Programming in Lua

Roberto Ierusalimschy

January 22, 2003

Contents

1. Introduction

Currently, many languages are concerned with how to help you write programs with hundreds of thousands of lines. For that, they offer you packages, name spaces, complex type systems, myriads of constructions, and thousands of documentation pages to be studied.

Lua does not try to help you write programs with hundreds of thousands of lines. Instead, Lua tries to help you solve your problem with only hundreds of lines, or even less. To achieve this aim, Lua relies on *extensibility*, like many other languages. But, unlike most other languages, Lua is easily extended not only with software written in Lua itself, but also with software written in other languages, such as C/C++.

Lua was designed, from the beginning, to be easily integrated with software written in C and other "conventional" languages. This duality of languages brings many benefits. Lua is a very small and simple language, partially because it does not try to do what C is already good for, such as sheer performance, low level operations, or interface with third-party software. Lua relies on C for those tasks. What Lua offers you is what C is not good for: a good distance from the hardware, dynamic structures, no redundancies, ease of testing and debugging. For that, Lua has a secure environment, automatic memory management, and great facility to handle strings and other kinds of data with dynamic size.

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

Of course, Lua is not the only scripting language around. There are other

languages that you can use for more or less the same purposes, such as Perl, Tcl, Ruby, Forth, or Python. The following features set Lua apart from these languages; although some other languages share some of these features with Lua, no other language offers a similar profile:

- *Extensibility:* the extensibility of Lua is so remarkable that many people regard Lua not as a language, but as a kit for building domain specific languages. Lua has been designed from scratch to be extended both through Lua code and through external C code. As a proof of concept, it implements most of its own basic functionality through external libraries. It is really easy to interface Lua with C (or other language).

- *Simplicity:* Lua is a simple and small language. It has few (but powerful) concepts. That makes Lua easy to learn, and also makes for a small implementation.

- *Efficiency:* Lua has a quite efficient implementation. Several benchmarks show Lua as one of the fastest languages in the realm of scripting (interpreted) languages.

- *Portability:* when we talk about portability, we are not talking about a program that runs both in Windows and Unix platforms. We are talking about a program that runs in all platforms we ever heard about: NextStep, OS/2, PlayStation II (Sony), Mac ??, BeOS, MS-DOS, OS-9, OSX, EPOC, PalmOS, plus of course all flavors of Unix (Linux, Solaris, SunOS, AIX, ULTRIX, IRIX) and Windows (3.1, NT, 95, 98, 2000, Millenium Edition). The source code for each of these platforms is virtually the same. Lua does not use `#if`s to adapt its code to different machines; instead, it sticks to the standard ANSI (ISO) C. That way, usually you do not need to adapt it to a new environment: If you have an ANSI C compiler, you just have to compile Lua.

Typically, Lua users fall into three broad groups: those that use Lua already "embedded" in an application program, those that use Lua stand alone, and those that use Lua and C together.

Many people use Lua embedded in an application program, such as CGILua (for building dynamic Web pages), LuaMan (for network management), or LuaOrb (for accessing CORBA objects). These applications use Lua's API to register new functions, and *tag methods* (also called *fallbacks*) to create new types and to change the behavior of some language operations, configuring Lua for their specific domain. Frequently, the users of such applications do not even know that Lua is an independent language adapted for that domain; for instance, CGILua users tend to think of Lua as a language specifically designed for the Web.

Lua is also useful as a stand-alone language, mainly for text-processing and "quick-and-dirt" little programs. For such uses, the main functionality of Lua

comes from its standard libraries, which offer pattern-matching and other functions for string handling. In fact, we can regard the stand-alone language as the embedding of Lua in the domain of string and (text) file manipulation.

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

Although this book focuses on the stand-alone use of the language, most techniques described here can be applied when Lua is being used embedded into another application.

Part I

The Language

In the following chapters, we give an overview of the whole language, starting from the *Hello World* example. We will focus on different language constructs, and use numerous examples to show how to use them for practical tasks.

2.   Getting Started

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

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

If you are using the stand-alone Lua interpreter, all you have to do to run your first program is to call the interpreter (usually named `lua`) with the name of the text file that contains your program. For instance, if you write the above program in a file `first.lua`, then the command

```
prompt> lua first.lua
```

should run it.

As a more complex example, the following program defines a function to compute the factorial of a given number, then asks the user for a number, and prints its factorial:

```
-- defines a factorial function  (this line is a comment)
function factorial (n)
  if n == 0 then return 1
  else return n*factorial(n-1) end
end


print "enter a number:"
a = read("*number")          -- read a number
print(factorial(a))
```

If you are using Lua embedded in an application, such as CGILua or IUPLua, please refer to the application manual (or to a "guru friend") to learn how to run your programs. Nevertheless, Lua is still the same language, and most things 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 to run your first examples and experiments.

2.1   Environment and Chunks

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

Any command may be optionally followed by a semicolon. Usually, we use semicolons only to separate two or more commands written in the same line, but this is only a convention. Line breaks play no role in Lua's syntax. So, 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
```

Lua has no declarations, and function definitions are also commands (in fact, they are assignments, as we will see later). So, a chunk may be as simple as a single statement, such as the "hello world" example, or it may be composed by a mix of usual commands and function definitions, such as the factorial example.

A chunk may be as large as you wish, and may comprise thousands of commands and function definitions. Because Lua is also used 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 4.0  Copyright (C) 1994-2000 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), or call the `exit` function (you have to type `exit()<enter>`).

In interactive mode, each line that you type is usually interpreted as a whole chunk. Therefore, you cannot enter a multi-line definition, such as the `factorial` function, directly in interactive mode. As soon as you enter the first line, Lua complains that the function has no `end` to close it.

```
> function fat (n)
lua error: 'end' expected;
  last token read: '<eof>' at line 1 in string 'function fat (n)'
```

If you really want to enter a multi-line function in interactive mode, you can end each intermediate line with a backslash, to prevent Lua from closing the chunk:

```
prompt> lua
> function fat (n) \
>   if n == 0 then return 1 \
>   else return n*fat(n-1) end \
> end
```

But it is more convenient to put such definitions in a file, and then call Lua to run that file.

All chunks in Lua are executed in a global environment. This environment, which keeps all global variables, is initialized at the beginning of the program and persists until its end. All modifications a chunk effects on the global environment, such as the definition of functions and variables, persist after its end.

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

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

(Throughout this book, we will use the comment `-->` to show the output of a command or the result of an expression.)

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, 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.

You may execute a sequence of chunks by giving them all as arguments to the stand-alone interpreter. 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 a b
```

will run the chunk in **a**, then the one in **b**, and will print the expected 1. You may also mix files with the interactive mode, using the option -i to start an interaction. A command line like

```
prompt> lua a b -i
```

will run the chunk in **a**, then the one in **b**, and then prompt you for interaction. This is specially useful for debugging and manual testing. We will see later other options for the stand-alone interpreter.

Another way to link chunks is with the **dofile** function, which causes the immediate execution of a file. For instance, you may have a file **lib1.lua**, with your "library":

```
-- file 'lib1.lua'

function norm (x, y)
  local n2 = x^2 + y^2
  return 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))
```

The `dofile` function is also very useful 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 Lua running in interactive mode. After saving a modification that you make at your program, you execute `dofile "prog.lua"` in the Lua console to load the new code, and then you can exercise the new code, calling its functions and printing the results.

2.2   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
```

Identifiers starting with one underscore followed by one or more uppercase letters (e.g., `_INPUT`) should not be defined in regular programs; they are reserved for special uses in Lua. Identifiers made up of underscores only (e.g., `__`) are conventionally used for dummy variables or short-lived temporary values.

In Lua, the concept of what is a letter is locale dependent. So, 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, and cannot be used as identifiers:

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

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

Comments start anywhere with a double hyphen (`--`) and run until the end of the line. Moreover, the first line of a chunk is skipped if it starts with `#`. This facility allows the use of Lua as a script interpreter in Unix systems.

3. 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 six basic types in Lua: *nil*, *number*, *string*, *userdata*, *function*, and *table*. *Nil* is a type with a single value, **nil**, whose main property is to be different from any other value. It also represents *false* in boolean expressions, because Lua has no boolean type (any other value is considered *true*). Also, 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.

Number represents real (double-precision floating-point) numbers. There is no distinction between integers and floating-point numbers in Lua. Numeric constants may be written with an optional decimal part, and an optional decimal exponent. Examples of valid numeric constants are:

