like "-12", "23" and "+1009". The '[+-]' is a character class that matches both a `+´ or a `-´ sign; the following `?´ makes that sign optional.

Unlike some other systems, in Lua a modifier can only be applied 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 you can circumvent this limitation using some of the advanced techniques that we will see later.

If a pattern begins with a `^´, it will match only at the beginning of the subject string. Similarly, if it ends with a `$´, it will match only at the end of the subject string. These marks can be used both to restrict the patterns that you find and to anchor patterns. For instance, the test

```
    if string.find(s, "^%d") then ...
```

checks whether the string s starts with a digit and the test

```
    if string.find(s, "^[+-]?%d+$") then ...
```

checks whether that string represents an integer number, without other leading or trailing characters.

Another item in a pattern is the '`%b`', that matches balanced strings. Such item is written 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 `` `(´ `` and finish at the respective `` `)´ ``:

```
    print(string.gsub("a (enclosed (in) parentheses) line",
                      "%b()", ""))
      --> a  line
```

Typically, this pattern is used as '`%b()`', '`%b[]`', '`%b%{%}`', or '`%b<>`', but you can use any characters as delimiters.

20.3 - Captures

The *capture* mechanism allows a pattern to yank parts of the subject string that match parts of the pattern, for further use. You specify a capture by writing the parts of the pattern that you want to capture between parentheses.

When you specify captures to `string.find`, it returns the captured values as extra results from the call. A typical use of this facility is to break a string into parts:

```
    pair = "name = Anna"
    _, _, key, value = string.find(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 `` `=´ ``, again followed by spaces plus another sequence of letters. Both sequences of letters have their patterns enclosed by parentheses, so that they will be captured if a match occurs. The `find` function always returns first the indices where the matching happened (which we store in the dummy variable _ in the previous example) and then the captures made during the pattern matching. Below is a similar example:

```
    date = "17/7/1990"
    _, _, d, m, y = string.find(date, "(%d+)/(%d+)/(%d+)")
    print(d, m, y)  --> 17  7  1990
```

We can also use captures in the pattern itself. In a pattern, an item like '`%`*d*', where *d* is a single digit, matches only a copy of the *d*-th capture. As a typical use, suppose you want to find, inside a string, a substring enclosed between single or double quotes. You could try a pattern such as '`["'].-["']`', that is, a quote followed by anything followed by another quote; but you would have problems with strings like `"it's all right"`. To solve that problem, you can capture the first quote and use it to specify the second one:

```
    s = [[then he said: "it's all right"!]]
    a, b, c, quotedPart = string.find(s, "([\"'])(.-)%1")
    print(quotedPart)   --> it's all right
    print(c)            --> "
```

The first capture is the quote character itself and the second capture is the contents of the quote (the substring matching the '`.-`').

The third use of captured values is in the replacement string of `gsub`. Like the pattern, the replacement string may contain items like '`%`*d*', which are changed to the respective captures when the substitution is made. (By the way, because of those changes, a `%´ 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)", "%1-%1"))
  -->  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, such as

```
\command{some text}
```

and changes them to a format in XML style,

```
<command>some text</command>
```

For this specification, the following line does the job:

```
s = string.gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
```

For instance, if `s` is the string

```
the \quote{task} is to \em{change} that.
```

that `gsub` call will change it to

```
the <quote>task</quote> is to <em>change</em> that.
```

Another useful example is how to trim a string:

```
function trim (s)
  return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
end
```

Note the judicious use of pattern formats. The two anchors (`^´ and `$´) ensure that we get the whole string. Because the '.-' tries to expand as little as possible, the two patterns '%s*' match all spaces at both extremities. Note also that, because `gsub` returns two values, we use extra parentheses to discard the extra result (the count).

The last use of captured values is perhaps the most powerful. We can call `string.gsub` with a function as its third argument, instead of a replacement string. When invoked this way, `string.gsub` calls the given function every time it finds a match; the arguments to this function are the captures, while the value that the function returns is used as the replacement string. 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)
  s = string.gsub(s, "$(%w+)", function (n)
        return _G[n]
      end)
  return s
end

name = "Lua"; status = "great"
print(expand("$name is $status, isn't it?"))
  --> Lua is great, isn't it?
```

If you are not sure whether the given variables have string values, you can apply `tostring` to their values:

```
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
```

A more powerful example uses `loadstring` to evaluate whole expressions that we write in the text enclosed by square brackets preceded by a dollar sign:

```
s = "sin(3) = $[math.sin(3)]; 2^5 = $[2^5]"

print((string.gsub(s, "$(%b[])", function (x)
         x = "return " .. string.sub(x, 2, -2)
         local f = loadstring(x)
         return f()
       end)))
  -->  sin(3) = 0.1411200080598672; 2^5 = 32
```

The first match is the string "`$[math.sin(3)]`", whose corresponding capture is "`[math.sin(3)]`". The call to `string.sub` removes the brackets from the captured string, so the string loaded for execution will be "`return math.sin(3)`". The same happens for the match "`$[2^5]`".

Often we want a kind of `string.gsub` only to iterate on a string, without any interest in the resulting string. For instance, we could collect the words of a string into a table with the following code:

```
words = {}
string.gsub(s, "(%a+)", function (w)
  table.insert(words, w)
end)
```

If `s` were the string "`hello hi, again!`", after that command the `word` table would be

```
{"hello", "hi", "again"}
```

The `string.gfind` function offers a simpler way to write that code:

```
words = {}
for w in string.gfind(s, "(%a)") do
  table.insert(words, w)
end
```

The `gfind` function fits perfectly with the generic **for** loop. It returns a function that iterates on all occurrences of a pattern in a string.

We can simplify that code a little bit more. When we call `gfind` with a pattern without any explicit capture, the function will capture the whole pattern. Therefore, we can rewrite the previous example like this:

```
words = {}
for w in string.gfind(s, "%a") do
  table.insert(words, w)
end
```

For our next example, we use *URL encoding*, which is the encoding used by HTTP to send parameters in a URL. This encoding encodes special characters (such as `=´, `&´, and `+´) as "%*XX*", where *XX* is the hexadecimal representation of the character. Then, it changes spaces to `+´. 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 `=´ in between and appends all 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
```

The first statement changes each `+´ in the string to a space. The second gsub matches all two-digit hexadecimal numerals preceded by `%´ and calls an anonymous function. That function converts the hexadecimal numeral into a number (tonumber, with base 16) and returns the corresponding character (string.char). For instance,

```
print(unescape("a%2Bb+%3D+c"))  --> a+b = c
```

To decode the pairs name=value we use gfind. Because both names and values cannot contain either `&´ or `=´, we can match them with the pattern '[^&=]+':

```
cgi = {}
function decode (s)
  for name, value in string.gfind(s, "([^&=]+)=([^&=]+)") do
    name = unescape(name)
    value = unescape(value)
    cgi[name] = value
  end
end
```

That call to gfind matches all pairs in the form name=value and, for each pair, the iterator returns the corresponding captures (as marked by the parentheses in the matching string) as the values to name and value. The loop body simply calls unescape on 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 `%´ followed by the character ASCII code in hexadecimal (the format option "%02X" makes an hexadecimal number with two digits, using 0 for padding), and then changes spaces to `+´:

```
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 s = ""
      for k,v in pairs(t) do
        s = s .. "&" .. escape(k) .. "=" .. escape(v)
      end
      return string.sub(s, 2)     -- remove first `&'
    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
```

20.4 - Tricks of the Trade

Pattern matching is a powerful tool for manipulating strings. You can perform many complex operations with only a few calls to `string.gsub` and `find`. However, as with any power, you must use it carefully.

Pattern matching is not a replacement for a proper parser. For quick-and-dirty programs, you can do useful manipulations on source code, but it is hard to build a product with quality. As a good example, consider the pattern we used to match comments in a C program: '`/%*.-%*/`'. If your program has a string containing `"/*"`, you will 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 and, for your own use, that pattern will probably do its job. But you cannot sell a program with such a flaw.

Usually, pattern matching is efficient enough for Lua programs: A Pentium 333MHz (which is not a fast machine by today's standards) takes less than a tenth of a second to match all words in a text with 200K characters (30K words). But you can take precautions. You should always make the pattern as specific as possible; loose patterns are slower than specific ones. An extreme example is '`(.-)%$`', to get all text in a string up to the first dollar sign. If the subject string has a dollar sign, everything goes fine; but suppose that the string does not contain any dollar signs. The algorithm will first try to match the pattern starting at the first position of the string. It will go through all the string, looking for a dollar. When the string ends, the pattern fails *for the first position* of the string. Then, the algorithm will do the whole search again, starting at the second position of the string, only to discover that the pattern does not match there, too; and so on. This will take a quadratic time, which results in more than three hours in a Pentium 333MHz for a string with 200K characters. You can correct this problem simply by anchoring the pattern at the first position of the string, with '`^(.-)%$`'. The anchor tells the algorithm to stop the search if it cannot find a match at the first position. With the anchor, the pattern runs in less than a tenth of a second.

Beware also of *empty* patterns, that is, patterns that match the empty string. For instance, if you try to match names with a pattern like '`%a*`', you 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 begins or ends with the modifier `-´, because it will match only the empty string. This modifier always needs something around it, to anchor its expansion. Similarly, a pattern that includes '.*' is tricky, because this construction can expand much more than you intended.

Sometimes, it is useful to use Lua itself to build a pattern. As an example, let us see how we can find long lines in a text, say lines with more than 70 characters. Well, 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, followed by zero or more of those characters. Instead of writing this pattern by hand, we can create it with `string.rep`:

```
pattern = string.rep("[^\n]", 70) .. "[^\n]*"
```

As another example, suppose you want to make a case-insensitive search. A way to do that is to change any letter *x* in the pattern for the class '[*xX*]', that is, a class including both the upper and the lower versions of the original letter. We can automate that conversion with a function:

```
function nocase (s)
  s = string.gsub(s, "%a", function (c)
        return string.format("[%s%s]", string.lower(c),
                                       string.upper(c))
      end)
  return s
end

print(nocase("Hi there!"))
  -->  [hH][iI] [tT][hH][eE][rR][eE]!
```

Sometimes, you want to change every plain occurrence of `s1` to `s2`, without regarding any character as magic. If the strings `s1` and `s2` are literals, you can add proper escapes to magic characters while you write the strings. But if those strings are variable values, you can use another `gsub` to put the escapes for you:

```
s1 = string.gsub(s1, "(%W)", "%%%1")
s2 = string.gsub(s2, "%%", "%%%%")
```

In the search string, we escape all non-alphanumeric characters. In the replacement string, we escape only the `%´.

Another useful technique for pattern matching is to pre-process the subject string before the real work. A simple example of the use of pre-processing is 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\"!".
```

Our approach to handling such cases is to pre-process the text so as to encode the problematic sequence to 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(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\"!".
```

or, in a more compact notation,

```
print(decode(string.gsub(code(s), '(".-")', string.upper)))
```

As a more complex task, let us return to our example of a primitive format converter, which changes format commands written as `\command{string}` to XML style:

```
<command>string</command>
```

But now our original format is more powerful and uses the backslash character as a general escape, so that we can represent the characters `\´, `{´, and `}´, writing "\\", "\{", and "\}". To avoid our pattern matching mixing up commands and escaped characters, we should recode those sequences in the original string. However, this time we cannot code all sequences \*x*, because that would code our commands (written as `\command`) too. Instead, we code *x* only when *x* is not a letter:

```
function code (s)
  return (string.gsub(s, '\\(%A)', function (x)
           return string.format("\\%03d", string.byte(x))
         end))
end
```

The `decode` is like that of the previous example, but it does not include the backslashes in the final string; therefore, we can call `string.char` directly:

```
function decode (s)
  return (string.gsub(s, '\\(%d%d%d)', string.char))
end

s = [[a \emph{command} is written as \\command\{text\}.]]
s = code(s)
s = string.gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
print(decode(s))
  -->  a <emph>command</emph> is written as \command{text}.
```

Our last example here deals with *Comma-Separated Values* (CSV), a text format supported by many programs, such as Microsoft Excel, to represent tabular data. A CSV file represents a list of records, where each record is a list of string values written in a single line, with commas between the values. Values that contain commas must be written between double quotes; if such values also have quotes, the quotes are written as two quotes. As an example, the array

```
{'a b', 'a,b', ' a,"b"c', 'hello "world"!', ''}
```

can be represented as

```
a b,"a,b"," a,""b""c", hello "world"!,
```

To transform an array of strings into CSV is easy. All we have to do is to concatenate the strings with commas between them:

```
function toCSV (t)
  local s = ""
  for _,p in pairs(t) do
    s = s .. "," .. escapeCSV(p)
  end
  return string.sub(s, 2)      -- remove first comma
end
```

If a string has commas or quotes inside, we enclose it between quotes and escape its original quotes:

```
function escapeCSV (s)
  if string.find(s, '[,"]') then
    s = '"' .. string.gsub(s, '"', '""') .. '"'
  end
  return s
end
```

To break a CSV into an array is more difficult, because we must avoid mixing up the commas written between quotes with the commas that separate fields. We could try to escape the commas between quotes. However, not all quote characters act as quotes; only quote characters after a comma act as a starting quote, as long as the comma itself is acting as a comma (that is, it is not between quotes). There are too many subtleties. For instance, two quotes may represent a single quote, two quotes, or nothing:

```
"hello""hello", "",""
```

The first field in this example is the string `"hello"hello"`, the second field is the string `" """` (that is, a space followed by two quotes), and the last field is an empty string.

We could try to use multiple `gsub` calls to handle all those cases, but it is easier to program this task with a more conventional approach, using an explicit loop over the fields. The main task of the loop body is to find the next comma; it also stores the field contents in a table. For each field, we explicitly test whether the field starts with a quote. If it does, we do a loop looking for the closing quote. In this loop, we use the pattern `'"("?)'` to find the closing quote of a field: If a quote is followed by another quote, the second quote is captured and assigned to the `c` variable, meaning that this is not the closing quote yet.