```
4    4.     .4     0.4     4.57e-3     0.3e12
```

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 ('\0'). Strings in Lua are immutable values. You cannot change a character inside a string, as in C; instead, you create a new string with the desired modifications:

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

Strings in Lua use automatic memory management, like all Lua objects. This 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 a whole book. Lua handles long strings quite efficiently. Programs that manipulate strings with 100K or even 1M characters are not unusual in Lua.

Literal strings can be delimited 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

We illustrate their use in the following examples:

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

A character in a string may also be specified 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.

Literal strings can also be delimited by matching double square brackets [[...]]. Literals in this bracketed form may run for several lines, may contain nested [[...]] pairs, and do not interpret escape sequences. This form is specially convenient for writing strings that contain program pieces; for instance,

```
page = [[<HTML>
  <HEAD>
  <TITLE>An HTML Page</TITLE>
  </HEAD>
  <BODY>
   <A HREF="http://www.tecgraf.puc-rio.br/lua">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 ("hello" cannot be converted)
```

Lua applies such *coercions* not only to arithmetic operands, but everywhere it expects a number. Conversely, whenever a number is used when a string is expected, the number is converted 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 explicitly convert a string to a number, you can use the function `tonumber`, which returns **nil** if the string does not denote a proper number:

```
l = read()          -- read a line
n = tonumber(l)     -- try to convert it to a number
if n == nil then
  error(l.." 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 it with the empty string. Such conversions are always valid.

```
print(tostring(10) == "10")   --> 1 (true)
print(10.."" == "10")         --> 1 (true)
```

The `type` function gives the type 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(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 types, and 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
```

Usually, when you use a single variable for different types, you end messing up
your code. However, sometimes the judicious use of these facilities is helpful,
for instance in the use of **nil** to differentiate a "normal" return value from an
exceptional condition.

The type *userdata* allows arbitrary C pointers to be stored in Lua variables.
It has no pre-defined operations in Lua, besides 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, together with the C
API, in Chapter V.

Functions are *first-class* values in Lua. This 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 to run a piece of untrusted code (such as code received
through a network). Moreover, Lua offers all mechanisms for functional pro-
gramming, 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 **??**.

Lua can call functions written in Lua and functions written in C. In the
stand-alone interpreter, all basic functions and the library functions (for string
manipulation, I/O, and mathematics) are written in C. Application programs
may define other functions in C.

The type *table* 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**). Also, tables have no fixed size; you
can dynamically add as many elements as you want to a table. Tables are the
main (in fact, the only) data structuring mechanism in Lua, and a powerful
one. Tables can be used not only to represent ordinary arrays, but also symbol
tables, sets, records, queues, and other data structures, in simple and efficient
ways.

Tables in Lua are neither values nor variables, but *objects*. If you are familiar
with arrays in Java or Scheme, then you have a fair idea of what we mean by
objects. 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 fact, there is no way to declare anything in

Lua. Tables are only created by means of *constructor* expressions, which in its simplest form is written as {}:

```
-- creates a new table, and stores a reference to it in 'a'
a = {}
k = "x"
a[k] = 10        -- new field, with key="x" and value=10
a[20] = "great"  -- new field, with key=20 and value="great"
print(a["x"])    --> 10
k = 20
print(a[k])      --> "great"
a["x"] = a["x"]+1    -- increments field "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 is no references left to the table
```

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

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

```
a = {}
-- creates 1000 new fields
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. (This is not a coincidence: Lua stores all its global variables in an ordinary table. More about this in Chapter ??.)

To represent records, Lua uses the field name as an index. The language supports this representation by providing `a.name` as syntactic sugar for `a["name"]`. Therefore, the last lines of the previous example could be written more cleanly as

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

From a semantics point of view, the two forms are equivalent, and can be freely intermixed.

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`.

```
a = {};   x = "y"; y = "x"
a[x] = 20;  a[y] = 10
print(a.x)    --> 10
print(a.y)    --> 20
```

As we have seen, when you use a table to represent a conventional array, you do not declare its size. You just initialize the elements you need. Sometimes it is convenient to use an extra field, usually called `n`, to keep the array size:

```
-- read 10 lines and store them in a table
local a = {}
for i=1,10 do
  a[i] = read()
end
a.n = 10
```

This use of `a.n` is only a convention, although a quite convenient one: Most functions from the standard library adhere to this convention. But you do not need to store an array size anywhere. In an array iteration, the first non initialized index will result in **nil**, and you can use this fact 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
local i = 1
while a[i] do
  print(a[i])
  i = i+1
end
```

The basic Lua library provides a handy function, `getn`, that gives the "size" of an array. If the array (which is actually a table) does have a field `n`, then the value of this field is the array size. Otherwise, the size is the largest numeric index with a non-nil value.

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 1 (and not with 0, 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 a table can be indexed both by the number 0 and by 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. Particularly, when in doubt about the actual types of your indices, use tonumber 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 have subtle bugs if you do not pay attention to this point.

4. 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 quite unconventional upvalues, function definitions, and constructors.

4.1 Arithmetic Operators

Lua supports the usual arithmetic operators: the binary `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and the unary `-` (negation). All of them operate over 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 (through the `pow` function in C) would mean that Lua should be linked with the C mathematic library. To avoid this, the core of Lua offers only the syntax for the `^` binary operator, which have the higher precedence among all operations. The mathematic library (which is standard, but not part of the Lua core) then gives to this operator its expected meaning.

4.2 Relational Operators

Lua provides the following relational operators:

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

All these operators return **nil** as false and a value different from **nil** as true.

The order operators can be applied only to two numbers or two strings. The operator `==` tests for equality, and can be applied to any two values. If the values have different types, they are considered different values. Otherwise they are compared according to their types. The operator `~=` is exactly the negation of equality.

Numbers are compared in the usual way. Alphabetical order for strings follows the locale set for Lua. For instance, with the European Latin-1 locale, we have `"acai" < "açaí" < "acorde"`. Other types can only be compared

for equality (and inequality). Tables, userdata and functions are compared by reference, that is, two tables are considered equal only if they are the very same table. 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`.

When comparing values with different types, you must be careful. Although you may apply arithmetic and other numeric operations over the string `"0"`, this is still a string. Thus, `"0"==0` is false. Moreover, `2<15` is obviously true, but `"2"<"15"` is false (alphabetical order!). To avoid inconsistent results, Lua gives an error when you mix strings and numbers in an order comparison, such as `2<"15"`.

4.3   Logical Operators

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

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

Both `and` and `or` use short-cut evaluation, that is, the second operand is evaluated only when necessary.

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

```
if x == nil then x = v end
```

i.e., it sets `x` to a default value `v` when `x` is not set.

Another useful idiom is `(e and a) or b`, which is equivalent to the C expression

```
e ? a : b
```

(provided that $Ta is not **nil**). For instance, we can select the maximum of two numbers `x` and `y` with a statement like

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

If `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` results in the value of its first expression (`x`). If `x > y` is false, the whole `and` expression is false, and so the `or` results in its second expression, `y`.

4.4 Concatenation

The string concatenation operator in Lua is denoted by "`..`" (two dots). If any of its operands is a number, it is converted 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
```

4.5 Precedence

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

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

All binary operators are left associative, except for `^` (exponentiation), which is right associative. Therefore, the following expressions on the left are equivalent to the ones 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 to look in the manual, and probably you will have the same doubt when you read the code again.

4.6 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 have seen it before. Constructors can also be used to initialize arrays (also called sequences or lists). For instance, the statement

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

will initialize `days[1]` with the string `"Sunday"`, `days[2]` with `"Monday"`, and so on:

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

The first element has always index 1 (not 0). 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)}
```

If we want to initialize a table used as a record, we can use the following syntax:

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

which is equivalent to

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

No matter which 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
```

That is, *all tables are created equal*; only the initialization is affected by the constructor.

Every time a constructor is evaluated, it creates and initializes a new table. Therefore, we can use tables to implement linked lists:

```
list = nil
while 1 do
  local line = read()
  if not line then break end
  list = {next=list, value=line}
end
```

This code will read lines until an empty one, and will store them in a linked list. Each node in the list is a table with two fields: a `value`, with the line contents, and `next`, with a reference to the next node. To print the list contents, we can use the following code: (Because we implemented our list as a stack, the lines will be printed in reverse order.)

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

Although instructive, we hardly use the above implementation in real Lua programs; lists are better implemented as arrays (we will see more details about this in ??).

We can use record-style and list-style initializations in the same constructor, if we separate both parts with a semicolon:

```
colors = {"blue", "red", "green", "yellow"; n=4}

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 constructors can be nested to represent more complex data structures. The elements `polyline[1]`, ..., `polyline[4]` are each a table representing a record:

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

Although useful, those two constructor forms have their limitations. For instance, you cannot initialize fields with negative indices, or with string indices which are not proper identifiers. For that, there is yet another constructor form, where each index to be initialized is explicitly written as an expression, between square brackets.