```
function fromCSV (s)
  s = s .. ','         -- ending comma
  local t = {}         -- table to collect fields
  local fieldstart = 1
  repeat
    -- next field is quoted? (start with `"'?)
    if string.find(s, '^"', fieldstart) then
      local a, c
      local i  = fieldstart
      repeat
        -- find closing quote
        a, i, c = string.find(s, '"("?)', i+1)
      until c ~= '"'    -- quote not followed by quote?
      if not i then error('unmatched "') end
      local f = string.sub(s, fieldstart+1, i-1)
```

```
        table.insert(t, (string.gsub(f, '""', '"')))
        fieldstart = string.find(s, ',', i) + 1
      else                -- unquoted; find next comma
        local nexti = string.find(s, ',', fieldstart)
        table.insert(t, string.sub(s, fieldstart, nexti-1))
        fieldstart = nexti + 1
      end
    until fieldstart > string.len(s)
    return t
  end

  t = fromCSV('"hello "" hello", "",""')
  for i, s in ipairs(t) do print(i, s) end
    --> 1       hello " hello
    --> 2        ""
    --> 3
```

21 - The I/O Library

The I/O library offers two different models for file manipulation. The simple model assumes a *current* input and a *current* output files, and its I/O operations operate on those files. The complete model uses explicit file handles and it adopts an object-oriented style that defines all operations as methods on file handles.

The simple model is convenient for simple things; we have been using it all along the book until now. But it is not enough for more advanced file manipulation, such as reading from several files simultaneously. For those manipulations, the complete model is more convenient.

The I/O library puts all its functions into the `io` table.

21.1 - The Simple I/O Model

The simple model does all of its operations on two current files. The library initializes the current input file as the process's standard input (`stdin`) and the current output file as 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 those current files with the `io.input` and `io.output` functions. A call like `io.input(filename)` opens the given file (in read mode) and sets it as the current input file. From this point on, all input will come from this file, until another call to `io.input`; `io.output` does a similar job for output. In case of errors, both functions raise the error. If you want to handle errors directly, you must use `io.open`, from the complete model.

As `write` is simpler than `read`, we will look at it first. The `io.write` function simply gets an arbitrary number of string arguments and writes them to the current output file. Numbers are converted to strings following the usual conversion rules; for full control over this conversion, you should use the `format` function, from the `string` library:

```
  > io.write("sin (3) = ", math.sin(3), "\n")
    --> sin (3) = 0.1411200080598672
  > io.write(string.format("sin (3) = %.4f\n", math.sin(3)))
    --> sin (3) = 0.1411
```

Avoid code 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` for quick-and-dirty programs, or for debugging, and `write` when you need full control over your output:

```
> print("hello", "Lua"); print("Hi")
  --> hello   Lua
  --> Hi

> io.write("hello", "Lua"); io.write("Hi", "\n")
  --> helloLuaHi
```

Unlike `print`, `write` adds no extra characters to the output, such as tabs or newlines. Moreover, `write` uses the current output file, whereas `print` always uses the standard output. Finally, `print` automatically applies `tostring` to its arguments, so it can also show tables, functions, and **nil**.

The `read` function reads strings from the current input file. Its arguments control what is read:

`"*all"`	reads the whole file
`"*line"`	reads the next line
`"*number"`	reads a number
num	reads a string with up to *num* characters

The call `io.read("*all")` reads the whole current input file, starting at its current position. If we are at the end of 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, do the processing to the string (typically with `gsub`), and then write the string to the output:

```
t = io.read("*all")          -- read the whole file
t = string.gsub(t, ...)      -- do the job
io.write(t)                  -- write the file
```

As an example, the following code is a complete program to code a file's content using the *quoted-printable* encoding of MIME. In this encoding, non-ASCII characters are coded as =*XX*, where *XX* is the numeric code of the character in hexadecimal. To keep the consistency of the encoding, the `=′ character must be encoded as well. The pattern used in the `gsub` captures all characters with codes from 128 to 255, plus the equal sign.

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

On a Pentium 333MHz, this program takes 0.2 seconds to convert a file with 200K characters.

The call `io.read("*line")` returns the next line from the current input file, without the newline character. When we reach the end of file, the call returns **nil** (as there is no next line to return). This pattern is the default for `read`, so `io.read()` has the same effect as `io.read("*line")`. Usually, we use this pattern only when our algorithm naturally handles the file line by line; otherwise, we favor reading the whole file at once, with `*all`, or in blocks, as we will see later. As a simple example of the use of this pattern, the following program copies its current input to the current output, numbering each line:

```
local count = 1
while true do
  local line = io.read()
  if line == nil then break end
```

```
    io.write(string.format("%6d  ", count), line, "\n")
    count = count + 1
  end
```

However, to iterate on a whole file line by line, we do better to use the `io.lines` iterator. For instance, we can write a complete program to sort the lines of a file as follows:

```
  local lines = {}
  -- read the lines in table 'lines'
  for line in io.lines() do
    table.insert(lines, line)
  end
  -- sort
  table.sort(lines)
  -- write all the lines
  for i, l in ipairs(lines) do io.write(l, "\n") end
```

This program sorts a file with 4.5 MB (32K lines) in 1.8 seconds (on a Pentium 333MHz), against 0.6 seconds spent by the system `sort` program, which is written in C and highly optimized.

The call `io.read("*number")` reads a number from the current input file. This is the only case where `read` returns a number, instead of a string. When you need to read many numbers from a file, the absence of the intermediate strings can make a significant performance improvement. The `*number` option skips any spaces before the number and accepts number formats like `-3`, `+5.2`, `1000`, and `-3.4e-23`. If it cannot find a number at the current file position (because of bad format or end of file), it returns **nil**.

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

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

Now you want to print the maximum of each line. You can read all three numbers in a single call to `read`:

```
  while true do
    local n1, n2, n3 = io.read("*number", "*number",
                               "*number")
    if not n1 then break end
    print(math.max(n1, n2, n3))
  end
```

In any case, you should always consider the alternative of reading the whole file with option `"*all"` from `io.read` and then using `gfind` to break it up:

```
  local pat = "(%S+)%s+(%S+)%s+(%S+)%s+"
  for n1, n2, n3 in string.gfind(io.read("*all"), pat) do
    print(math.max(n1, n2, n3))
  end
```

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

```
  local size = 2^13      -- good buffer size (8K)
  while true do
```

```
      local block = io.read(size)
      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.

21.2 - The Complete I/O Model

For more control over I/O, you can use the complete model. A central concept in this model is the *file handle*, which is equivalent to streams (`FILE*`) in C: It represents an open file with a current position.

To open a file, you use the `io.open` function, which mimics the `fopen` function in C. It receives as arguments the name of the file to open plus a *mode* string. That mode string may 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 `open` function returns a new handle for the file. In case of errors, `open` returns **nil**, plus an error message and an error number:

```
    print(io.open("non-existent file", "r"))
      --> nil     No such file or directory       2

    print(io.open("/etc/passwd", "w"))
      --> nil   Permission denied       13
```

The interpretation of the error numbers is system dependent.

A typical idiom to check for errors is

```
    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 you open a file, you can read from it or write to it with the methods `read`/`write`. They are similar to the `read`/`write` functions, but you call them as methods on the file handle, using the colon syntax. For instance, to open a file and read it all, you can use a chunk like this:

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

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

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

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

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

21.2.1 - A Small Performance Trick

Usually, in Lua, it is much faster to read a file as a whole than to read it line by line. However, sometimes we must face some big files (say, tens or hundreds megabytes) for which it is not reasonable to read them all at once. If you want to handle such big files with maximum performance, the fastest way is to read them in reasonably large chunks (e.g., 8 KB each). To avoid the problem of breaking lines in the middle, you simply ask to read a chunk plus a line:

```
    local lines, rest = f:read(BUFSIZE, "*line")
```

The variable `rest` will get the rest of any line broken by the chunk. We then concatenate the chunk and this rest of line. That way, the resulting chunk will always break at line boundaries.

A typical example of that technique is this implementation of `wc`, a program to count the number of characters, words, and lines in a file:

```
    local BUFSIZE = 2^13     -- 8K
    local f = io.input(arg[1])   -- open input file
    local cc, lc, wc = 0, 0, 0   -- char, line, and word counts
    while true do
      local lines, rest = f:read(BUFSIZE, "*line")
      if not lines then break end
      if rest then lines = lines .. rest .. '\n' end
      cc = cc + string.len(lines)
      -- count words in the chunk
      local _,t = string.gsub(lines, "%S+", "")
      wc = wc + t
      -- count newlines in the chunk
      _,t = string.gsub(lines, "\n", "\n")
      lc = lc + t
    end
    print(lc, wc, cc)
```

21.2.2 - Binary Files

The simple model functions `io.input` and `io.output` always open a file in text mode (the default). In Unix, there is no difference between binary files and text files. But in some systems, notably Windows, binary files must be opened with a special flag. To handle such binary files, you must use `io.open`, with the letter `` `b´ `` in the mode string.

Binary data in Lua are handled similarly to text. A string in Lua may contain any bytes and almost all functions in the libraries can handle arbitrary bytes. (You can even do pattern matching over binary data, as long as the pattern does not contain a zero byte. If you want to match the byte zero, you can use the class `%z` instead.)

Typically, you read binary data either with the `*all` pattern, that reads the whole file, or with the pattern *n*, that reads *n* bytes. As a simple example, the following program converts a text file from DOS format to Unix format (that is, it translates sequences of carriage return-newlines to newlines). It does not use the standard I/O files (`stdin`/`stdout`), because those files are open in text mode. Instead, it assumes that the names of the input file and the output file are given as arguments to the program:

```
    local inp = assert(io.open(arg[1], "rb"))
    local out = assert(io.open(arg[2], "wb"))
    local data = inp:read("*all")
    data = string.gsub(data, "\r\n", "\n")
    out:write(data)

    assert(out:close())
```

You 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. 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, that comprises the alphanumeric, the punctuation, and the space characters. We use concatenation and `string.rep` to create a pattern that captures all sequences of six or more `validchars`. The `%z` at the end of the pattern matches the byte zero at the end of a string.

```
    local f = assert(io.open(arg[1], "rb"))
    local data = f:read("*all")
    local validchars = "[%w%p%s]"
    local pattern = string.rep(validchars, 6) .. "+%z"
    for w in string.gfind(data, pattern) do
      print(w)
    end
```

As a last example, the following program makes a dump of a binary file. Again, the first program argument is the input file name; the output goes to the standard output. The program reads the file in chunks of 10 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.

```
    local f = assert(io.open(arg[1], "rb"))
    local block = 10
    while true do
      local bytes = f:read(block)
      if not bytes then break end
      for b in string.gfind(bytes, ".") do
        io.write(string.format("%02X ", string.byte(b)))
      end
      io.write(string.rep("   ", block - string.len(bytes) + 1))
      io.write(string.gsub(bytes, "%c", "."), "\n")
    end
```

Suppose we store that program in a file named `vip`; if we apply the program to itself, with the call

```
    prompt> lua vip vip
```

it will produce an output like this (in a Unix machine):

```
    6C 6F 63 61 6C 20 66 20 3D 20    local f =
    61 73 73 65 72 74 28 69 6F 2E    assert(io.
    6F 70 65 6E 28 61 72 67 5B 31    open(arg[1
    5D 2C 20 22 72 62 22 29 29 0A    ], "rb")).
                 ...
    22 25 63 22 2C 20 22 2E 22 29    "%c", ".")
    2C 20 22 5C 6E 22 29 0A 65 6E    , "\n").en
    64 0A                            d.
```

21.3 - Other Operations on Files

The `tmpfile` function returns a handle for a temporary file, open in read/write mode. That file is automatically removed (deleted) when your program ends. The `flush` function executes all pending writes to a file. Like the `write` function, you can call it as a function, `io.flush()`, to flush the current output file; or as a method, `f:flush()`, to flush file `f`.

The `seek` function can be used both to get and to set the current position of a file. Its general form is `filehandle:seek(whence, offset)`. The `whence` parameter is a string that specifies how the offset will be interpreted. Its valid values are `"set"`, when offsets are interpreted from the beginning of the file; `"cur"`, when offsets are interpreted from the current position of the file; and `"end"`, when offsets are interpreted from the end of the file. Independently of the value of `whence`, the call returns the final current position of the file, 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 file 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()      -- get current position
  local size = file:seek("end")    -- get file size
  file:seek("set", current)        -- restore position
  return size
end
```

All the previous functions return **nil** plus an error message in case of errors.

22 - The Operating System Library

The Operating System library includes functions for file manipulation, for getting the current date and time, and other facilities related to the operating system. It is defined in table `os`. This library pays a price for Lua portability. Because Lua is written in ANSI C, it uses only the functions that the ANSI standard defines. Many OS facilities, such as directory manipulation and sockets, are not part of this standard and therefore the system library does not provide them. There are other Lua libraries, not included in the main distribution, that provide extended OS access. Examples are the `posix` library, which offers all functionality of the POSIX.1 standard to Lua; and `luasocket`, for network support.

For file manipulation, all that this library provides is an `os.rename` function, that changes the name of a file; and `os.remove`, that removes (deletes) a file.

22.1 - Date and Time

Two functions, `time` and `date`, do all date and time queries in Lua.

The `time` function, when called without arguments, returns the current date and time, coded as a number. (In most systems, that number is the number of seconds since some epoch.) When called with a table, it returns the number representing the date and time described by the table. Such *date* *tables* have the following significant fields:

`year`	a full year
`month`	01-12
`day`	01-31
`hour`	01-31
`min`	00-59
`sec`	00-59
`isdst`	a boolean, **true** if daylight saving

The first three fields are mandatory; the others default to noon (12:00:00) when not provided. In a Unix system (where the epoch is 00:00:00 UTC, January 1, 1970) running in Rio de Janeiro (which is three hours west of Greenwich), we have the following examples:

```
-- obs: 10800 = 3*60*60 (3 hours)
print(os.time{year=1970, month=1, day=1, hour=0})
  --> 10800
print(os.time{year=1970, month=1, day=1, hour=0, sec=1})
  --> 10801
print(os.time{year=1970, month=1, day=1})
  --> 54000   (obs: 54000 = 10800 + 12*60*60)
```

The `date` function, despite its name, is a kind of a reverse of the `time` function: It converts a number representing the date and time back to some higher-level representation. Its first parameter is a *format string*, describing the representation we want. The second is the numeric date-time; it defaults to the current date and time.

To produce a date table, we use the format string `"*t"`. For instance, the following code

```
temp = os.date("*t", 906000490)
```

produces the table

```
{year = 1998, month = 9, day = 16, yday = 259, wday = 4,
 hour = 23, min = 48, sec = 10, isdst = false}
```

Notice that, besides the fields used by `os.time`, the table created by `os.date` also gives the week day (`wday`, 1 is Sunday) and the year day (`yday`, 1 is January 1).

For other format strings, `os.date` formats the date as a string, which is a copy of the format string where specific tags are replaced by information about time and date. All tags are represented by a `` `% ``´ followed by a letter, as in the next examples:

```
print(os.date("today is %A, in %B"))
  --> today is Tuesday, in May
print(os.date("%x", 906000490))
  --> 09/16/1998
```

All representations follow the current locale. Therefore, in a locale for Brazil-Portuguese, `%B` would result in `"setembro"` and `%x` in `"16/09/98"`.

The following table shows each tag, its meaning, and its value for September 16, 1998 (a Wednesday), at 23:48:10. For numeric values, the table shows also their range of possible values:

`%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]
`%M`	minute (`48`) [00-59]
`%m`	month (`09`) [01-12]
`%p`	either `"am"` or `"pm"` (`pm`)
`%S`	second (`10`) [00-61]
`%w`	weekday (`3`) [0-6 = Sunday-Saturday]
`%x`	date (e.g., `09/16/98`)
`%X`	time (e.g., `23:48:10`)
`%Y`	full year (`1998`)
`%y`	two-digit year (`98`) [00-99]
`%%`	the character `%′

If you call `date` without any arguments, it uses the `%c` format, that is, complete 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 `mm/dd/yyyy`, use an explicit format string, such as `"%m/%d/%Y"`.

The `os.clock` function returns the number of seconds of CPU time for 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))
```

22.2 - Other System Calls

The `os.exit` function terminates the execution of a program. The `os.getenv` function gets the value of an environment variable. It receives the name of the variable and returns a string with its value:

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

If the variable is not defined, the call returns **nil**. The function `os.execute` runs a system command; it is equivalent to the `system` function in C. It receives a string with the command and returns an error code. For instance, both in Unix and in DOS-Windows, you can write the following function to create new directories:

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

The `os.execute` function is powerful, but it is also highly system dependent.

The `os.setlocale` function sets the current locale used by a Lua program. Locales define behavior that is sensitive to cultural or linguistic differences. The `setlocale` function has two string parameters: the locale name and a category, which specifies what features the locale will affect. There are six categories of locales: `"collate"` controls the alphabetic order of strings; `"ctype"` controls the types of individual characters (e.g., what is a letter) and the conversion between lower and upper cases; `"monetary"` has no influence in Lua programs; `"numeric"` controls how numbers are formatted; `"time"` controls how date and time are formatted (i.e., function `os.date`); and `"all"` controls all the above functions. The default category is `"all"`, so that if you call `setlocale` with only the locale name it will set all categories. The `setlocale` function returns the locale name or **nil** if it fails (usually because the system does not support the given locale).

```
print(os.setlocale("ISO-8859-1", "collate"))   --> ISO-8859-1
```

The category `"numeric"` is a little tricky. Although Portuguese and other Latin languages use a comma instead of a point to represent decimal numbers, the locale does not change the way that Lua parses numbers (among other reasons because expressions like `print(3,4)` already have a meaning in Lua). Therefore, you may end with a system that cannot recognize numbers with commas, but cannot understand numbers with points either:

```
-- set locale for Portuguese-Brazil
print(os.setlocale('pt_BR'))    --> pt_BR
print(3,4)                      --> 3    4
print(3.4)       --> stdin:1: malformed number near `3.4'
```

23 - The Debug Library

The debug library does not give you a debugger for Lua, but it offers all the primitives that you need for writing a debugger for Lua. For performance reasons, the official interface to these primitives is through the C API. The debug library in Lua is a way to access these functions directly within Lua code. This library declares all its functions inside the `debug` table.

Unlike the other libraries, you 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 you cannot access a local variable from outside the function that created it. Frequently, you may not want to open this library in your final version of a product, or else you may want to erase it:

```
debug = nil
```

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 you to trace the execution of a program.

An important concept in the debug library is the *stack level*. A stack level is a number that refers to a particular function that is active at that moment, that is, it has been called and has not returned yet. The function calling the debug library has level 1, the function that called it has level 2, and so on.

23.1 - Introspective Facilities

The main introspective function in the debug library is the `debug.getinfo` function. Its first parameter may be a function or a stack level. When you call `debug.getinfo(foo)` for some function `foo`, you get a table with some data about that function. The table may have the following fields:

- `source` --- Where the function was defined. If the function was defined in a string (through `loadstring`), `source` is that string. If the function was defined in a file, `source` is the file name prefixed with a `` `@' ``.
- `short_src` --- A short version of `source` (up to 60 characters), useful for error messages.
- `linedefined` --- The line of the source where the function was defined.
- `what` --- 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` --- A reasonable name for the function.
- `namewhat` --- What the previous field means. This field may be `"global"`, `"local"`, `"method"`, `"field"`, or `""` (the empty string). The empty string means that Lua did not find a name for the function.
- `nups` --- Number of upvalues of that function.
- `func` --- The function itself; see later.

When `foo` is a C function, Lua does not have much data about it. For such functions, only the fields `what`, `name`, and `namewhat` are relevant.

When you call `debug.getinfo(n)` for some number *n*, you get data about the function active at that stack level. For instance, if *n* is 1, you get data about the function doing the call. (When *n* is 0, you get data about `getinfo` itself, a C function.) If *n* is larger than the number of active functions in the stack, `debug.getinfo` returns **nil**. When you query an active function, calling `debug.getinfo` with a number, the result table has an extra field, `currentline`, with the line where the function is at that moment. Moreover, `func` has the function that is active at that level.

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 for a global variable with that value, or else looking into the code that called the function, to see how it was called. This second option works only when we call `getinfo` with a number, that is, we get information about a particular invocation.

The `getinfo` function 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. With this parameter, it 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 data, according to the following table:

`` `n' ``	selects fields `name` and `namewhat`
`` `f' ``	selects field `func`
`` `S' ``	selects fields `source`, `short_src`, `what`, and `linedefined`
`` `l' ``	selects field `currentline`
`` `u' ``	selects field `nup`

The following function illustrates the use of `debug.getinfo`. It prints a primitive traceback of the active stack:

```
    function traceback ()
```

```
  local level = 1
  while true do
    local info = debug.getinfo(level, "Sl")
    if not info then break end
    if info.what == "C" then   -- is a C function?
      print(level, "C function")
    else   -- a Lua function
      print(string.format("[%s]:%d",
                          info.short_src, info.currentline))
    end
    level = level + 1
  end
end
```

It is not difficult to improve this function, including more data from `getinfo`. Actually, the debug library offers such an improved version, `debug.traceback`. Unlike our version, `debug.traceback` does not print its result; instead, it returns a string.

23.1.1 - Accessing Local Variables

You can access the local variables of any active function by calling `getlocal`, from the `debug` library. It has two parameters: the stack level of the function you are querying and a variable index. It returns two values: the name and the current value of that 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. (You can use `debug.getinfo` to check the validity of a 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, the code

```
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

foo(10, 20)
```

will print

```
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 another `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.)

You 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 that variable. It returns the variable name, or **nil** if the variable index is out of scope.

23.1.2 - Accessing Upvalues

The debug library also allows us to access the upvalues that a Lua function uses, with getupvalue. Unlike local variables, however, a function has its upvalues even when it 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 upvalue index. Lua numbers upvalues in the order they are first referred in a function, but this order is not relevant, because a function cannot have two upvalues with the same name.

You can also update upvalues, with debug.setupvalue. As you might expect, it has three parameters: a closure, an upvalue index, and the new value. Like setlocal, it returns the name of the upvalue, or **nil** if the upvalue index is out of range.

The following code shows how we can access the value of any given variable of a calling function, given the variable name:

```
function getvarvalue (name)
  local value, found

  -- try local variables
  local i = 1
  while true do
    local n, v = debug.getlocal(2, i)
    if not n then break end
    if n == name then
      value = v
      found = true
    end
    i = i + 1
  end
  if found then return value end

  -- try upvalues
  local func = debug.getinfo(2).func
  i = 1
  while true do
    local n, v = debug.getupvalue(func, i)
    if not n then break end
    if n == name then return v end
    i = i + 1
  end

  -- not found; get global
  return getfenv(func)[name]
end
```

First, we try a local variable. If there is more than one variable with the given name, we must get the one with the highest index; so we must always go through the whole loop. If we cannot find any local variable with that name, then we try upvalues. First, we get the calling function, with debug.getinfo(2).func, and then we traverse its upvalues. Finally, if we cannot find an upvalue with that name, then we get a global variable. Notice the use of the argument 2 in the calls to debug.getlocal and debug.getinfo to access the calling function.

23.2 - Hooks

The hook mechanism of the debug library allows us to register a function that will be called at specific events as your 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; and *count* events happen after a given number of instructions. Lua calls hooks with a single argument, a string describing the event that generated the call: `"call"`, `"return"`, `"line"`, or `"count"`. Moreover, for line events, it also passes a second argument, the new line number. We can always use `debug.getinfo` to get more information inside a hook.

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 string that describes the events we want to monitor; and an 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´ ``) in 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 the number of each new line the interpreter executes:

```
debug.sethook(print, "l")
```

It 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")
```

23.3 - Profiles

Despite its name, the debug library is useful for tasks other than debugging. A common such task is profiling. For a profile with timing, it is better to use the C interface: The overhead of a Lua call for each hook is too high and usually invalidates any measure. However, for counting profiles, Lua code does a decent job. In this section, we will develop a rudimentary profiler, which lists the number of times that each function in the program is called in a run.

The main data structure of our program is a table that associates functions to their call counters and a table that associates functions to their names. The indices to these tables are the functions themselves.

```
local Counters = {}
local Names = {}
```

We could retrieve the name data 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 and increment the corresponding counter; it also collects the function name:

```
local function hook ()
  local f = debug.getinfo(2, "f").func
  if Counters[f] == nil then    -- first time `f' is called?
```

```
      Counters[f] = 1
      Names[f] = debug.getinfo(2, "Sn")
    else  -- only increment the counter
      Counters[f] = Counters[f] + 1
    end
  end
```

The next step is to run the program with this hook. We will assume that the main chunk of the program is in a file and that the user gives this file name as an argument to the profiler:

```
prompt> lua profiler main-prog
```

With this scheme, we 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
f()   -- run the main program
debug.sethook()   -- turn off the hook
```

The last step is to show the results. The next function produces a name for a function. 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 only its location. If a function is a C function, we use only its name (it has no location).

```
function getname (func)
  local n = Names[func]
  if n.what == "C" then
    return n.name
  end
  local loc = string.format("[%s]:%s",
                            n.short_src, n.linedefined)
  if n.namewhat ~= "" then
    return string.format("%s (%s)", loc, n.name)
  else
    return string.format("%s", loc)
  end
end
```

Finally, 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 Section 10.2, we get a result like this:

```
[markov.lua]:4 884723
write   10000
[markov.lua]:0 (f)     1
read    31103
sub     884722
[markov.lua]:1 (allwords)      1
[markov.lua]:20 (prefix)       894723
find    915824
[markov.lua]:26 (insert)       884723
random  10000
sethook 1
insert  884723
```

That means that the anonymous function at line 4 (which is the iterator function defined inside `allwords`) was called 884,723 times, `write` (`io.write`) was called 10,000 times, and so on.

There are several improvements that you can make to this profiler, such as to sort the output, to print better function names, and to improve the output format. Nevertheless, this basic profiler is already useful as it is and can be used as a base for more advanced tools.

24 - An Overview of the C API

Lua is an *embedded language*. That means that Lua is not a stand-alone package, but a library that can be linked with other applications so as to incorporate Lua facilities into these applications.

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? The solution to this puzzle is the Lua interpreter (the executable `lua`). This interpreter is a tiny application (with less than five hundred lines of code) that uses the Lua library to implement the stand-alone interpreter. This 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 *extension 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) and 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 extension 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 that allow C code to interact with Lua. 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. (Throughout this text, the term "function" actually means "function or macro". The API implements several facilities as macros.)

The C API follows the C *modus operandi*, which is quite different from Lua. When programming in C, we must care about type checking (and type errors), 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 your responsibility to make sure that the arguments are valid before calling a function. If you make mistakes, you can get a "segmentation fault" error or something similar, 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 you full control over all details, such as error handling, buffer sizes, and the like.

As its title says, the goal of this chapter is to give an overview of what is involved when you use Lua from C. Do not bother understanding all the details of what is going on now. Later we will fill in the details. Nevertheless, do not forget that you can find more details about specific functions in the Lua reference manual. Moreover, you can find several examples of the use of the API 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 programmers' hat. When we talk about "you", we mean you when programming in C, or you impersonated by the C code you write.