```
opnames = {["+"] = "add", ["-"] = "sub",
           ["*"] = "mul", ["/"] = "div"}

i = 20; s = "-"
a = {[i+0] = s, [i+1] = s..s, [i+2] = s..s..s}

print(opnames[s])    --> sub
print(a[22])         --> ---
```

This syntax is more cumbersome, but also more flexible: Both the list-style and the record-style forms are special cases of this more general one. So,

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

is equivalent to

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

and

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

is equivalent to

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

You can always put a comma after the last entry. These trailing commas are optional, but are always valid. Moreover, you can use a semicolon even when one of the parts (or both) are empty:

```
{[1]="red", [2]="green", [3]="blue",}
{3, 4, ; x=0, y=12,}
{3, 4 ; }
{ ; x=0, y=12,}
{;}        -- empty table
```

This flexibility makes it easier to write programs that generate Lua tables, as they do not need to handle special cases neither for empty lists nor for the last elements.

5. Statements

Lua supports an almost conventional set of statements, similar to those in Pascal or C. The conventional statements include assignment, control structures and procedure calls. Non-conventional commands include multiple assignment and local variable declarations.

5.1   Assignment

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

```
a = a/2
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 are written with their elements separated by commas:

```
a, b = 10, 2*x
a, b = f(x)
t[x], t.n, a = t[x]+1, f(t.n), t.n+a
```

Lua first evaluates all values on the right side, and after that it makes the assignments. Therefore, we can use a multiple assignment to swap two values, as in

```
x, y = y, x
a[i], a[j] = a[j], a[i]
```

In any assignment, when the list of values is shorter than the list of variables, the extra variables receive **nil** as their values. One the other hand, if 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, c+1    -- c+1 is ignored
print(a,b)              --> 1    2
a, b, c = 0
print(a,b,c)            --> 0    nil    nil
```

33

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
```

5.2   Local Variables and Blocks

Besides global variables, Lua supports local variables. Local variables are created with the statement `local`:

```
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)
  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)
```

It is good programming style to use local variables whenever possible. The access to local variables is faster than to global ones; moreover, with local variables you avoid cluttering the global environment with unnecessary names, and also avoid name clashes.

In Lua, local variable declarations are handled as statements. As such, you can write local declarations anywhere you can write a statement. The scope begins after the declaration and lasts 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 particular case, if a declaration has no initial assignment, its variables are initialized 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
```

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.

A block may be explicitly delimited, bracketed with the keywords `do ...  end`. This can be useful when you need a 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
print(x1, x2)
```

5.3   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` for `if`, `for` and `while`; `until` for `repeat`.

If

An `if` 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 equivalent to a `else if`, but avoids the need of 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
```

The condition expression of a control structure may result to any value. All values different from **nil** are considered true; only **nil** is considered false.

```
if a then
  print(a)
else
  error("variable not initialized")
end
```

While

As usual, `while` first tests its condition; if it is false, then the loop ends; otherwise, the *body* of the loop is executed, and the whole process is repeated.

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

Repeat

As the name implies, `repeat-until` repeats its body until its condition is true. The test is done after the body, so the body is always executed at least one time.

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

For

The `for` statement has two variants: one for numbers and one for tables. Let us see first the numeric `for`.

Numeric `For`

A numeric `for` has the following syntax:

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

This loop will execute `something` for each value of `var` between `exp1` and `exp2`, using `exp3` as the *step* to increment `var`. This third expression is optional, and when absent Lua assumes 1 as the step value. As typical examples of such loops, we have

```
for i=1,10 do print(i) end
```

```
for i=10,1,-1 do print(i) end
```

```
a = {'a', 'b', 'c', 'd'}
for i=1,getn(a) do print(a[i]) end
```

The for loop has some subtleties that you should learn to make a good use of it. First, all expressions are evaluated once, before the loop starts. For instance, in the last example, `getn(a)` is called only once, and not four times. Second, the loop 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 a global variable
```

If you really need the value of the loop variable after the loop (usually when you break the loop), you must put this value in another variable:

```
-- find a value in a list
local found = nil
for i=1,getn(a) 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 loop variable: Lua may use some kind of internal counter to control the loop, so the effect of such change is unpredictable. More specifically, if you want to break a for loop before its normal termination, use `break`.

Table `For`

The `for` loop for tables *traverses* a table, running its body for each key-value pair in the table. Follows a typical example of its use:

```
t = {x = 1, y = 10, z = -4}
for k,v in t do print(k,v) end
```

For each iteration, the first loop variable (`k` in our example) gets one table index, while the second variable (`v`) gets the value associated with that index. The order of the traversal is completely undefined, because it depends on the way that the table elements are stored.

   The loop for tables has the same properties of the numeric loop. The expression denoting the table is evaluated only once; 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 how traversing tables can help us. 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 order in the week. You can search the table, looking for the given name; because this table is a list, you can traverse it with a numeric for:

```
function find (d)
  for i = 1, getn(days) do
    if days[i] == d then return i end
  end
  return nil    -- not found
end
```

Frequently, however, a better approach in Lua is to build a *reverse table*, say `rev_days`, that has the names as indices and the numbers as values. Then, all you have to do to find the order of a name is to index this reverse table:

```
x = "Tuesday"
print(rev_days[x])    --> 3
```

   To build a reverse table, the following code is enough:

```
rev_days = {}
for i,v in days do
  rev_days[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"`, ...).

   You can always use a `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. For syntactical reasons, a `break` can only appear as the last statement in a block. 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
```

If you really need to write a break in the middle of a list of statements, you can always create a block, so that the break is its last statement statement:

```
...
do break end
...
```

6. Functions

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

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

In both cases, a list of *arguments* are given enclosed in parentheses. If the function call has no arguments, an empty list () must be used to indicate the call. There is a special case to this rule: If the function has exactly one 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({})
```

Functions used by a Lua program can be defined both in Lua and in C (or other language used by the host application). Both are used in exactly the same way inside Lua. For instance, all builtin functions, as well as the library functions, are written in C, but this fact has no relevance to Lua programmers.

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

```
-- add all elements of array 'a'
function add (a)
  local sum = 0
  for i = 1, getn(a) do
    sum = sum + a[i]
  end
```

41

```
    return sum
end
```

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

   Parameters work as local variables, initialized with the actual arguments given in the function call. You can use them as any other local variable:

```
function factorial (n)
  local f = 1
  while n>1 do
    f = n*f; n = n-1
  end
  return f
end
```

   You can call a function with a number of arguments different from its number of parameters. Like in an assignment, extra arguments are thrown away, extra parameters get **nil**. For instance, if we have a function as

```
function f(a, 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, specially for default parameter values. For instance, consider the following function:

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

When you call `inc_count()`, without arguments, `n` is first initialized with **nil**, the **or** returns its second operand, and then a default 1 is assigned to `n`.

   A **return** statement can be used to return eventual results from a function, or just to finish it. 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:

```
function dummy (s)
  if s == "" then return end  -- do not print an empty string
  print(s)
end
```

For syntactical reasons, a return statement can only appear as the last statement of a block (in other words, as the last statement in your chunk, or just before an `end` or an `else`). Usually, these are the places where we use a return, because any statement following a return is useless. Sometimes, however, it may be useful to write a return 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 return:

```
function foo ()
  return           --<< SYNTAX ERROR
  -- 'return' is the last statement in the next block
  do return end    -- OK
  ...              -- statements not reached
end
```

An unconventional, but quite convenient, feature of Lua is that functions may return multiple results. Many builtin and pre-defined functions in Lua return multiple values. For instance, `strfind` locates a pattern in a string. It returns two indices: where the pattern starts, and where it ends (or **nil** if it cannot find the pattern). A multiple assignment gets these multiple results:

```
s, e = strfind("hello $name, hello", "$%w+")
--    indices:  123456789012345678

print(s, e)   --> 7       11
```

The pattern `%w` matches a single letter, `%w+` matches a sequence of one or more letters, so `$%w+` matches `$name`, which starts at the 7th character of the sting and ends at the 11th. (More about pattern matching in Chapter **??**.)

Functions written in Lua can also 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 index:

```
function maximum (a)
  local mi = 1           -- maximum index
  local m = a[mi]        -- maximum value
  for i = 2,getn(a) do
    if a[i] > m then
      mi = i; m = a[mi]
    end
  end
  return m, mi
end

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

Following the dynamic nature of Lua, the number of results from a function is always adapted to the circumstances of the call. When we call a function as

a statement, all its results are discarded. When we use a call as an expression, only the first result is kept. We get all results only when we put the call as the only (or the last) expression in an expression list (expression lists occur in multiple assignment, return statements, and function arguments). If a function has no results, or not as many results as we need, **nil**s are produced. See the example:

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

foo0()
foo1()          -- 'a' discarded
foo2()          -- 'a' and 'b' discarded
print(foo0())   -->
print(foo1())   -->  a
print(foo2())   -->  a    b
print(foo2().."x")   -->  ax
x,y=foo0()      -- x=nil, y=nil
x,y=foo1()      -- x='a', y=nil
x,y=foo2()      -- x='a', y='b'
x=foo2()        -- x='a', 'b' is discarded
```

(Until version Lua 3.2, the expression `print(foo2())` would print only the first result, `"a"`.) When `foo2` is used inside an expression, Lua adjusts the number of results to 1, and so only the `"a"` is used in the concatenation.

7. More About Functions

Functions in Lua are first-class values, and they have a form of proper lexical scoping through *upvalues*. Through those mechanisms, Lua properly contains the lambda calculus, and therefore many powerful programming techniques used in functional languages can be applied in Lua as well. If you know nothing about functional programming, do not worry; as we have seen, functions in Lua can be used in conventional ways, too. Nevertheless, it is worth learning a little about how to explore those mechanisms, which can make your programs smaller and simpler.

What does it mean for functions to be "first-class values"? It means that a function is a value in Lua, with the same rights as conventional values like numbers and strings. Functions can be stored in variables (both global and local), passed as arguments and returned by other functions, and can be stored in tables.

A somewhat difficult notion in Lua is that functions, like all other values, do not have names. When we talk about a function name, say `print`, we are actually talking about a variable that holds that function. The "name" of a function in Lua is the name of a plain global variable that happens to hold its value. Like any other variable holding any other value, we can manipulate it in many ways. The following example, although a little silly, shows our point:

```
a = {p = print}
a.p("Hello World")   --> Hello World
print = sin
a.p(print(10))       --> 0.173648
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 what we call a *syntactic sugar*, 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, more or less in the same way that `{}` is a table constructor. We call the result of such function constructor an *anonymous function*. Although we usually assign functions to global names, therefore giving them something like a name, there are several occasions where functions remain anonymous. Let us see some examples.

Lua provides a function `sort`, which receives a table and sorts its elements. Such function must allow endless 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 argument, which is the *order function*: A function that gets two elements and tells whether the first must come before the second in the sort. So, 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 it by the field `name`, in reverse alphabetical order, we just write

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

See how handy the anonymous function is in this statement. Without this facility, we would have to declare this function at the global level, usually far from the only place where we need it.

A function that gets another function as an argument, like `sort`, is usually called a *higher-order* function. Higher-order functions is a powerful programming mechanism, and the use of anonymous functions to create their function arguments is always a great source of flexibility. But remember that higher-order functions have no special rights; they are a simple consequence of Lua's ability to handle functions as first-class values.

To illustrate the use of functions as arguments, we will write a naive implementation for a common higher-order function, `plot`, that plots a 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 erase_terminal ()
  write("\27[2J")
end

-- writes an '*' at collumn 'x' , row 'y'
function mark (x,y)
  write(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 [0,1])
function plot (f)
  erase_terminal()
  for i=1,TermSize.w do
    mark(i, (1-f(i/TermSize.w))*TermSize.h)
  end
end
```

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

```
plot(function (x) return (1+sin(x*360))/2 end)
```

(We need to massage the data a little to put values in the proper range; notice that the `sin` function in Lua gets its argument in degrees.) When we call `plot`, its parameter `f` gets the value of the given anonymous function, which is then called repeatedly inside the `for` loop to do 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 in Chapter **??**, the use of functions in table fields is a key ingredient for some advanced uses of Lua, such as packages and object-oriented programming.

7.1 Closures & Upvalues

Though powerful, the mechanism of anonymous functions has a major drawback: When a function is written enclosed in another function, it has no access to local variables from the enclosing function. For instance, suppose you want to sort a list of names, according to grades given to the names; those grades are given in a separate table, as follows:

```
names = {"Peter", "Paul", "Mary"}
grades = {Mary = 10, Paul = 7, Peter = 8}
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. Sounds easy, right?

```
-- Warning: invalid Lua code
function sortbygrade (names, grades)
  sort(names, function (n1, n2)
    return grades[n1] < grades[n2]    -- compare the grades
  end)
end
```

What is wrong in this example is that the anonymous function called by `sort` cannot access the parameters `names` and `grades`.

Why not? This is an old problem regarding scope and life span of local variables. Suppose that an enclosed function has free access to the local variables of its enclosing function. Then we could create counters with the following code:

```
-- Warning: invalid Lua code
function new_counter ()
  local i = 0
  return function ()
          i = i+1
          return i
        end
end

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

In this code, `new_counter` is supposed to be a function that creates and then returns another function (a "counter"). But when we call this counter, in the two last lines of our example, which variable `i` is it going to increment? After all, `i` is a local variable of `new_counter`, and as such it no longer exists after `new_counter` has returned!

Different languages have managed this problem in different ways. C avoids the problem altogether by not providing nested functions; it also avoids the benefits of this kind of programming. Pascal (the standard one) allows functions to be passed as arguments, but not to be returned or stored in variables. Functional languages (such as Scheme and ML) do not use a stack to keep local variables, so these variables do not need to be deleted when a function ends, but this kind of solution is expensive for procedural languages.

How does Lua handle this problem? Lua keeps its local variables in a stack, and to avoid invalid references, an enclosed function cannot access a local variable from the enclosing function. (That is why both previous examples were invalid.) But it can access an *upvalue* from those local variables. An upvalue is written as `%varname`, where `varname` is the name of a variable visible at the point where the function is declared. It behaves as a kind of a constant inside a function, whose value is fixed when the function is instantiated. Consider the following example:

```
function create_print (i)
  local p = function () print(%i) end
  i = 300
  return p
end

p = create_print(10)
p()            --> 10
```

When `create_print` starts running and creates the enclosed function (`p`), the upvalue `%i` is "frozen" with the current value of `i`; 10 in our case. After that, it does not matter what else happens to `i`, not even whether it still exists. Function `p` does not access that variable when it runs, but a private, read-only copy of its value (the *upvalue*). Each time `create_print` is called, a new instance of `p` is created, with its own private upvalue. This pair of a function with its upvalues is called a *closure*.

Using upvalues, we are now able to write the `sortbygrade` function;

```
function sortbygrade (names, grades)
  sort(names, function (n1, n2)
    return %grades[n1] < %grades[n2]    -- compare the grades
  end)
end
```

Now, the anonymous function (the argument to `sort`) does not refer to an outside variable anymore, but to an upvalue with its value (`%grades`).

Our other example, the counter, presents a small subtlety. Because upvalues are constants, we cannot assign to them. The trick for such cases is to use a table as the (constant) value, and to do the assignment to a field, `v` for instance.

```
function new_counter ()
  local count = {v=0}
  return function ()
          %count.v = %count.v+1
          return %count.v
         end
end

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

Now, each time `new_counter` is called, a new table is created in `count` and kept as the upvalue for the anonymous function. Therefore, each function has its private counter, and they do not interfere with each other.

Closures are a valuable tool in many contexts. As we have seen, they are useful as arguments to higher-order functions such as `sort`. Closures are also valuable for functions that build other functions, as our `new_counter` example; this mechanism allows Lua programs to incorporate fancy programming techniques from the functional world. Finally, closures are very useful for *callback* functions. The typical example here is when you create buttons in some GUI toolkit. Each button has a callback function that is 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 digit_button (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. The callback function can be called a long time after the `digit_button` function did its task and after the local variable `digit` was destroyed, but it still can access its upvalue `%digit`.

Surprisingly, upvalues are valuable also in a quite different context. Because functions are stored in global variables, it is trivial to redefine functions in Lua, even pre-defined functions. This facility is one of the reasons why 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 radians instead of degrees. This new function must convert its argument and then call the original `sin` function to do the real work. Your code could look like

```
old_sin = sin
sin = function (x)
        return old_sin(x*180/PI)
      end
```

A better way to do that is as follows:

```
sin = function (x)
        return %sin(x*180/PI)
      end
```

Because `sin` is a (global) variable which is visible at the point where the function is being defined, it can be used as an upvalue. When the new function `sin` is instantiated, the current value of `sin` (that is, the original function, before the assignment) is captured in the upvalue `%sin`. Thereafter, whenever `sin` (the

new one) is called, it uses its private copy of the old function to do the actual work.

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 read, we can redefine function `readfrom` using upvalues.

```
readfrom = function (filename)
             if not access_OK(filename) then
               return nil, "access denied"
             else
               return %readfrom(filename)
             end
           end
```

What makes this example nice is that, after this redefinition, there is no way for the program to call the unrestricted `readfrom`, but through the new restricted version. The insecure version is kept as a private upvalue, unaccessible from the outside. With this facility, Lua sandboxes can be built 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.