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, you can use the stack to keep intermediate results too. The stack helps to solve two impedance mismatches between Lua and C: The first is caused by Lua being garbage collected, whereas C requires explicit deallocation; the second results from the shock between dynamic typing in Lua versus the static typing of C. We will discuss the stack in more detail in Section 24.2.

24.1 - 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 primitive stand-alone interpreter as follows:

WARNING: this code is for Lua 5.0. To run it in Lua 5.1, you must change the five calls `luaopen_*(L)` to a single call to `luaL_openlibs(L)`.

```
#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 = lua_open();   /* opens Lua */
  luaopen_base(L);             /* opens the basic library */
  luaopen_table(L);            /* opens the table library */
  luaopen_io(L);               /* opens the I/O library */
  luaopen_string(L);           /* opens the string lib. */
  luaopen_math(L);             /* opens the math lib. */

  while (fgets(buff, sizeof(buff), stdin) != NULL) {
    error = luaL_loadbuffer(L, buff, strlen(buff), "line") ||
            lua_pcall(L, 0, 0, 0);
    if (error) {
      fprintf(stderr, "%s", lua_tostring(L, -1));
      lua_pop(L, 1);  /* pop error message from the stack */
    }
  }

  lua_close(L);
  return 0;
}
```

The header file `lua.h` defines the basic functions provided by Lua. That includes functions to create a new Lua environment (such as `lua_open`), to invoke Lua functions (such as `lua_pcall`), to read and write global variables in the Lua environment, to register new functions to be called by Lua, and so on. Everything defined in `lua.h` has the `lua_` prefix.

The header file `lauxlib.h` defines the functions provided by the *auxiliary library* (auxlib). All its definitions start with `luaL_` (e.g., `luaL_loadbuffer`). The auxiliary library uses the basic API provided by `lua.h` to provide a higher abstraction level; all Lua standard libraries use the auxlib. The basic API strives for economy and orthogonality, whereas auxlib strives for practicality for common tasks. Of course, it is very easy for your program to create other abstractions that it needs, too. Keep in mind that the auxlib has no access to the internals of Lua. It does its entire job through the official basic API.

The Lua library defines no global variables at all. It keeps all its state in the dynamic structure `lua_State` and a pointer to this structure is passed as an argument to all functions inside Lua.

This implementation makes Lua reentrant and ready to be used in multithreaded code.

The `lua_open` function creates a new environment (or *state*). When `lua_open` creates a fresh environment, this environment contains no predefined functions, not even `print`. To keep Lua small, all standard libraries are provided as separate packages, so that you do not have to use them if you do not need to. The header file `lualib.h` defines functions to open the libraries. The call to `luaopen_io`, for instance, creates the `io` table and registers the I/O functions (`io.read`, `io.write`, etc.) inside it.

After creating a state and populating it with the standard libraries, it is time to interpret the user input. For each line the user enters, the program first calls `luaL_loadbuffer` to compile the code. If there are no errors, the call returns zero and pushes the resulting chunk on the stack. (Remember that we will discuss this "magic" stack in detail in the next section.) Then the program calls `lua_pcall`, which pops the chunk from the stack and runs it in protected mode. Like `luaL_loadbuffer`, `lua_pcall` returns zero if there are no errors. In case of errors, both functions push an error message on the stack; we get this message with `lua_tostring` and, after printing it, we remove it from the stack with `lua_pop`.

Notice that, in case of errors, this program simply prints the error message to the standard error stream. Real error handling can be quite complex in C and how to do it depends on the nature of your application. The Lua core never writes anything directly to any output stream; it signals errors by returning error codes and error messages. Each application can handle these signals in a way most appropriate for its needs. To simplify our discussions, we will assume for now a simple error handler like the following one, which prints an error message, closes the Lua state, and exits from 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, argp);
  va_end(argp);
  lua_close(L);
  exit(EXIT_FAILURE);
}
```

Later we will discuss more about error handling in the application code.

Because you can compile Lua both as C and as C++ code, `lua.h` does not include this typical adjustment code that is present in several other C libraries:

```
#ifdef __cplusplus
extern "C" {
#endif
   ...
#ifdef __cplusplus
}
#endif
```

Therefore, if you have compiled Lua as C code (the most common case) and are using it in C++, you must include `lua.h` as follows:

```
extern "C" {
#include <lua.h>
}
```

A common trick is to create a header file `lua.hpp` with the above code and to include this new file in your C++ programs.

24.2 - The Stack

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 `a[k] = v`, both `k` and `v` can have several different types (even `a` may have different types, due to metatables). If we want to offer this operation in C, however, any `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; Lua has been designed to interface easily not only with C/C++, but also with Java, Fortran, and the like. Second, Lua does garbage collection: If we keep a Lua value in a C variable, the Lua engine has no way to know about this use; it may (wrongly) assume that this value is garbage and collect it.

Therefore, the Lua API does not define anything like a `lua_Value` type. Instead, it uses an abstract stack to exchange values between Lua and C. Each slot in this stack can hold any Lua value. Whenever you want to ask for a value from Lua (such as the value of a global variable), you call Lua, which pushes the required value on the stack. Whenever you want to pass a value to Lua, you first push the value on the stack, and then you call Lua (which will pop the value). We still need a different function to push each C type on the stack and a different function to get each value from the stack, but we avoid the combinatorial explosion. Moreover, because this stack is managed by Lua, 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_loadbuffer` 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 occasional error message there.

Lua manipulates this stack in a strict LIFO discipline (Last In, First Out; that is, always through the top). When you call Lua, it only changes the top part of the stack. Your C code has more freedom; specifically, it can inspect any element inside the stack and even insert and delete elements in any arbitrary position.

24.2.1 - Pushing Elements

The API has one push function for each Lua type that can be represented in C: `lua_pushnil` for the constant **nil**, `lua_pushnumber` for numbers (`double`), `lua_pushboolean` for booleans (integers, in C), `lua_pushlstring` for arbitrary strings (`char *`), 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, double n);
    void lua_pushlstring (lua_State *L, const char *s,
                                         size_t length);
    void lua_pushstring (lua_State *L, const char *s);
```

There are also functions to push C functions and userdata values on the stack; we will discuss them later.

Strings in Lua are not zero-terminated; in consequence, they can contain arbitrary binary data and rely on an explicit length. The official function to push a string onto the stack is `lua_pushlstring`, which requires an explicit length as an argument. For zero-terminated strings, you can use also `lua_pushstring`, which uses `strlen` to supply the string length. Lua never keeps pointers to external strings (or to any other object, except to C functions, which are always static). For any string that it has to keep, Lua either makes an internal copy or reuses one. Therefore, you can free or modify your buffer as soon as these functions return.

Whenever you push an element onto the stack, it is your 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 (this constant is defined as `LUA_MINSTACK` in `lua.h`). This is more than enough for most common uses, so usually we do not even think about that. However, some tasks may need more stack space (e.g., for calling a function with a variable number of arguments). In such cases, you may want to call

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

which checks whether the stack has enough space for your needs. (More about that later.)

24.2.2 - Querying Elements

To refer to elements in the stack, the API uses *indices*. The first element in the stack (that is, the element that was pushed first) 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, using negative indices. In that case, *-1* refers to the element at the 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 at 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 an element has a specific type, the API offers a family of functions `lua_is*`, where the `*` can be any Lua type. So, there are `lua_isnumber`, `lua_isstring`, `lua_istable`, and the like. All these functions have the same prototype:

```
    int lua_is... (lua_State *L, int index);
```

The `lua_isnumber` and `lua_isstring` functions do not check whether the value has that specific type, but whether the value can be converted to that type. For instance, any number satisfies `lua_isstring`.

There is also a function `lua_type`, which returns the type of an element in the stack. (Some of the `lua_is*` functions are actually macros that use this function.) Each type is represented by a constant defined in the header file `lua.h`: `LUA_TNIL`, `LUA_TBOOLEAN`, `LUA_TNUMBER`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, and `LUA_TTHREAD`. This function is mainly used in conjunction with a switch statement. It is also useful when we need to check for strings and numbers without coercions.

To get a value from the stack, there are the `lua_to*` functions:

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

It is OK to call them even when the given element does not have the correct type. In this case, `lua_toboolean`, `lua_tonumber` and `lua_strlen` return zero and the others return `NULL`. The zero is not useful, but ANSI C provides us with no invalid numeric value that we could use to signal errors. For the other functions, however, we frequently do not need to use the corresponding `lua_is*` function: We just call `lua_to*` and then test whether the result is not `NULL`.

The `lua_tostring` function returns a pointer to an internal copy of the string. You cannot change it (there is a `const` there to remind you). Lua ensures that this pointer is valid as long as the corresponding value is in the stack. When a C function returns, Lua clears its stack; therefore, as a rule, you should never store pointers to Lua strings outside the function that got them.

Any string that `lua_tostring` returns always has a zero at its end, but it can have other zeros inside it. The `lua_strlen` function returns the correct length of the string. In particular, assuming that the value at the top of the stack is a string, the following assertions are always valid:

```
const char *s = lua_tostring(L, -1);   /* any Lua string */
size_t l = lua_strlen(L, -1);          /* its length */
assert(s[l] == '\0');
assert(strlen(s) <= l);
```

24.2.3 - Other Stack Operations

Besides the above functions, which interchange 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_remove (lua_State *L, int index);
void  lua_insert (lua_State *L, int index);
void  lua_replace (lua_State *L, int index);
```

The `lua_gettop` function returns the number of elements in the stack, which is also the index of the top element. Notice that a negative index `-x` is equivalent to the positive index `gettop - x + 1`.

`lua_settop` sets the top (that is, the number of elements in the stack) to a specific value. If the previous top was higher than the new one, the top values are discarded. Otherwise, the function pushes **nil**s on the stack to get the given size. As a particular case, `lua_settop(L, 0)` empties the stack. You can also use negative indices with `lua_settop`; that will set the top element to the given index. 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 `lua_pushvalue` function pushes on the top of the stack a copy of the element at the given index; `lua_remove` removes the element at the given index, shifting down all elements on top of that position to fill in the gap; `lua_insert` moves the top element into the given position, shifting up all elements on top of that position to open space; finally, `lua_replace` pops a value from the top and sets it as the value of the given index, without moving anything. Notice that the following operations have no effect on the stack:

```
lua_settop(L, -1);  /* set top to its current value */
lua_insert(L, -1);  /* move top element to the top */
```

To illustrate the use of those functions, here is a useful helper function that dumps the entire content of the stack:

```
static void stackDump (lua_State *L) {
  int i;
  int top = lua_gettop(L);
  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 other values (tables, functions, etc.) it prints only their types (`lua_typename` converts a type code to a type name).

The following program uses `stackDump` to further illustrate the manipulation of the API stack:

```
#include <stdio.h>
#include <lua.h>

static void stackDump (lua_State *L) {
  ...
}

int main (void) {
  lua_State *L = lua_open();
  lua_pushboolean(L, 1); lua_pushnumber(L, 10);
  lua_pushnil(L); lua_pushstring(L, "hello");
  stackDump(L);
                 /* true  10  nil  `hello'  */

  lua_pushvalue(L, -4); stackDump(L);
                 /* true  10  nil  `hello'  true  */

  lua_replace(L, 3); stackDump(L);
                 /* true  10  true  `hello'  */

  lua_settop(L, 6); stackDump(L);
                 /* true  10  true  `hello'  nil  nil  */
```

```
      lua_remove(L, -3); stackDump(L);
                       /* true  10  true  nil  nil  */

      lua_settop(L, -5); stackDump(L);
                       /* true  */

      lua_close(L);
      return 0;
    }
```

24.3 - Error Handling with the C API

Unlike C++ or Java, the C language does not offer an exception handling mechanism. To ameliorate this difficulty, Lua uses the `setjmp` facility from C, which results in a mechanism similar to exception handling. (If you compile Lua with C++, it is not difficult to change the code so that it uses real exceptions instead.)

All structures in Lua are dynamic: They grow as needed, and eventually shrink again when possible. That means that the possibility of a memory-allocation failure is pervasive in Lua. Almost any operation may face this eventuality. Instead of using error codes for each operation in its API, Lua uses exceptions to signal these errors. That means that almost all API functions may throw an error (that is, call `longjmp`) instead of returning.

When we write library code (that is, C functions to be called from Lua), the use of long jumps is almost as convenient as a real exception-handling facility, because Lua catches any occasional error. When we write application code (that is, C code that calls Lua), however, we must provide a way to catch those errors.

24.3.1 - Error Handling in Application Code

Typically, your application code runs *unprotected*. Because its code is not called by Lua, Lua cannot set an appropriate context to catch errors (that is, it cannot call `setjmp`). In such environments, when Lua faces an error like "not enough memory", there is not much that it can do. It calls a panic function and, if the function returns, exits the application. (You can set your own panic function with the `lua_atpanic` function.)

Not all API functions throw exceptions. The functions `lua_open`, `lua_close`, `lua_pcall`, and `lua_load` are all safe. Moreover, most other functions can only throw an exception in case of memory-allocation failure: For instance, `luaL_loadfile` fails if there is not enough memory for a copy of the file name. Several programs have nothing to do when they run out of memory, so they may ignore these exceptions. For those programs, if Lua runs out of memory, it is OK to panic.

If you do not want your application to exit, even in case of a memory-allocation failure, then you must run your code in *protected mode*. Most (or all) of your Lua code typically runs through a call to `lua_pcall`; therefore, it runs in protected mode. Even in case of memory-allocation failure, `lua_pcall` returns an error code, leaving the interpreter in a consistent state. If you also want to protect all your C code that interacts with Lua, then you can use `lua_cpcall`. (See the reference manual for further details of this function; see file `lua.c` in the Lua distribution for an example of its use.)

24.3.2 - Error Handling in Library Code

Lua is a *safe* language. That means that, no matter what you write, no matter how wrong it is, you 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 only be explained in terms of the underling hardware and error positions are given as a program counter.

Whenever you add new C functions to Lua, you can break that safety. For instance, a function like `poke`, which stores an arbitrary byte at an arbitrary memory address, can cause all sorts of memory corruption. You must strive to ensure that your add-ons are safe to Lua and provide good error handling.

As we discussed earlier, each C program has its own way to handle errors. When you write library functions for Lua, however, there is a standard way to handle errors. Whenever a C function detects an error, it simply calls `lua_error`, (or better yet `luaL_error`, which formats the error message and then calls `lua_error`). The `lua_error` function clears whatever needs to be cleared in Lua and jumps back to the `lua_pcall` that originated that execution, passing along the error message.

25 - 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 more complex tasks.

As our first task, let us imagine a simple configuration scenario: Your C program (let us call it `pp`) has a window and you want the user to be able to specify the initial window size. Clearly, for such simple tasks, there are several options simpler than using Lua, such as environment variables or files with name-value pairs. But even using a simple text file, you have to parse it somehow; so, you 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 next lines:

```
-- configuration file for program `pp'
-- define window size
width = 200
height = 300
```

Now, you 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 following function does the job:

```
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

void load (char *filename, int *width, int *height) {
  lua_State *L = lua_open();
  luaopen_base(L);
  luaopen_io(L);
  luaopen_string(L);
  luaopen_math(L);

  if (luaL_loadfile(L, filename) || lua_pcall(L, 0, 0, 0))
    error(L, "cannot run configuration file: %s",
             lua_tostring(L, -1));
```

```
      lua_getglobal(L, "width");
      lua_getglobal(L, "height");
      if (!lua_isnumber(L, -2))
        error(L, "`width' should be a number\n");
      if (!lua_isnumber(L, -1))
        error(L, "`height' should be a number\n");
      *width = (int)lua_tonumber(L, -2);
      *height = (int)lua_tonumber(L, -1);

      lua_close(L);
    }
```

First, it opens the Lua package and loads the standard libraries (they are optional, but usually it is a good idea to have them around). Then, it uses `luaL_loadfile` to load the chunk from file `filename` and calls `lua_pcall` to run it. In case of errors in any of these functions (e.g., a syntax error in your configuration file), the call returns a non-zero error code and pushes the error message onto the stack. As usual, our program uses `lua_tostring` with index *-1* to get the message from the top of the stack. (We defined the `error` function in Section 24.1.)

After running the chunk, the program needs to get the values of the global variables. For that, it calls twice `lua_getglobal`, whose single parameter (besides the omnipresent `lua_State`) is the variable name. Each call pushes the corresponding global value onto the top of the stack, so that the width will be at index *-2* and the height at index *-1* (at the top). (Because the stack was previously empty, you could also index from the bottom, using *1* from the first value and *2* from the second. By indexing from the top, however, your code would work even if the stack was not empty.) Next, our example uses `lua_isnumber` to check whether each value is numeric. It then uses `lua_tonumber` to convert such values to `double` and C does the coercion to `int`. Finally, it closes the Lua state and returns.

Is it worth using Lua? As I said before, for such simple tasks, a simple file with only two numbers in it would be much easier to use than Lua. Even so, the use of Lua brings some advantages. First, Lua handles all syntax details (and errors) for you; your configuration file can even have comments! Second, the user is already able to do more 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 for program `pp'
    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, your C application works without changes.

A final reason for using Lua is that now it is easy to add new configuration facilities to your program; this easiness creates an attitude that results in programs that are more flexible.

25.1 - 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, those numbers are integers in some range like *[0,255]*. In Lua, because all numbers are real, we can 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 for program `pp'
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 = {r=0.30, g=0.10, b=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 = {r=0, g=0, b=1}
...
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 valid color table");

red = getfield("r");
green = getfield("g");
blue = getfield("b");
```

As usual, we first get the value of the global variable `background` and ensure that it is a table. Next, we use `getfield` to get each color component. This function is not part of the API; we must define it, as follows:

```
#define MAX_COLOR       255

/* assume that table is on the stack top */
int getfield (const char *key) {
  int result;
  lua_pushstring(L, key);
  lua_gettable(L, -2);  /* get background[key] */
  if (!lua_isnumber(L, -1))
    error(L, "invalid component in background color");
  result = (int)lua_tonumber(L, -1) * MAX_COLOR;
  lua_pop(L, 1);  /* remove number */
  return result;
}
```

Again, we face the problem of polymorphism: There are potentially many versions of `getfield` functions, varying the key type, value type, error handling, etc. The Lua API offers a single function, `lua_gettable`. It receives the position of the table in the stack, pops the key from the stack, and pushes the corresponding value. Our private `getfield` assumes that the table is on the top of the stack; so, after pushing the key (`lua_pushstring`), the table will be at index *-2*. Before returning, `getfield` pops the retrieved value from the stack, to leave 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},
  {"BLACK",     0, 0, 0},
  ...
  {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 = {r=1, g=1, b=1}
RED   = {r=1, g=0, b=0}
...
```

The only difference from these user-defined colors is that the application defines these colors in C, before running the user script.

To set the table fields, we define an auxiliary function, `setfield`; it pushes the index and the field value on the stack, and then calls `lua_settable`:

```
/* assume that table is at the top */
void setfield (const char *index, int value) {
  lua_pushstring(L, index);
  lua_pushnumber(L, (double)value/MAX_COLOR);
  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 receives the table index as an argument and pops the key and the value. The `setfield` function assumes that before the call the table is at the top of the stack (index *-1*); after pushing the index and the value, the table will be at index *-3*.

The `setcolor` function defines a single color. It must create a table, set the appropriate fields, and assign this table to the corresponding global variable:

```
void setcolor (struct ColorTable *ct) {
  lua_newtable(L);              /* creates a table */
  setfield("r", ct->red);       /* table.r = ct->r */
  setfield("g", ct->green);     /* table.g = ct->g */
  setfield("b", ct->blue);      /* table.b = ct->b */
  lua_setglobal(L, ct->name);   /* `name' = table */
}
```

The `lua_newtable` function creates an empty table and pushes it on the stack; the `setfield` calls set the table fields; finally, `lua_setglobal` pops the table and sets it as the value of the global with the given name.

With those previous functions, the following loop will register all colors in the application's global environment:

```
int i = 0;
while (colortable[i].name != NULL)
  setcolor(&colortable[i++]);
```

Remember that the application must execute this loop before running the user script.

There is another option for implementing named colors. 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 implementation, 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 has to test whether the value has type string, and then look up the string in the color table:

```
lua_getglobal(L, "background");
if (lua_isstring(L, -1)) {
  const char *name = lua_tostring(L, -1);
  int i = 0;
  while (colortable[i].name != NULL &&
         strcmp(colorname, colortable[i].name) != 0)
    i++;
  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 = getfield("r");
  green = getfield("g");
  blue = getfield("b");
} else
    error(L, "invalid value for `background'");
```

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, global variables do not need declarations, so Lua does not signal any error when a user misspells a color name. If the user writes `WITE` instead of `WHITE`, the `background` variable receives **nil** (the value of `WITE`, a variable not initialized), and that is all that the application knows: that `background` is **nil**. There is no other information about what is wrong. With strings, on the other hand, the value of `background` would be the misspelled string; so, the application can add that 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 odd to register hundreds of colors (and to create hundreds of tables and global variables) only for the user to choose a few. With strings, you avoid this overhead.

25.2 - Calling Lua Functions

A great strength of Lua is that a configuration file can define functions to be called by the application. For instance, you can write an application to plot the graph of a function and use Lua to define the functions to be plotted.

The API protocol to call a function is simple: First, you push the function to be called; second, you push the arguments to the call; then you use `lua_pcall` to do the actual call; finally, you pop the results from the stack.

As an example, let us assume that our configuration file has a function like

```
function f (x, y)
  return (x^2 * math.sin(y))/(1 - x)
end
```

and you want to evaluate, in C, `z = f(x, y)` for given `x` and `y`. Assuming that you have already opened the Lua library and run the configuration file, you can encapsulate this call in the following C function:

```
    /* call a function `f' defined in Lua */
    double f (double x, double y) {
      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) != 0)
        error(L, "error running function `f': %s",
                 lua_tostring(L, -1));

      /* retrieve result */
      if (!lua_isnumber(L, -1))
        error(L, "function `f' must return a number");
      z = lua_tonumber(L, -1);
      lua_pop(L, 1);  /* pop returned value */
      return z;
    }
```

You call `lua_pcall` with the number of arguments you are passing and the number of results you want. The fourth argument indicates an error-handling function; we will discuss it in a moment. As in a Lua assignment, `lua_pcall` adjusts the actual number of results to what you have asked for, pushing **nil**s or discarding extra values as needed. Before pushing the results, `lua_pcall` removes from the stack the function and its arguments. If a function returns multiple results, the first result is pushed first; so, if there are *n* results, the first one will be at index *-n* and the last at index *-1*.

If there is any error while `lua_pcall` is running, `lua_pcall` returns a value different from zero; 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 error handler function, if there is one. To specify an error handler function, we use the last argument of `lua_pcall`. A zero means no error handler function; that is, the final error message is the original message. Otherwise, that argument should be the index in the stack where the error handler function is located. Notice that, in such cases, the handler must be pushed in the stack before the function to be called and its arguments.

For normal errors, `lua_pcall` returns the error code `LUA_ERRRUN`. Two special kinds of errors deserve different codes, because they never run the error handler. The first kind is a memory allocation error. For such errors, `lua_pcall` always returns `LUA_ERRMEM`. The second kind is an error while Lua is running the error handler itself. In that case it is of little use to call the error handler again, so `lua_pcall` returns immediately with a code `LUA_ERRERR`.

25.3 - A Generic Call Function

As a more advanced example, we will build a wrapper for calling Lua functions, using the `vararg` facility in C. Our wrapper function (let us call it `call_va`) receives the name of the 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 previous example simply as

```
    call_va("f", "dd>d", x, y, &z);
```

where 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.

```
    #include <stdarg.h>

    void call_va (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);  /* get function */

      /* push arguments */
      narg = 0;
      while (*sig) {  /* push arguments */
        switch (*sig++) {

          case 'd':  /* double argument */
            lua_pushnumber(L, va_arg(vl, double));
            break;

          case 'i':  /* int argument */
            lua_pushnumber(L, va_arg(vl, int));
            break;

          case 's':  /* string argument */
            lua_pushstring(L, va_arg(vl, char *));
            break;

          case '>':
            goto endwhile;

          default:
            error(L, "invalid option (%c)", *(sig - 1));
        }
        narg++;
        luaL_checkstack(L, 1, "too many arguments");
      } endwhile:

      /* do the call */
      nres = strlen(sig);  /* number of expected results */
      if (lua_pcall(L, narg, nres, 0) != 0)  /* do the call */
        error(L, "error running function `%s': %s",
                 func, lua_tostring(L, -1));

      /* retrieve results */
      nres = -nres;  /* stack index of first result */
      while (*sig) {  /* get results */
        switch (*sig++) {

          case 'd':  /* double result */
            if (!lua_isnumber(L, nres))
              error(L, "wrong result type");
            *va_arg(vl, double *) = lua_tonumber(L, nres);
            break;

          case 'i':  /* int result */
            if (!lua_isnumber(L, nres))
              error(L, "wrong result type");
            *va_arg(vl, int *) = (int)lua_tonumber(L, nres);
            break;

          case 's':  /* string result */
            if (!lua_isstring(L, nres))
              error(L, "wrong result type");
            *va_arg(vl, const char **) = lua_tostring(L, nres);
            break;

          default:
            error(L, "invalid option (%c)", *(sig - 1));
        }
        nres++;
      }
      va_end(vl);
    }
```

Despite its generality, this function follows the same steps of our previous example: It pushes the function, pushes the arguments, does the call, and gets the results. 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 any occasional error. Second, because it pushes an arbitrary number of arguments, it must check the stack space. Third, because the function may return strings, `call_va` cannot pop the results from the stack. It is up to the caller to pop them, after it finishes using occasional string results (or after copying them to other buffers).

26 - Calling C from Lua

One of the basic means for extending Lua is for the application to *register* new C functions into Lua.

When we say that Lua can call C functions, this does not mean that Lua can call any C function. (There are packages that allow Lua to call any C function, but they are neither portable nor robust.) As we saw previously, when C calls a Lua function, it must follow a simple protocol to pass the arguments and to get the results. Similarly, for a C function to be called from Lua, it must follow a protocol to get its arguments and to return its results. Moreover, for a C function to be called from Lua, we must register it, 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. To distinguish the results from other values on the stack, the function returns (in C) the number of results it is leaving on the stack. An important concept 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.

26.1 - 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 (a more professional implementation should check whether its argument is a 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 as `lua_CFunction` in `lua.h`:

```
    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 (in C) an integer with the number of values it is returning (in Lua). Therefore, the function does not need to clear the stack before pushing its results. After it returns, Lua automatically removes whatever is in the stack below the results.

Before we can use this function from Lua, we must register it. We do this magic with `lua_pushcfunction`: It gets a pointer to a C function and creates a value of type `"function"` to represent this function inside Lua. A quick-and-dirty way to test `l_sin` is to put its code directly into the file `lua.c` and add the following lines right after the call to `lua_open`:

```
    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, you rebuild your Lua executable; then you can use the new function `mysin` in your Lua programs. In the next section, we will discuss better ways to link new C functions with Lua.

For a more professional sine function, we must check the type of its argument. Here, the auxiliary library helps us. The `luaL_checknumber` function checks whether a given argument is a number: In case of errors, it throws an informative error message; otherwise, it returns the number. The modification in 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 the message

```
    bad argument #1 to `mysin' (number expected, got string)
```

Notice how `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 ANSI C does not have functions for this job. Here, we will assume that we have a POSIX compliant system. Our function, `dir`, gets as argument a string with the directory path and returns an array with the directory entries. For instance, a call `dir("/home/lua")` may return the table `{".", "..", "src", "bin", "lib"}`. In case of errors, the function returns **nil** plus a string with the error message.