The main rule when using upvalues is that an upvalue inside a function gets the value that its respective variable has at the point where the function appears. Therefore, you can only use as upvalue a variable that is visible at that point. Here are some tricky cases:

```
a,b,c = 1,2,3   -- global variables
function f (x)
  local b    -- x and b are local to f
  local g = function (a)
    local y  -- a and y are local to g
    p = a    -- OK, access local 'a'
    p = c    -- OK, access global 'c'
    p = b    -- ERROR: cannot access a variable in outer scope
    p = %b   -- OK, access frozen value of 'b' (local to 'f')
    p = %c   -- OK, access frozen value of global 'c'
    p = %y   -- ERROR: 'y' is not visible where 'g' is defined
  end  -- g
end  -- f
```

We seldom use more than one level of nested functions. It is still less common the need to access, as upvalue, a variable that is local to a function more than one level above. But if you really need to do that, you can always use an intermediate local variable, in the function in between, to carry the value you need:

```
-- INVALID CODE
```

```
function f(x)
  local g = function (y)
            local h = function (z)
                      return %x      -- ERROR
                    end
          end
end

-- valid code
function f(x)
  local g = function (y)
            local xx = %x    -- OK
            local h = function (z)
                      return %xx     -- OK
                    end
          end
end
```

In the first implementation, the access to `x` is invalid, because `x` is not visible
at the point where `h` is defined (inside `g`). In the second implementation, an
intermediate local variable, `xx`, keeps the value of `x`, and `h` gets the value it
needs from this intermediate variable. (In fact, we did not need a new name for
this intermediate variable; we could call it `x`, too; `local x = x` is OK in Lua.
We chose `xx` in the example only for explanatory purposes.)

7.2   Variable Number of Arguments

Some functions in Lua may be called with a variable number of arguments. For
instance, we have already called `print` with 1, 2, and more arguments.

Suppose now that we want to redefine `print` in Lua: Our new 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:

```
PRINT_RESULT = ""

function print (...)
  for i=1,arg.n do
    PRINT_RESULT = PRINT_RESULT..tostring(arg[i]).."\t"
  end
  PRINT_RESULT = PRINT_RESULT.."\n"
end
```

The three dots (`...`) in the parameter list indicates that this 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 arguments plus a variable number of arguments. For instance, a function `placeargs` could have a first argument to act as a template string; a `$n` in the template signals where to put argument `n`.

```
print(placeargs("$2 $1!", "World", "Hello"))
  --> Hello World!
print(placeargs("$1 $2 $1!", "Brazil", "Hi"))
  --> Brazil Hi Brazil!
```

To accommodate that need, a function may have a regular list of parameters before the `...`. Then, the first arguments are assigned to those parameters, and only the extra arguments (if any) go to `arg`. As an example, suppose 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}
```

Back to our example, we can write our version of `placeargs` as

```
function placeargs (s, ...)
  return gsub(s, "$(%d)", function (i)
    return %arg[tonumber(i)]
  end)
end
```

The `gsub` function, from the standard library, does the hard work. We will see in great details how `gsub` and other pattern-matching functions work in Section 17.1; here we will just give an overview of what goes on in this example. When we call `gsub`, it scans the string `s`, looking for the pattern `$(%d)`, that is, a dollar sign followed by a digit. When it has a match, it calls the given function (anonymous, in our example) with the part of the pattern between parentheses (that is, the digit) as its argument, and uses the function result to substitute the match. Therefore, a `$2` is replaced by the value of `arg[2]`, and so on. (The `tonumber` is needed because the pattern is always captured as a string; when `$1` is found, the parameter `i` gets the value `"1"`. But `arg` is indexed by numbers, so we need to convert the argument.)

7.3 Named Parameters

The parameter passing mechanism in Lua is *positional*, meaning that, when you 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 be able to specify the arguments by name. To illustrate this point, let us consider the function `rename`, that 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, a `oldname` and a `newname`:

```
-- invalid code
rename(oldname="temp.lua", newname="temp1.lua")
```

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

```
rename{oldname="temp.lua", newname="temp1.lua"}
```

Accordingly, we define the function with only one parameter, and get the actual arguments from this parameter.

```
function rename (arg)
  return %rename(arg.oldname, arg.newname)
end
```

This style of parameter passing is specially 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=1
          }
```

The `Window` function can then check for mandatory arguments, add default values, and the like.

8. Errors

Errare humanum est. So, we must handle errors the best way we can. Because Lua is an extension language, frequently used 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. (You can modify this behavior using *tag methods*, as we will see later.) 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, to run a chunk with syntax errors, and so on. Also, you can explicitly raise an error calling the function `error`; its only optional argument is the error message. Usually, this function is a good way to handle errors in your code.

```
print "enter a number:"
n = 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 builtin function just for that job, called `assert`.

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

The `assert` function checks whether its first argument is not **nil**, and raises an error if it is. Its second argument is optional, so that if you do not want to say anything in the error message, you do not have to.

When a function finds a somewhat 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 these two options, but we can provide a general rule: An exception which are 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 over a table? Suppose it returns an error code. If we need to check for errors, we would have to write something like

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

But 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 are not going to check neither the argument nor the result of a call to `sin`; if the argument is not a number, probably it means that there is a bug 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 `readfrom` function, which opens a file for reading. How should it behave when called over a non-existent file? In this case, there is no simple way to check for the exception before calling the function. In many systems, the only way to know whether a file exists is trying to open it. So, if `readfrom` 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 this situation in an appropriate way, for instance asking the user for another file name:

```
local status, msg
repeat
  print "enter a file name:"
  local name = read()
  if not name then return end    -- no input
  status, msg = readfrom(name)
  if not status then print(msg) end
until status
```

If you do not want to handle such situations, but still want to play safe, you can use the `assert` function to guard the operation:

```
assert(readfrom(name))
```

If `readfrom` fails, `assert` will raise an error. (This is a typical Lua idiom.)

When you use Lua in interactive mode, each line is an independent chunk. So, whenever an error occurs the rest of the line is skipped, and you get a new prompt. For instance:

```
> error("test"); print(10)   -- this "10" is not printed
lua error: test
Active Stack:
  function 'error' [(C code)]
  main of string 'error("test"); print(10) ...'
> print(20)   -- new chunk
```

```
20
>
```

When you run a file with `dofile`, the whole file is executed as an independent chunk. If there is an error, that chunk ends and `dofile` returns with an error code. The original chunk, which called `dofile`, does not stop. The same happens with `dostring`, which executes a string as a chunk:

```
> dostring("error('test'); print(10)"); print(20)
lua error: test
Active Stack:
  function 'error' [(C code)]
  main of string 'error('test'); print(10)'
  function 'dostring' [(C code)]
  main of string 'dostring("error('test'); ...'
20
```

Lua does not print the 10, because the chunk ran by `dostring` ends when the error happens; but it prints the 20 when `dostring` returns, after the error message.

8.1   Error Messages

When an error happens, the interpreter calls the function `_ERRORMESSAGE` with an error message as its only argument, right before finishing the current chunk. The default implementation of this function only calls the function `_ALERT`, which then prints the error message on the console (`stderr`).

The I/O library redefines the function `_ERRORMESSAGE`, so that it gives more information about the error. Again, it uses `_ALERT` to display this extended error message. This new implementation uses the *debug API* of Lua to gather informations such as the name of the function where the error happens, the calling stack, and the like. For instance,

```
error: attempt to perform arithmetic on global 'a' (a nil value)
Active Stack:
  function 'f' at line 3 [file 'temp']
  function 'g' at line 7 [file 'temp']
  main of file 'temp' at line 10
```

Particularly, the first line of the active stack gives exactly the line where the error happens (line 3 of file `temp`, in that example).

You can change where Lua shows its messages by redefining the `_ALERT` function. For instance, if you are using a window system without consoles, you can redefine `_ALERT` as a function that, when called, opens a pop-up window to show its only argument.

If you want to change what Lua shows in an error message, or the message format, then you should redefine `_ERRORMESSAGE`. As a special case, if you set

_ERRORMESSAGE to **nil**, Lua suppresses error messages. As a typical use of this facility, imagine that your program needs to execute a piece of code received from outside (for instance from a network connection). If there is any error, it does not make sense to show to the user the error message; instead, your function must only return an error code. The following function does the job:

```
function exec (code)
  local oldEM = _ERRORMESSAGE
  _ERRORMESSAGE = nil      -- to avoid error messages
  local result = dostring(code)
  _ERRORMESSAGE = oldEM   -- restore previous value
  return result
end
```

During the execution of the string `code`, **_ERRORMESSAGE** is **nil**; so, any error that happens will not be reported. Nevertheless, in case of errors `dostring` will stop the execution of its argument, and will return **nil** to signal the error. Just try:

```
> exec("a=1")        -- OK
> exec("a=2+")       -- syntax error!
```

8.2   Error Handling and Exceptions

For most applications, you do not need to do any error handling in Lua. Usually this handling is done by the application. 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 ignores the error code, and continues showing the prompt and running the commands.

If you need to handle errors in Lua, you can use the `dostring` function to encapsulate your code, as we did previously with the `exec` function. But a better way to do that is with the `call` function.

Suppose you want to run some 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
  ...
end
```

Then, you call `foo` with the `call` function:

```
if not call(foo, {}, "x", nil) then
  -- 'foo' raised an error: take appropriate actions
```

```
  ...
else
  -- no errors while running 'foo'
  ...
end
```

We will discuss other uses of the `call` function, and the meaning of its arguments, in the next chapter. Here, what we need to know is that, when used as shown previously, it calls its first argument (`foo`) in protected mode, using the last argument as a temporary `_ERRORMESSAGE` function (remember that, when `_ERRORMESSAGE` is **nil**, Lua suppresses any error messages). If there are no errors, `call` returns a non **nil** value; otherwise, it returns **nil**.

When you catch an error, you may want to know the original error message. For that, you may set an appropriate `_ERRORMESSAGE` function that stores the message for later use:

```
local up = {msg=nil}
if not call(foo, {}, "x", function (msg) %up.msg = msg end) then
  -- 'foo' raised an error: the error message is in 'up.msg'
  ...
end
```

(We initialized field `msg` with **nil** only for documentation; when we create a new table all fields are nil by default.)

There are two main reasons to catch errors in Lua. The first one is when you need to run some code external to your code (e.g. from a library or given by the user), and therefore you cannot control whether the code raises an error. The second reason is when you raise some error in your code as a quick and simple way to recover from some unexpected situation, knowing that the error will be caught in another part of your program. This is what we call *exception handling*.

9.   Complete Examples

To end this chapter, we show two complete programs that illustrate different facilities of the language.

9.1   Data Description

Our first example is a real program from the Lua site. We keep a record of projects around the world that use Lua. Each entry is represented by a constructor, in a quite auto-documented way:

```
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,
    who 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, that does a sequence of calls to a function `entry`, using the described table as the call argument.

Our goal is to write a program that shows that data in HTML. Because there are lots of projects, the final page first shows a list of all project titles, and then shows the details of each project. The final result of the program is something like

```
<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>

<A NAME="1"></A><HR>
<H3>
<A 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`). Because we
have to traverse the entry list 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:
To run the data file twice, each time with a different definition for `entry`. We
follow this approach in the next program. The first function writes all fixed
page headers:

```
function BEGIN()
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 the title project as a list item. The argument o will be the table describing the whole project.

```
function entry0 (o)
  N=N+1
  write(format('<LI><A HREF="#%d">%s</A>\n', N, o.title or '(no 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 information about a project. It is a little more complex, because all items are optional.

```
function entry1 (o)
  N=N+1
  write('\n<A NAME="',N,'"></A>',"<HR>\n<H3>\n")
  if o.url then write(format('<A HREF="%s">', o.url)) end
  write(o.title or o.org or "")
  if o.url then write("</A>\n") end
  if o.title and o.org then
    write("<BR>\n<SMALL><EM>", o.org,"</EM></SMALL>")
  end
  write("\n</H3>\n");
  if o.description then
    write(gsub(o.description, "\n\n\n*", "<P>\n"),"<P>\n")
  end
  if o.email then
    write(format('Contact: <A HREF="mailto:%s">', o.email))
    write(o.contact or o.email)
    write("</A>\n")
  elseif o.contact then
    write("Contact: ", o.contact,"\n")
  end
end
```

The last function closes the page:

```
function END()
  write("</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 again the data file with the second definition for **entry**, and closes the page:

```
BEGIN()

N=0
```

```
entry=entry0
write("<UL>\n")
assert(dofile("db.lua"))
write("</UL>\n")

N=0
entry=entry1
assert(dofile("db.lua"))

END()
```

9.2   Markov Chain Algorithm

Our second example is an implementation of the *Markov chain algorithm*, described by Kernighan & Pike in their book *The Practice of Programming* (Addison-Wesley, 1999). The program generates random text, based on what words may follow a sequence of n previous words in a base text. For this particular implementation, we will fix $n = 2$.

The first part of the program reads the base text and builds a table that, for each prefix of two words, gives an array with the words that may follow that prefix. Each prefix is represented by the 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 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"},
}
```

To insert a new word in a list, we use the following function:

```
function insert (index, value)
  if not statetab[index] then
    statetab[index] = {n=0}
```

```
    end
    tinsert(statetab[index], value)
end
```

We first check whether that prefix already has a list; if not, we create an empty one. Then we use the pre-defined function `tinsert` to insert the value at the end of the list.

To build this 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. The `prefix` variable is re-initialized; then, for each prefix, we randomly choose a next word from the list of valid next words. Below we show the whole program.

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

function insert (index, value)
  if not statetab[index] then
    statetab[index] = {n=0}
  end
  tinsert(statetab[index], value)
end

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

-- build table
statetab = {}
w1 = NOWORD; w2 = NOWORD
file = read("*all")        -- read the whole file at once
gsub(file, "(%S+)", function (w)        -- for each word
  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)]
  local r = random(list.n)      -- choose a random item from list
  local nextword = list[r]
  if nextword == NOWORD then return end
  write(nextword, "\n")
```

```
  w1 = w2; w2 = nextword
end
```

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.*"

10. The Stand-Alone Interpreter

to do: explain its options and different ways to pass arguments to a script

Part II

Tables and Objects

Tables in Lua are not a data structure mechanism; they are *the* data structure mechanism. 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*.

Lua also offers a powerful way to denote tables, the so called *constructors*. Constructors allow Lua to be used as a powerful data-representation language. (One user called this mechanism "XML ahead of its time".) When you represent data with constructors, you do not need parsers or other facilities to read that data. Data files are actually Lua programs, and to read them you only have to "dofile" them.

But tables are more than data structuring and representation. Because tables can also store functions in its fields, they are also a mechanism to structure programs. Tables can implement modules and objects in many different ways.

11. Data Structures

In traditional languages, such as C and Pascal, most data structures are implemented with arrays and lists ("lists = records + pointers"). Although we can implement arrays and lists using 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 some typical data structures with tables, and provide some examples of their use. We will start with arrays, records, 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 we 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 you need. Usually, when you initialize the array you indirectly define its size. 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 $1 \ldots 1000$ will return **nil**, instead of 0.

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. Many functions in the Lua libraries (e.g. `sort` and `tinsert`) adhere to this convention, and so, if your arrays also start with 1, you will be able to use those functions directly.

We can use constructors to create and initialize arrays in a single expression:

```
squares = {0, 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`, `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
  while j <= M do
    mt[i][j] = 0
  end
end
```

Because tables are objects in Lua, you have to explicitly create each row to create a matrix. At 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 diagonal matrix changing the line

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

in the previous example to

```
    for j=i,M do
```

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

The second way to represent a matrix in Lua is 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 with the second index. With this approach, our matrix of zeros with dimensions `N`, `M` would be created with

```
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, positions like ("a:", "b") and ("a", ":b") would collapse into a single position "a::b"). When in doubt, you can use a control character like "\0" to separate the indices.

Sparse Matrices

Quite often, applications use a *sparse matrix*, a matrix wherein most elements are 0 or nil. For instance, you can represent a graph through 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 100 million entries, but approximately only 50 thousand of them will not be nil. Many books of data structures discuss at length how to implement such sparse matrices without wasting 400MB of memory, but you do not need these 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 10 thousand tables, each one with about 5 elements, with a grand total of 50 thousand entries; with the second representation, you will have a single table, with 50 thousand entries on it. Whatever the representation, you only need space for the non nil elements.

11.3 Linked Lists

Because tables are dynamic entities, it is very 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 some value v, we do

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

Other kinds of lists, such as double-linked lists or circular lists, are also easily implemented. However, you seldom need these structures in Lua, because usually there is a better way to represent your data without using lists. For instance, a stack is better represented with an array, with the field n pointing to the top. We have already seen such implementation, when we discussed the tinsert and tremove functions.