```
    #include <dirent.h>
    #include <errno.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 and ... */
        lua_pushstring(L, strerror(errno));  /* error message */
        return 2;  /* number of results */
      }

      /* create result table */
      lua_newtable(L);
      i = 1;
      while ((entry = readdir(dir)) != NULL) {
        lua_pushnumber(L, i++);  /* push key */
        lua_pushstring(L, entry->d_name);  /* push value */
        lua_settable(L, -3);
      }

      closedir(dir);
      return 1;  /* table is already on top */
    }
```

The `luaL_checkstring` function, from the auxiliary library, is the equivalent of `luaL_checknumber` for strings.

(In extreme conditions, that implementation of `l_dir` may cause a small memory leak. Three of the Lua functions it calls can fail due to insufficient memory: `lua_newtable`, `lua_pushstring`, and `lua_settable`. If any of these calls fails, it will raise an error and interrupt `l_dir`, which therefore will not call `closedir`. As we discussed earlier, on most programs this is not a big problem: If the program runs out of memory, the best it can do is to shut down anyway. Nevertheless, in Chapter 29 we will see an alternative implementation for a directory function that avoids this problem.)

26.2 - C Libraries

A Lua library is a chunk that defines several Lua functions and stores them in appropriate places, typically as entries in a table. A C library for Lua mimics this behavior. Besides the definition of its C functions, it must also define a special function that corresponds to the main chunk of a Lua library. Once called, this function registers all C functions of the library and stores them in appropriate places. Like a Lua main chunk, it also initializes anything else that needs initialization in the library.

Lua "sees" C functions through this registration process. Once a C function is represented and stored in Lua, a Lua program calls it through 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 library has one single public (extern) function, which is the function that opens the library. All other functions may be private, declared as `static` in C.

When you extend Lua with C functions, it is a good idea to design your code as a C library, even when you want to register only one C function: Sooner or later (usually sooner) you will need other functions. As usual, the auxiliary library offers a helper function for this job. The `luaL_openlib` function receives a list of C functions and their respective names and registers all of them inside a table with the library name. As an example, suppose we want to create a library with the `l_dir` function 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 and their respective names. This array has elements of type `luaL_reg`, which is a structure with two fields: 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. Notice that the last pair in the array must be `{NULL, NULL}`, to signal its end. Finally, we declare a main function, using `luaL_openlib`:

```
int luaopen_mylib (lua_State *L) {
  luaL_openlib(L, "mylib", mylib, 0);
  return 1;
}
```

The second argument to `luaL_openlib` is the library name. This function creates (or reuses) a table with the given name, and fills it with the pairs name-function specified by the array `mylib`. The `luaL_openlib` function also allows us to register common upvalues for all functions in a library. For now, we are not using upvalues, so the last argument in the call is zero. When it returns, `luaL_openlib` leaves on the stack the table wherein it opened the library. The `luaopen_mylib` function returns 1 to return this value to Lua. (As with Lua libraries, this return is optional, because the library is already assigned to a global variable. Again, like in Lua libraries, it costs nothing, and may be useful occasionally.)

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. (Remember the discussion about dynamic linking in Section 8.2.) In this case, you must create a dynamic library with your code (a `.dll` file in Windows, a `.so` file in Linux). After that, you can load your library directly from within Lua, with `loadlib`. The call

```
mylib = loadlib("fullname-of-your-library", "luaopen_mylib")
```

transforms the `luaopen_mylib` function into a C function inside Lua and assigns this function to `mylib`. (That explains why `luaopen_mylib` must have the same prototype as any other C function.) Next, the call `mylib()` runs `luaopen_mylib`, opening the library.

If your interpreter does not support dynamic linking, then you have to recompile Lua with your new library. Besides that, you need some way to tell the stand-alone interpreter that it should open this library when it opens a new state. Some macros facilitate this task. First, you must create a header file (let us call it `mylib.h`) with the following content:

```
int luaopen_mylib (lua_State *L);

#define LUA_EXTRALIBS { "mylib", luaopen_mylib },
```

The first line declares the open function. The next line defines the macro `LUA_EXTRALIBS` as a new entry in the array of functions that the interpreter calls when it creates a new state. (This array has type `struct luaL_reg[]`, so we need to put a name there.)

To include this header file in the interpreter, you can define the macro `LUA_USERCONFIG` in your compiler options. For a command-line compiler, you typically must add an option like

```
    -DLUA_USERCONFIG=\"mylib.h\"
```

(The backslashes protect the quotes from the shell; those quotes are necessary in C when we specify an include file name.) In an integrated development environment, you must add something similar in the project settings. Then, when you re-compile `lua.c`, it includes `mylib.h`, and therefore uses the new definition of `LUA_EXTRALIBS` in the list of libraries to open.

27 - 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 special mechanisms for array manipulation, for string manipulation, and for storing Lua values in C.

27.1 - Array Manipulation

"Array", in Lua, is just a name for a table used in a specific way. We can manipulate arrays using the same functions we use to manipulate tables, namely `lua_settable` and `lua_gettable`. However, contrary to the general philosophy of Lua, *economy and simplicity*, the API provides special functions for array manipulation. The reason for that is performance: Frequently we have an array access operation inside the inner loop of an algorithm (e.g., sorting), so that any performance gain in this operation can have a big impact on the overall performance of the function.

The functions that the API provides for array manipulation are

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

The description of `lua_rawgeti` and `lua_rawseti` is a little confusing, as it involves two indices: `index` refers to where the table is in the stack; `key` refers to where the element is in the table. The call `lua_rawgeti(L, t, key)` is equivalent to the sequence

```
    lua_pushnumber(L, key);
    lua_rawget(L, t);
```

when `t` is positive (otherwise, you must compensate for the new item in the stack). The call `lua_rawseti(L, t, key)` (again for `t` positive) is equivalent to

```
    lua_pushnumber(L, key);
    lua_insert(L, -2);  /* put `key' below previous value */
    lua_rawset(L, t);
```

Note that both functions use raw operations. They are faster and, anyway, tables used as arrays seldom use metamethods.

As a concrete example of the use of these functions, we could rewrite the loop body from our previous `l_dir` function from

```
        lua_pushnumber(L, i++);  /* key */
        lua_pushstring(L, entry->d_name);  /* value */
        lua_settable(L, -3);
```

to

```
        lua_pushstring(L, entry->d_name);  /* value */
        lua_rawseti(L, -2, i++);  /* set table at key `i' */
```

As a more complete example, the following code implements the map function: It applies a given function to all elements of an array, replacing each element by the result of the call.

```
    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_getn(L, 1);  /* get size of table */

      for (i=1; i<=n; i++) {
        lua_pushvalue(L, 2);   /* push f */
        lua_rawgeti(L, 1, i);  /* push t[i] */
        lua_call(L, 1, 1);     /* call f(t[i]) */
        lua_rawseti(L, 1, i);  /* t[i] = result */
      }

      return 0;  /* no results */
    }
```

This example introduces three new functions. The `luaL_checktype` function (from `lauxlib.h`) ensures that a given argument has a given type; otherwise, it raises an error. The `luaL_getn` function gets the size of the array at the given index (`table.getn` calls `luaL_getn` to do its job). The `lua_call` function does an unprotected call. It is similar to `lua_pcall`, but in case of errors it throws the error, instead of returning an error code. When you are writing the main code in an application, you should not use `lua_call`, because you want to catch any errors. When you are writing functions, however, it is usually a good idea to use `lua_call`; if there is an error, just leave it to someone that cares about it.

27.2 - 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 accessing 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 overflow, and the like. Nevertheless, the Lua API provides some functions to help with those 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 you want to pass to Lua a substring of a string `s` ranging from position `i` to `j` (inclusive), all you have to do is

```
    lua_pushlstring(L, s+i, j-i+1);
```

As an example, suppose you 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,,there", ",")
```

should return the table `{"hi", "", "there"}`. We could write a simple implementation as follows. It needs no extra buffers and puts no constraints on the size of the strings it can handle.

```
static int l_split (lua_State *L) {
  const char *s = luaL_checkstring(L, 1);
  const char *sep = luaL_checkstring(L, 2);
  const char *e;
  int i = 1;

  lua_newtable(L);  /* result */

  /* repeat for each separator */
  while ((e = strchr(s, *sep)) != NULL) {
    lua_pushlstring(L, s, e-s);  /* push substring */
    lua_rawseti(L, -2, i++);
    s = e + 1;  /* skip separator */
  }

  /* push last substring */
  lua_pushstring(L, s);
  lua_rawseti(L, -2, i);

  return 1;  /* return the table */
}
```

To concatenate strings, Lua provides a specific function in its API, called `lua_concat`. It is equivalent to the `..` 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 `n` values at the top of the stack and leave the result on the top.

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, you do not need to provide a buffer. Lua dynamically creates the string for you, as large as it needs to be. There are no worries about buffer overflow and the like. The function pushes the resulting string on the stack and returns a pointer to it. Currently, this function accepts only the directives `%%` (for the character `` `%' ``), `%s` (for strings), `%d` (for integers), `%f` (for Lua numbers, that is, doubles), and `%c` (accepts an integer and formats it as a character). It does not accept any options (such as width or precision).

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 Section 11.6. Instead, we can use the buffer facilities provided by the auxiliary library. Auxlib implements these buffers in two levels. The first level is similar to buffers in I/O operations: It collects small strings (or individual characters) in a local buffer and passes them to Lua (with `lua_pushlstring`) when the buffer fills up. The second level uses `lua_concat` and a variant of the stack algorithm that we saw in Section 11.6 to concatenate the results of multiple buffer flushes.

To describe the buffer facilities from auxlib in more detail, let us see a simple example of its use. The next code shows the implementation of `string.upper`, right from the file `lstrlib.c`:

```
static int str_upper (lua_State *L) {
  size_t l;
  size_t i;
  luaL_Buffer b;
  const char *s = luaL_checklstr(L, 1, &l);
  luaL_buffinit(L, &b);
```

```
      for (i=0; i<l; i++)
        luaL_putchar(&b, toupper((unsigned char)(s[i])));
      luaL_pushresult(&b);
      return 1;
    }
```

The first step for using a buffer from auxlib is to declare a variable with type `luaL_Buffer`, and then to initialize it with a call to `luaL_buffinit`. After the initialization, the buffer keeps a copy of the state `L`, so we do not need to pass it when calling other functions that manipulate the buffer. The macro `luaL_putchar` puts a single character into the buffer. Auxlib also offers `luaL_addlstring`, to put a string with an explicit length into the buffer, and `luaL_addstring`, to put a zero-terminated string. Finally, `luaL_pushresult` flushes the buffer and leaves the final string on the top of the stack. The prototypes of those functions are as follows:

```
    void luaL_buffinit (lua_State *L, luaL_Buffer *B);
    void luaL_putchar (luaL_Buffer *B, char c);
    void luaL_addlstring (luaL_Buffer *B, const char *s,
                                          size_t l);
    void luaL_addstring (luaL_Buffer *B, const char *s);
    void luaL_pushresult (luaL_Buffer *B);
```

Using these functions, we do not have to worry about buffer allocation, overflows, and other such details. As we saw, the concatenation algorithm is quite efficient. The `str_upper` function handles huge strings (more than 1 MB) without any problem.

When you use the auxlib buffer, you have to worry about one detail. As you put things into the buffer, it keeps some intermediate results in the Lua stack. Therefore, you cannot assume that the stack top will remain where it was before you started using the buffer. Moreover, although you can use the stack for other tasks while using a buffer (even to build another buffer), the push/pop count for these uses must be balanced every time you access the buffer. There is one obvious situation where this restriction is too severe, namely when you want to put into the buffer a string returned from Lua. In such cases, you cannot pop the string before adding it to the buffer, because you should never use a string from Lua after popping it from the stack; but also you cannot add the string to the buffer before popping it, because then the stack would be in the wrong level. In other words, you cannot do something like this:

```
    luaL_addstring(&b, lua_tostring(L, 1));   /* BAD CODE */
```

Because this is a common situation, auxlib provides a special function to add the value on the top of the stack into the buffer:

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

Of course, it is an error to call this function if the value on the top is not a string or a number

27.3 - 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 or static variables for that need. When you are programming library functions for Lua, however, global and static variables are not a good approach. First, you cannot store a generic Lua value in a C variable. Second, a library that uses such variables cannot be used in multiple Lua states.

An alternative approach is to store such values into Lua global variables. This approach solves the two previous problems. Lua global variables store any Lua value and each independent state has its own independent set of global variables. However, this is not always a satisfactory solution, because Lua code can tamper with those global variables and therefore compromise the integrity of C data. To avoid this problem, Lua offers a separate table, called the *registry*, that C code can freely use, but Lua code cannot access.

27.3.1 - The Registry

The registry is always located at a *pseudo-index*, whose value is defined by `LUA_REGISTRYINDEX`. A pseudo-index is like an index into the stack, except that its associated value is not in the stack. Most functions in the Lua API 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, you can use the following code:

```
    lua_pushstring(L, "Key");
    lua_gettable(L, LUA_REGISTRYINDEX);
```

The registry is a regular Lua table. As such, you can index it with any Lua value but **nil**. However, because all C libraries share the same registry, you must choose with care what values you use as keys, to avoid collisions. A bulletproof method is to use as key the address of a static variable in your code: The C link editor ensures that this key is unique among all libraries. To use this option, you need the function `lua_pushlightuserdata`, which pushes on the Lua stack a value representing a C pointer. The following code shows how to store and retrieve a number from the registry using this method:

```
    /* variable with an unique address */
    static const char Key = 'k';

    /* store a number */
    lua_pushlightuserdata(L, (void *)&Key);  /* push address */
    lua_pushnumber(L, myNumber);  /* push value */
    /* registry[&Key] = myNumber */
    lua_settable(L, LUA_REGISTRYINDEX);

    /* retrieve a number */
    lua_pushlightuserdata(L, (void *)&Key);  /* push address */
    lua_gettable(L, LUA_REGISTRYINDEX);  /* retrieve value */
    myNumber = lua_tonumber(L, -1);  /* convert to number */
```

We will discuss light userdata in more detail in Section 28.5.

Of course, you can also use strings as keys into the registry, as long as you choose unique names. String keys are particularly useful when you want to allow other independent libraries to access your data, because all they need to know is the key name. For such keys, there is no bulletproof method of choosing names, but there are some good practices, such as avoiding common names and prefixing your names with the library name or something like it. Prefixes like `lua` or `lualib` are not good choices. Another option is to use a *universal unique identifier* (`uuid`), as most systems now have programs to generate such identifiers (e.g., `uuidgen` in Linux). An `uuid` is a 128-bit number (written in hexadecimal to form a string) that is generated by a combination of the host IP address, a time stamp, and a random component, so that it is assuredly different from any other `uuid`.

27.3.2 - References

You should never use numbers as keys in the registry, because such keys are reserved for the *reference system*. This system is composed by a couple of functions in the auxiliary library that allow you to store values in the registry without worrying about how to create unique names. (Actually, those functions can act on any table, but they are typically used with the registry.)

The call

```
int r = luaL_ref(L, LUA_REGISTRYINDEX);
```

pops a value from the stack, stores it into the registry with a fresh integer key, and returns that 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 `r` onto the stack, we simply write

```
lua_rawgeti(L, LUA_REGISTRYINDEX, r);
```

Finally, to release both the value and the reference, we call

```
luaL_unref(L, LUA_REGISTRYINDEX, r);
```

After this call, `luaL_ref` may return the value in `r` again as a new reference.

The reference system treats **nil** as a special case. Whenever you call `luaL_ref` for a nil value, it does not create a new reference, but instead returns the constant reference `LUA_REFNIL`. The call

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

has no effect, whereas

```
lua_rawgeti(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

pushes a **nil**, as expected.

The reference system also defines the constant `LUA_NOREF`, which is an integer different from any valid reference. It is useful to mark references as invalid. As with `LUA_REFNIL`, any attempt to retrieve `LUA_NOREF` returns **nil** and any attempt to release it has no effect.

27.3.3 - Upvalues

While the registry implements global values, the *upvalue* mechanism implements an equivalent of C static variables, which are visible only inside a particular function. Every time you create a new C function in Lua, you can associate with it any number of upvalues; each upvalue can hold a single Lua value. Later, when the function is called, 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*. Remember that, in Lua code, a closure is a function that uses local variables from an outer function. A C closure is a C approximation to a Lua closure. One interesting fact about closures is that you can create different closures using the same function code, but with different upvalues.

To see a simple example, let us create a `newCounter` function in C. (We already defined this same function in Lua, in Section 6.1.) This function is a factory function: It returns a new counter function each time it is called. Although all counters share the same C code, each one keeps its own independent counter. The factory function is like this:

```
/* forward declaration */
static int counter (lua_State *L);

int newCounter (lua_State *L) {
  lua_pushnumber(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 the number 0 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) {
  double val = lua_tonumber(L, lua_upvalueindex(1));
  lua_pushnumber(L, ++val);  /* new value */
  lua_pushvalue(L, -1);  /* duplicate it */
  lua_replace(L, lua_upvalueindex(1));  /* update upvalue */
  return 1;  /* return new value */
}
```

Here, the key function is `lua_upvalueindex` (which is actually a macro), which produces the pseudo-index of an upvalue. Again, this pseudo-index is like any stack index, except that it does not live in the stack. The expression `lua_upvalueindex(1)` refers to the index of the first upvalue of the function. So, the `lua_tonumber` in function `counter` retrieves the current value of the first (and only) upvalue as a number. Then, function `counter` pushes the new value `++val`, makes a copy of it, and uses one of the copies to replace the upvalue with the new value. Finally, it returns the other copy as its return value.

Unlike Lua closures, C closures cannot share upvalues: Each closure has its own independent set. However, we can set the upvalues of different functions to refer to a common table, so that this table becomes a common place where those functions can share data.

28 - 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 that we will extend through the chapter with metamethods and other goodies.

Our example is a quite simple type: numeric arrays. The main motivation for this example is that it does not involve complex algorithms, so we can concentrate on API issues. Despite its simplicity, this type is useful for some applications. Usually, we do not need external arrays in Lua; hash tables do the job quite well. But hash tables can be memory-hungry for huge arrays, as for each entry they must store a generic value, a link address, plus some extra space to grow. A straight implementation in C, where we store the numeric values without any extra space, uses less than 50% of the memory used by a hash table.

We will represent our arrays with the following structure:

```
typedef struct NumArray {
  int size;
  double values[1];  /* variable part */
} NumArray;
```

We declare the array `values` with size 1 only as a placeholder, because C does not allow an array with size 0; we will define the actual size by the space we allocate for the array. For an array with `n` elements, we need `sizeof(NumArray) + (n-1)*sizeof(double)` bytes. (We subtract one from `n` because the original structure already includes space for one element.)

28.1 - Userdata

Our first concern is how to represent array values in Lua. Lua provides a basic type specifically for this: *userdata*. A userdatum offers a raw memory area with no predefined operations in Lua.

The Lua API offers the following function to create a userdatum:

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

The `lua_newuserdata` function allocates a block of memory with the given size, pushes the corresponding userdatum on the stack, and returns the block address. If for some reason you need to allocate memory by other means, it is very easy to create a userdatum 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 the next chapter.

Using `lua_newuserdata`, the function that creates new arrays is as follows:

```
static int newarray (lua_State *L) {
  int n = luaL_checkint(L, 1);
  size_t nbytes = sizeof(NumArray) + (n - 1)*sizeof(double);
  NumArray *a = (NumArray *)lua_newuserdata(L, nbytes);
  a->size = n;
  return 1;  /* new userdatum is already on the stack */
}
```

(The `luaL_checkint` function is a variant of `luaL_checknumber` for integers.) Once `newarray` is registered in Lua, you can create new arrays with a statement like `a = array.new(1000).`

To store an entry, we will use a call like `array.set(array, index, value)`. Later we will see how to use metatables to support the more conventional syntax `array[index] = value`. For both notations, the underlying function is the same. It assumes that indices start at 1, as is usual in Lua.

```
static int setarray (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
  int index = luaL_checkint(L, 2);
  double value = luaL_checknumber(L, 3);

  luaL_argcheck(L, a != NULL, 1, "`array' expected");

  luaL_argcheck(L, 1 <= index && index <= a->size, 2,
                   "index out of range");
  a->values[index-1] = value;
  return 0;
}
```

The `luaL_argcheck` function checks a given condition, raising an error if necessary. So, if we call `setarray` with a bad argument, we get an elucidative error message:

```
array.set(a, 11, 0)
--> stdin:1: bad argument #1 to `set' (`array' expected)
```

The next function retrieves an entry:

```
static int getarray (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
  int index = luaL_checkint(L, 2);

  luaL_argcheck(L, a != NULL, 1, "`array' expected");

  luaL_argcheck(L, 1 <= index && index <= a->size, 2,
                 "index out of range");

  lua_pushnumber(L, a->values[index-1]);
  return 1;
}
```

We define another function to retrieve the size of an array:

```
static int getsize (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
  luaL_argcheck(L, a != NULL, 1, "`array' expected");
  lua_pushnumber(L, a->size);
  return 1;
}
```

Finally, we need some extra code to initialize our library:

```
static const struct luaL_reg arraylib [] = {
  {"new", newarray},
  {"set", setarray},
  {"get", getarray},
  {"size", getsize},
  {NULL, NULL}
};

int luaopen_array (lua_State *L) {
  luaL_openlib(L, "array", arraylib, 0);
  return 1;
}
```

Again, we use `luaL_openlib`, from the auxiliary library. It creates a table with the given name (`"array"`, in our example) and fills it with the pairs name-function specified by the array `arraylib`.

After opening the library, we are ready to use our new type in Lua:

```
a = array.new(1000)
print(a)               --> userdata: 0x8064d48
print(array.size(a))   --> 1000
for i=1,1000 do
  array.set(a, i, 1/i)
end
print(array.get(a, 10))  --> 0.1
```

Running this implementation on a Pentium/Linux, an array with 100K elements takes 800 KB of memory, as expected; an equivalent Lua table needs more than 1.5 MB.

28.2 - Metatables

Our current implementation has a major security hole. Suppose the user writes something like `array.set(io.stdin, 1, 0)`. The value in `io.stdin` is a userdatum with a pointer to a stream (`FILE*`). Because it is a userdatum, `array.set` will gladly accept it as a valid argument; the probable result will be a memory corruption (with luck you can get an index-out-of-range error instead). Such behavior is unacceptable for any Lua library. No matter how you use a C library, it should not corrupt C data or produce a core dump from Lua.

To distinguish arrays from other userdata, we create a unique metatable for it. (Remember that userdata can also have metatables.) Then, every time we create an array, we mark it with this metatable; and every time we get an array, we check whether it has the right metatable. Because Lua code cannot change the metatable of a userdatum, it cannot fake our code.

We also need a place to store this new metatable, so that we can access it to create new arrays and to check whether a given userdatum is an array. As we saw earlier, there are two common 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. We will call this new type `"LuaBook.array"`.

As usual, the auxiliary library offers some functions to help us here. The new auxiliary functions we will use are

```
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 `luaL_newmetatable` function creates a new table (to be used as a metatable), leaves the new table in the top of the stack, and associates the table and the given name in the registry. It does a dual association: It uses the name as a key to the table and the table as a key to the name. (This dual association allows faster implementations for the other two functions.) The `luaL_getmetatable` function retrieves the metatable associated with `tname` from the registry. Finally, `luaL_checkudata` checks whether the object at the given stack position is a userdatum with a metatable that matches the given name. It returns `NULL` if the object does not have the correct metatable (or if it is not a userdata); otherwise, it returns the userdata address.

Now we can start our implementation. The first step it to change the function that opens the library. The new version must create a table to be used as the metatable for arrays:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_openlib(L, "array", arraylib, 0);
  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) {
  int n = luaL_checkint(L, 1);
  size_t nbytes = sizeof(NumArray) + (n - 1)*sizeof(double);
  NumArray *a = (NumArray *)lua_newuserdata(L, nbytes);
```

```
      luaL_getmetatable(L, "LuaBook.array");
      lua_setmetatable(L, -2);

      a->size = n;
      return 1;  /* new userdatum is already on the stack */
    }
```

The `lua_setmetatable` function 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 userdatum.

Finally, `setarray`, `getarray`, and `getsize` have to check whether they got a valid array as their first argument. Because we want to raise an error in case of wrong arguments, we define the following auxiliary function:

```
    static NumArray *checkarray (lua_State *L) {
      void *ud = luaL_checkudata(L, 1, "LuaBook.array");
      luaL_argcheck(L, ud != NULL, 1, "`array' expected");
      return (NumArray *)ud;
    }
```

Using `checkarray`, the new definition for `getsize` is straightforward:

```
    static int getsize (lua_State *L) {
      NumArray *a = checkarray(L);
      lua_pushnumber(L, a->size);
      return 1;
    }
```

Because `setarray` and `getarray` also share code to check the index as their second argument, we factor out their common parts in the following function:

```
    static double *getelem (lua_State *L) {
      NumArray *a = checkarray(L);
      int index = luaL_checkint(L, 2);

      luaL_argcheck(L, 1 <= index && index <= a->size, 2,
                       "index out of range");

      /* return element address */
      return &a->values[index - 1];
    }
```

After the definition of `getelem`, `setarray` and `getarray` are straightforward:

```
    static int setarray (lua_State *L) {
      double newvalue = luaL_checknumber(L, 3);
      *getelem(L) = newvalue;
      return 0;
    }

    static int getarray (lua_State *L) {
      lua_pushnumber(L, *getelem(L));
      return 1;
    }
```

Now, if you try something like `array.get(io.stdin, 10)`, you will get a proper error message:

```
    error: bad argument #1 to `getarray' (`array' expected)
```

28.3 - 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, such as

```
a = array.new(1000)
print(a:size())      --> 1000
a:set(10, 3.4)
print(a:get(10))     --> 3.4
```

Remember that `a:size()` is equivalent to `a.size(a)`. Therefore, we have to arrange for the expression `a.size` to return our `getsize` function. The key mechanism here is the `__index` metamethod. For tables, this metamethod is called whenever Lua cannot find a value for a given key. For userdata, it is called in every access, because userdata have no keys at all.

Assume that we run the following code:

```
local metaarray = getmetatable(array.new(1))
metaarray.__index = metaarray
metaarray.set = array.set
metaarray.get = array.get
metaarray.size = array.size
```

In the first line, we create an array only to get its metatable, which we assign to `metaarray`. (We cannot set the metatable of a userdata from Lua, but we can get its metatable without restrictions.) Then we set `metaarray.__index` to `metaarray`. When we evaluate `a.size`, Lua cannot find the key `"size"` in object `a`, because the object is a userdatum. Therefore, Lua will try 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 those 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 function 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 `luaopen_array`, the function that opens the library, has to create the metatable, to assign it to its own `__index` field, to register all methods there, and to create and fill the `array` table:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  lua_pushstring(L, "__index");
  lua_pushvalue(L, -2);  /* pushes the metatable */
  lua_settable(L, -3);  /* metatable.__index = metatable */

  luaL_openlib(L, NULL, arraylib_m, 0);

  luaL_openlib(L, "array", arraylib_f, 0);
  return 1;
}
```

Here we use another feature from `luaL_openlib`. In the first call, when we pass `NULL` as the library name, `luaL_openlib` does not create any table to pack the functions; instead, it assumes that the package table is on the stack, below any occasional upvalues. In this example, the package table is the metatable itself, which is where `luaL_openlib` will put the methods. The next call to `luaL_openlib` works regularly: It creates a new table with the given name (`array`) and registers the given functions there (only `new`, in this case).

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 (for instance, `array(1000)`). The function itself is here:

```
int array2string (lua_State *L) {
  NumArray *a = checkarray(L);
  lua_pushfstring(L, "array(%d)", a->size);
  return 1;
}
```

The `lua_pushfstring` function formats the string and leaves it on the stack top. 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},
  {"set", setarray},
  ...
};
```

28.4 - Array Access

An 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 respective metamethods. A quick solution is to define those metamethods right into our Lua code:

```
local metaarray = getmetatable(newarray(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
```

(We must run that code on the original implementation for arrays, without the modifications for object-oriented access.) That is all we need to use the usual syntax:

```
a = array.new(1000)
a[10] = 3.4         -- setarray
print(a[10])        -- getarray   --> 3.4
```

If we prefer, we can register those metamethods in our C code. For that, we change again our initialization function:

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_openlib(L, "array", arraylib, 0);

  /* now the stack has the metatable at index 1 and
     `array' at index 2 */
  lua_pushstring(L, "__index");
  lua_pushstring(L, "get");
  lua_gettable(L, 2);  /* get array.get */
  lua_settable(L, 1);  /* metatable.__index = array.get */

  lua_pushstring(L, "__newindex");
  lua_pushstring(L, "set");
  lua_gettable(L, 2); /* get array.set */
  lua_settable(L, 1); /* metatable.__newindex = array.set */

  return 0;
}
```

28.5 - Light Userdata

The userdata that we have been using until now is called full userdata. Lua offers another kind of userdata, called *light userdata*.

A light userdatum is a value that represents a C pointer (that is, a `void *` value). Because it is a value, we do not create them (in the same way that we do not create numbers). To put a light userdatum into the stack, we use `lua_pushlightuserdata`:

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

Despite their common name, light userdata are quite different from full userdata. Light userdata are not buffers, but single pointers. They have no metatables. Like numbers, light userdata do not need to be managed by the garbage collector (and are not).

Some people use light userdata as a cheap alternative to full userdata. This is not a typical use, however. First, with light userdata you have to manage memory by yourself, because they are not subject to garbage collection. 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.

As a typical example, suppose we are implementing a binding between Lua and a Window system. In this binding, we use full userdata to represent windows. (Each userdatum may contain the whole window structure or only a pointer to a window created by the system.) When there is an event inside a window (e.g., a mouse click), the system calls a specific callback, identifying the window by its address. To pass the callback to Lua, we must find the userdata that represents the given window. To find this userdata, we can keep a table where the indices are light userdata with the window addresses and the values are the full userdata that represent the windows in Lua. Once we have a window address, we push it into the API stack as a light userdata and use the userdata as an index into that table. (Note that the table should have weak values. Otherwise, those full userdata would never be collected.)

29 - Managing Resources

In our implementation of arrays in the previous chapter, we did not need to worry about managing resources. They need only memory. Each userdatum 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 those other resources must be released too. Several OO languages provide a specific mechanism (called *finalizer* or *destructor*) for that need. Lua provides finalizers in the form of the `__gc` metamethod. This metamethod only works for userdata values. When a userdatum is about to be collected and its metatable has a `__gc` field, Lua calls the value of this field (which should be a function), passing as an argument the userdatum itself. This function can then release any resource associated with that userdatum.

To illustrate the use of this metamethod and of the API as a whole, in this chapter we will develop two bindings from Lua to 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.

29.1 - A Directory Iterator

Previously, we implemented a `dir` function 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 one:

```
    for fname in dir(".") 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 by a call to `closedir`. Our previous implementation of `dir` kept its `DIR` instance as a local variable and closed that 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 store its address in a userdatum and use the `__gc` metamethod of this userdatum to release the directory structure.

Despite its central role in our implementation, this userdatum representing a directory does not need to be visible from Lua. The `dir` function returns an iterator function; this is what Lua sees. The directory may be an upvalue of the iterator function. As such, the iterator function has direct access to this structure, but Lua code has not (and does not need to).

In all, we need three C functions. First, we need the `dir` function, a factory that Lua calls to create iterators; it must open a `DIR` structure and put it as an upvalue of the iterator function. 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 a metatable for directories and to initialize this metatable.

Let us start our code with the `dir` function:

```
#include <dirent.h>
#include <errno.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 userdatum to store a DIR address */
  DIR **d = (DIR **)lua_newuserdata(L, sizeof(DIR *));

  /* 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 userdatum,
     is already on the stack top */
  lua_pushcclosure(L, dir_iter, 1);
  return 1;
}
```

A subtle point here is that we must create the userdatum before opening the directory. If we first open the directory, and then the call to `lua_newuserdata` raises an error, we lose the `DIR` structure. With the correct order, the `DIR` structure, once created, is immediately associated with the userdatum; whatever happens after that, the `__gc` metamethod will eventually release the structure.

The next function is the iterator itself:

```
static int dir_iter (lua_State *L) {
  DIR *d = *(DIR **)lua_touserdata(L, lua_upvalueindex(1));
  struct dirent *entry;
  if ((entry = readdir(d)) != NULL) {
    lua_pushstring(L, entry->d_name);
    return 1;
  }
  else return 0;  /* no more values to return */
}
```

The `__gc` metamethod closes a directory, but it must take one precaution: Because we create the userdatum before opening the directory, this userdatum will be collected whatever the result of `opendir`. If `opendir` fails, there will be nothing to close.

```
static int dir_gc (lua_State *L) {
  DIR *d = *(DIR **)lua_touserdata(L, 1);
  if (d) closedir(d);
  return 0;
}
```

Finally, there is the function that opens this one-function library:

```
int luaopen_dir (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.dir");
```

```
      /* set its __gc field */
      lua_pushstring(L, "__gc");
      lua_pushcfunction(L, dir_gc);
      lua_settable(L, -3);

      /* register the `dir' function */
      lua_pushcfunction(L, l_dir);
      lua_setglobal(L, "dir");

      return 0;
    }
```

This whole example has an interesting subtlety. At first, it may seem that `dir_gc` should check whether its argument is a directory. Otherwise, a malicious user could call it with another kind of userdata (a file, for instance), 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 and Lua programs never access those directories.

29.2 - An XML Parser

Now we will look at a simplified implementation of `lxp`, a binding between Lua and Expat. 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. That 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. It is easy to add bells and whistles later, after we have implemented this core functionality. 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). The part of the Expat API that we need for this example is small. First, we need functions to create and destroy an Expat parser:

```
    #include <xmlparse.h>

    XML_Parser XML_ParserCreate (const char *encoding);
    void XML_ParserFree (XML_Parser p);
```

The argument `encoding` is optional; we will use `NULL` in our binding.

After we have a parser, we must register its callback handlers:

```
    XML_SetElementHandler(XML_Parser p,
                          XML_StartElementHandler start,
                          XML_EndElementHandler end);

    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 receive some 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,
                            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 isFinal);
```

Expat receives the document to be parsed in pieces, through successive calls to `XML_Parse`. The last argument to `XML_Parse`, `isFinal`, informs Expat whether that piece is the last one of a document. Notice that each piece of text does not need to be zero terminated; instead, we supply an explicit length. The `XML_Parse` function returns zero if it detects a parse error. (Expat provides auxiliary 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. For instance, if we only 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>`", those handlers would print

```
    + 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:

```
    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 a newline
    end
    assert(p:parse())         -- finish document
    p:close()
```

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 userdatum, but what do we need to put inside it? At least, we must keep the actual Expat parser and the callback table. We cannot store a Lua table inside a userdatum (or inside any C structure); however, we can create a reference to the table and store the reference inside the userdatum. (Remember from Section 27.3.2 that a reference is a Lua-generated integer key in the registry.) Finally, we must be able to store a Lua state into a parser object, because these parser objects is all that an Expat callback receives from our program, and the callbacks need to call Lua. Therefore, the definition for a parser object is as follows:

```
    #include <xmlparse.h>

    typedef struct lxp_userdata {
      lua_State *L;
      XML_Parser *parser;          /* associated expat parser */
      int tableref;   /* table with callbacks for this parser */
    } lxp_userdata;
```

The next step is the function that creates parser objects. Here it is:

```
    static int lxp_make_parser (lua_State *L) {
      XML_Parser p;
      lxp_userdata *xpu;

      /* (1) create a parser object */
      xpu = (lxp_userdata *)lua_newuserdata(L,
                                 sizeof(lxp_userdata));

      /* pre-initialize it, in case of errors */
      xpu->tableref = LUA_REFNIL;
      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) create and store reference to callback table */
  luaL_checktype(L, 1, LUA_TTABLE);
  lua_pushvalue(L, 1);  /* put table on the stack top */
  xpu->tableref = luaL_ref(L, LUA_REGISTRYINDEX);

  /* (4) configure Expat parser */
  XML_SetUserData(p, xpu);
  XML_SetElementHandler(p, f_StartElement, f_EndElement);
  XML_SetCharacterDataHandler(p, f_CharData);
  return 1;
}
```

The `lxp_make_parser` function has four main steps:

- Its first step follows a common pattern: It first creates a userdatum; then it pre-initializes the userdatum with consistent values; and finally sets its metatable. The reason for the pre-initialization is subtle: If there is any error during the initialization, we must make sure that the finalizer (the `__gc` metamethod) will find the userdata in a consistent state.
- In step 2, the function creates an Expat parser, stores it in the userdatum, and checks for errors.
- Step 3 ensures that the first argument to the function is actually a table (the callback table), creates a reference to it, and stores the reference into the new userdatum.
- The last step initializes the Expat parser. It sets the userdatum as the object to be passed to 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, these 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, which parses a piece of XML data. 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:

```
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");
  luaL_argcheck(L, xpu, 1, "expat parser expected");

  /* get second argument (a string) */
  s = luaL_optlstring(L, 2, NULL, &len);

  /* prepare environment for handlers: */
  /* put callback table at stack index 3 */
  lua_settop(L, 2);
  lua_getref(L, xpu->tableref);
  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;
}
```

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. Therefore, lxp_parse first prepares an environment for these handlers. 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 callback functions f_StartElement, f_EndElement, and f_CharData. All those 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 that Lua handler.

Let us first see the f_CharData handler. Its code is quite simple. It calls its corresponding handler in Lua (when present) with only two arguments: the parser and the character data (a string):

```
static void f_CharData (void *ud, const char *s, int len) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  /* get handler */
  lua_pushstring(L, "CharacterData");
  lua_gettable(L, 3);
  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 */
}
```

Notice that all these C handlers receive a lxp_userdata structure as their first argument, due to our call to XML_SetUserData when we create the parser. Also notice how it uses the environment set by lxp_parse. First, it assumes that the callback table is at stack index 3. Second, it assumes that the parser itself is at stack index 1 (it must be there, because it should be the first argument to lxp_parse).

The f_EndElement handler is also simple and quite similar to f_CharData. It also calls its corresponding Lua handler with two arguments: the parser and the tag name (again a string, but now null-terminated):

```
static void f_EndElement (void *ud, const char *name) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  lua_pushstring(L, "EndElement");
  lua_gettable(L, 3);
  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 */
}
```

The last handler, f_StartElement, calls Lua 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. We will use a quite natural translation. For instance, a start tag like

```
<to method="post" priority="high">
```

generates the following table of attributes:

```
{ method = "post", priority = "high" }
```

The implementation of `f_StartElement` follows:

```
static void f_StartElement (void *ud,
                            const char *name,
                            const char **atts) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  lua_pushstring(L, "StartElement");
  lua_gettable(L, 3);
  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);
  while (*atts) {
    lua_pushstring(L, *atts++);
    lua_pushstring(L, *atts++);
    lua_settable(L, -3);
  }

  lua_call(L, 3, 0);  /* call the handler */
}
```

The last method for parsers is `close`. When we close a parser, we have to free all its resources, namely the Expat structure and the callback table. Remember that, due to occasional errors during its creation, a parser may not have these resources:

```
static int lxp_close (lua_State *L) {
  lxp_userdata *xpu;

  xpu = (lxp_userdata *)luaL_checkudata(L, 1, "Expat");
  luaL_argcheck(L, xpu, 1, "expat parser expected");

  /* free (unref) callback table */
  luaL_unref(L, LUA_REGISTRYINDEX, xpu->tableref);
  xpu->tableref = LUA_REFNIL;

  /* free Expat parser (if there is one) */
  if (xpu->parser)
    XML_ParserFree(xpu->parser);
  xpu->parser = NULL;
  return 0;
}
```

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. That ensures that every parser eventually frees its resources, even if the programmer does not close it.

The final step is to open the library, putting all those parts together. We will use here the same scheme that we used in the object-oriented array example (Section 28.3): We will create a metatable, put all methods inside it, and make its `__index` field point to itself. For that, we need a list with the parser methods:

```
static const struct luaL_reg lxp_meths[] = {
  {"parse", lxp_parse},
  {"close", lxp_close},
  {"__gc", lxp_close},
  {NULL, NULL}
};
```

We also need a list with the functions of this library. As is common with OO libraries, this library has a single function, which creates new parsers:

```
static const struct luaL_reg lxp_funcs[] = {
  {"new", lxp_make_parser},
  {NULL, NULL}
};
```

Finally, the open function must create the metatable, make it point to itself (through `__index`), and register methods and functions:

```
int luaopen_lxp (lua_State *L) {
  /* create metatable */
  luaL_newmetatable(L, "Expat");

  /* metatable.__index = metatable */
  lua_pushliteral(L, "__index");
  lua_pushvalue(L, -2);
  lua_rawset(L, -3);

  /* register methods */
  luaL_openlib (L, NULL, lxp_meths, 0);

  /* register functions (only lxp.new) */
  luaL_openlib (L, "lxp", lxp_funcs, 0);
  return 1;
}
```