11.4   Queues and Double Queues

Although we can implement queues trivially using `tinsert` and `tremove`, such implementation can be too slow for large structures. We can do much better using two indices, one for the first and another for the last element:

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

Now, we can insert or remove an element in 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 `List_pushright` and `List_popleft`, both `first` and `last` will increase continually. However, because "arrays" in Lua are actually hash tables, there is no difference whether you index them from 1 to 20 or from 16777216 to 16777236. And because Lua uses doubles 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 will 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 every time they need 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, a very 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"] = 1,
  ["end"] = 1,
  ["function"] = 1,
  ["local"] = 1,
}

if reserved[w] then
  -- 'w' is a reserved word
  ...
```

(Because `while` is a reserved word in Lua, we cannot use it as an identifier. So, 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 i=i,getn(list) do
    set[list[i]] = 1
  end
  return set
end

reserved = Set{"while", "end", "function", "local", }
```

12. Metafiles and Persistence

When dealing with data files, usually it is much easier to write the data than to read it back. When we write a file, we have full control of what is going on. When we read a file, on the other hand, we do not know what to expect. Besides all kinds of data that a correct file may contain, a good program should also handle bad files gracefully. Because of that, coding good read routines is always difficult.

As we have seen in a previous example, table constructors provide a good alternative for file formats. The technique is to write our data file as Lua code that, when ran, 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. In Chapter **??**, we have seen how to read CSVs (Comma-Separated Values) in Lua. If our data file is already in CSV format, we have little choice. But 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. Therefore, instead of writing

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

in our data file, we could 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 such 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:

```
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.)

```
authors = {}        -- set to collect authors
function Entry (b) authors[b[1]] = 1 end
dofile("data")
for name,_ in 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, this is not a coincidence. BiBTeX was one of the inspirations for Lua's constructor syntax.) This format is called a *self-describing* data format, because each piece of data has attached to it a short description of its meaning. Self-describing data are more readable (by humans, at least) than CSV or other compact notations; they are easy to edit by hand, when necessary; and they allow us to make small modifications in format without having to change the data file. For instance, we can add a new field only by changing 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

```
authors = {}        -- set to collect authors
function Entry (b) authors[b.author] = 1 end
dofile("data")
for name,_ in 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 the `Entry` function:

```
function Entry (b)
  if b.author then authors[b.author] = 1 end
end
```

12.1 Persistence

Frequently we need some kind of *persistence*, that is, a way to save internal values of a program into a file for future use. (Currently this is also called *serialization*, after Java mechanism for persistence.) Again, we can use a Lua chunk to represent persistent data, in such a way that, when we run the chunk, it reconstructs the saved values into the current program.

Usually, if we want to restore the value of a global variable, our chunk will be something like `varname = ...`, where `...` is the Lua code to create the value. Let us see how to write that code. For a numeric value the task is trivial:

```
function serialize (o)
  if type(o) == "number" then
    write(o)
  else ...
end
```

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

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

Do not do that! Double square brackets are very good for hand-written strings, but not for automatically generated ones. If the string you want to save is something like §]]..execute('rm *.*')..[[, your final chunk will be

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

The user will have a bad surprise trying to reload 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 escape 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
    write(o)
  elseif type(o) == "string" then
    write(format("%q", o))
  else ...
end
```

Our next task is to save tables. There are many ways to do that, according to what restrictions we assume about the table structure. No single algorithm is good for all cases. Simple tables not only need simpler algorithms, but also the resulting file can be more aesthetic.

Our first try is as follows:

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

Despite its simplicity, this function does a reasonable job. It even handle nested tables (that is, tables with other tables inside), as long as the table structure is a tree (no shared sub-tables and no cycles). A small aesthetic improvement would be to ident eventual nested tables; you can try that as an exercise.

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

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

to

```
    write("  [")
    serialize(k)
    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\""
}
```

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

Saving Tables with Cycles

To handle tables with generic topology (that is, with cycles) we need a different approach. Constructors cannot represent such tables, so we will not use them. To represent a cycle 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 avoid cycles. We will use an extra table for that. 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 basic_serialize (o)
  if type(o) == "number" then
    return o
  else    -- assume it is a string
    return format("%q", o)
  end
end
```

The next function does the hard work. The `saved` argument is the table that keep track of tables already saved.

```
function save (name, value, saved)
  write(name, " = ")
  if type(value) == "number" or type(value) == "string" then
    write(basic_serialize(value), "\n")
  elseif type(value) == "table" then
    if saved[value] then    -- value already saved?
      write(saved[value], "\n")   -- use its previous name
    else
      saved[value] = name   -- save name for next time
      write("{}\n")      -- create a new table
      for k,v in value do      -- save its fields
        local fieldname = name.."["..basic_serialize(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["y"] = 2
a["x"] = 1
a[2] = a
a["z"] = {}
a["z"][1] = 3
a["z"][2] = 4
a["z"][3] = 5
a[1] = a["z"]
```

(The actual order of some assignments may vary, as it depends on a traversal order.)

If we want to save several values, and they share 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"
```

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

```
local saved = {}
save('a', a, saved)
save('b', b, saved)
```

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 many 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, for each function, gives its name); etc. Lua gives you the power, you build the mechanisms.

13. Namespaces & Packages

Many languages provide mechanisms to organize their space of global names, such as *modules* in Modula, *packages* in Java and Perl, and *namespaces* in C++. Each of these mechanisms have different rules regarding the use of elements declared inside a module, visibility rules, and other details. But 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.

In Lua, such packages (or modules, or namespaces) are easily implemented with tables: We only have to put our identifiers as keys in a table, instead of as global variables. The main point here is that functions can be stored inside table, just as any other value. For instance, suppose we are writing a library to manipulate complex numbers. We represent each number as a table, with fields `r` (real part) and `i` (imaginary part). To avoid polluting the global namespace, we will declare all our new operations in a table, that acts as a new package:

```
Complex = {

  new = function (r, i) return {r=r, i=i} end,

  add = function (c1, c2)
          return {r=c1.r+c2.r, i=c1.i+c2.i}
        end,

  sub = function (c1, c2)
          return {r=c1.r-c2.r, i=c1.i-c2.i}
        end,

  mul = function (c1, c2)
          return {r = c1.r*c2.r - c1.i*c2.i,
                  i = c1.r*c2.i + c1.i*c2.r}
        end,

  inv = function (c)
          local n = c.r^2 + c.i^2
```

```
        return {r=c.r/n, i=c.i/n}
      end,

  i = {r=0, i=1},

}
```

With this definition, we can use any complex operation "qualifying" the operation name, as in

```
c = Complex.add(Complex.i, Complex.new(10, 20))
```

The use of tables for packages does not provide exactly the same functionality as provided by real packages. For instance, in Lua, a function that calls another function inside the same package must qualify the name of the called function. Moreover, it is up to you to follow the rules; there is neither a fixed way to implement packages, nor fixed operations to manipulate them.

On the other hand, exactly because packages are plain tables in Lua, we can manipulate them like any other table, and use the whole power of Lua to create many extra facilities. There are endless possibilities.

We do not need to define all 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
```

If we are going to use some operations often, we can give them global (or local) names:

```
add = Complex.add
local i = Complex.i

c1 = add(Complex.new(10, 20), i)
```

It is easy to write a function that opens the whole package, putting all its names in the global namespace:

```
function opennamespace (ns)
  for n,v in ns do setglobal(n,v) end
end

opennamespace(Complex)
c1 = mul(new(10, 20), i)
```

If you are afraid of name clashes when opening a package, you can add a prefix to all imported names:

```
for n,v in Complex do
  setglobal("C_"..n, v)
end)
```

```
c1 = C_mul(C_new(10, 20), i)
```

Because packages themselves are tables, we can even nest packages; that is, we can create a whole package inside another one. However, such facility is seldom necessary.

Usually, all names inside a package are *exported*; that is, they can be used by any client of the package. Sometimes, however, it is useful to have private names in a package. A naive (but useful nevertheless) way to do that is to adopt some convention for private names, such as to start all private names with an underscore. If you want to enforce privacy, you can declare the private name as a local variable in your file. With such solution, you must use upvalues whenever you need to use a private name inside a function; see the next example:

```
-- 'check_complex' is a private function
local check_complex  = function (c)
  local OK = (type(c) == "table") and
             tonumber(c.x) and tonumber(c.y)
  if not OK then error("bad complex number") end
end

Complex = {

  add = function (c1, c2)
        %check_complex(c1);  %check_complex(c2);
        return {r=c1.r+c2.r, i=c1.i+c2.i}
      end,

}
```

Typically, when we write a package, we put its whole code in a unique 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 `dofile("complex.lua")` will open the package. To avoid waste when someone loads a package that is already loaded, we write the whole package inside a conditional:

```
if not Complex then
Complex = {
   ...
}
end
```

Now, if you run `dofile("complex.lua")` when `Complex` is already defined, the whole file is skipped.

In the next chapter we will see how we can use tag methods to add more facilities to our package mechanism, such as how to protect a package (making it "read-only") and how to load a package automatically, when it is first referred.

14. Objects

A fundamental feature of our understanding of the world is that we organize our experience as a number of distinct objects (tables and chairs, bank loans and algebraic expressions, polynomials and persons, transistors and triangles, etc.); our thought, language, and actions are based on the designation, description, and manipulation of these objects, either individually or in relationship with other objects. (Tony Hoare, 1966)

A table in Lua is an "object" in more than one sense. Like objects, tables have a state. Like objects, tables have an identity that is independent of their particular values; particularly, two objects (tables) with exactly the same value can be different objects, while the same object can have different values at different times. Like objects, tables have a life cycle that is independent of who created them or where they were created.

Objects have their own operations. Tables also can have operations:

```
Account = {balance = 0}
function Account.withdraw (v)
  Account.balance = Account.balance - v
end
```

Such 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, such 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 this particular global variable. For instance, if we change the name of this object, the `withdraw` function does not work any more:

```
a = Account; Account = nil
a.withdraw(100.00)   -- ERROR!
```

This restriction violates the previous principle that objects have independent life cycles.

A much better approach is to operate over the *receiver* of the operation. For that, we would have to define our method with an extra parameter, that tells over which object it has to operate. Such parameter is usually called *self* or *this*:

```
function Account.withdraw (self, v)
  self.balance = self.balance - v
end
```

Now, when we call the method we also have to tell over which object it has to operate:

```
a1 = Account; Account = nil
...
a1.withdraw(a1, 100.00)    -- OK
```

With the use of a self parameter, the same method can be used for many objects:

```
a2 = { balance=0, withdraw = a1.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 partially 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). In Lua, we can also hide this parameter, using a special syntax for methods. 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. It is only a syntactical facility, although a very 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.

```
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, privacy, and inheritance. Let us tackle the first problem: How can we create several objects with similar behavior? Particularly, how can we create several accounts?

Our first solution to this problem is based on *clones*. If we have an `account` object, we can create a new one by coping the operations from the original object to the new one. We can do this field by field:

```
function new_account ()
  return {
    deposit = Account.deposit
    withdraw = Account.withdraw
    balance = Account.balance
  }
end
```

A better approach is to use a generic cloning algorithm, that copies all fields from an object to another:

```
function clone (old)
  local new = {}
  for k,v in old do new[k] = v end
  return new
end
```

We can still improve this function by providing an optional initialization for some fields:

```
function createObj (old, init)
  local new = init or {}
  for k,v in old do
    if not new[k] then new[k] = v
  end
  return new
end
```

With this new function, only absent fields are copied (or "inherited") from the old object to the new. Now, we can easily create multiple account objects:

```
a1 = createObj(Account)
a2 = createObj(Account, {balance = 350.00})
a1:deposit(1000.00)
a2:deposit(400.00)
print(a2.balance)    --> 750
```

The `a1` object inherits not only its methods, but also a default balance from the `Account` object; `a2`, on the other hand, defines its own initial balance, and so this field is not copied from `Account`.

How good is this solution? That depends on the specific problem at hand. The overhead is not too big, because both the strings (the keys of each table object) and the methods (the values for each key) are shared among all objects of the same "class". Nevertheless, each object must keep privately the association between keys and values. If we are going to create many thousands of objects, each one with dozens of methods, then this solution can prove itself too expensive, both in time (to create the clones) and in space (to store each clone). But if we are going to create a few objects, or if each object has a few methods, such use of clones offers a simple and affordable solution.

In some highly dynamic systems, you may need to change or add methods to an object during its lifetime. Using clones, each object is completely independent from all the others; when you change a method of one of them, none of its clones are affected. (Of course, objects cloned *after* the change will inherit it.) If this is what you want, good; otherwise, you need a different way to share behavior among classes of objects. Soon we will see how we can use tag methods to do that.

Despite its simplicity, our basic design for objects already presents some of the main features of object-oriented programming: *polymorphism* and *late binding*. Different objects, with different implementations, can offer the same interface. Because each object carries its own operations, we can manipulate them in an uniform way. For instance, let us consider a set of objects representing geometric figures, such as rectangles, circles, and the like. Each object must have at least an operation to compute its perimeter and another for its area:

```
Rectangle = {x = 0, y = 0, h = 1, w = 1}

function Rectangle:perimeter ()
  return self.h*2 + self.w*2
end

function Rectangle:area ()
  return self.h*self.w
end

Circle = {x = 0, y = 0, r = 1}

function Circle:perimeter ()
  return 2*PI*self.r
end

function Circle:area ()
  return PI*self.r^2
end
```

Now, if we have a list `l` of non-overlapping figures, like

```
l = {
  createObj(Rectangle, {x=10, y=20}), -- 'h' and 'w' inherited
  createObj(Circle, {r=20}),          -- 'x' and 'y' inherited
  createObj(Circle, {x=100, y=-10, r=10})
}
```

we can compute the total area with the following code:

```
function total_area (l)
  local a = 0
  for i=1,getn(l) do
    a = a + l[i]:area()
  end
  return a
end
```

The key point in this function is the call `l[i]:area()`. When we write such
call, we do not know in advance what function will be called. Because our list of
figures has different kinds of figures, for each `i` the call `l[i]:area()` will invoke
a different function. It is up to each object to provide an adequate method
to compute its area. Even if we create new classes of figures, the `total_area`
function will work, as long as each new class provides a correct `area` method.

14.1   Privacy

Many people consider privacy to be an integral part of an object-oriented lan-
guage: 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 *instance variable*) or a method is visible outside the object. Other lan-
guages, such as Smalltalk, makes all variables private and all methods public.
The first OO language, Simula-67, did not offer any kind of protection.

The main design for objects in Lua, that we shown previously, do not offer
privacy mechanisms. Partially, this is a consequence of our use of a very 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 large periods of time. Quite the opposite,
Lua aims small programs, usually part of a larger system, typically developed
by one or a few programmers, or even by non programmers. So, Lua avoids
redundancy and artificial restrictions. If you do not want to access something
outside an object, just *do not do it.*

Nevertheless, another aim of Lua is to be flexible, and to offer meta-mechanisms
through which we can emulate many different mechanisms. Therefore, 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 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, and another for its operations, or its *interface.* The object itself is accessed through the second table, that is, through the operations that comprise 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 as upvalues on the object methods. For instance, to represent our bank account with this design, we could create new objects running the next "factory" function:

```
function new_account (initial_balance)
  local self = {balance = initial_balance}
  local withdraw = function (v)
                     %self.balance = %self.balance - v
                   end
  local deposit = function (v)
                     %self.balance = %self.balance + v
                   end
  local get_balance = function () return %self.balance end
  return {
    withdraw = withdraw,
    deposit = deposit,
    get_balance = get_balance
  }
end
```

First, the function creates a table to keep the 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 all those methods do not get `self` as an extra parameter; instead, they are directly linked to that table through upvalues. Because there is no extra argument, we do not use the comma syntax to manipulate such objects. The methods are called just like any other function:

```
acc1 = new_account(100.00)
acc1.withdraw(40.00)
print(acc1.get_balance())      --> 60
```

This design gives full privacy to anything stored in the `self` table. After the `new_account` function returns, there is no way you can gain direct access to that table. It can only be accessed through functions created inside `new_account`. Although our example puts only one instance variable into the private table, we can store all private parts of an object in this table, including private methods. 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 new_account (initial_balance)
  local LIM = 10000.00
  local self = {
    balance = initial_balance,
    extra = function (self)
      if self.balance > %LIM then
        return self.balance*0.10
      else
        return 0
      end
    end
  }
  local get_balance = function ()
    return %self.balance + %self:extra()
  end
  ...
```

Again, there is no way for any user to access the `extra` function directly.

14.2 Delegation

Back to our first design for objects, we still have some problems to solve. First, clones are not efficient when we have lots of objects with lots of methods. Second, we still do not have inheritance. We will solve these two problems with the same mechanism: *delegation*. To implement delegation, we will need tag methods.

Tag methods are the subject of a whole part of this book, but here we will see only a small part of this mechanism, and we will see it through a restricted point of view, which is enough to solve our problems now.

What happens when Lua indexes a table with an absent index? For instance, if `a = {x=1}`, what is the value of `a.y`? It is **nil**. But this is not the whole true, because we can change this: We can define a function to be called by Lua in such an event, so that the final result of the indexing operation, instead of **nil**, will be the value returned by this function. This function is what we call a *tag method*, and this specific event is called an *index* event. In other words, an index tag method is a function that is called when Lua cannot find a value for a given index in a table.

To set a tag method, we use the `settagmethod` function. It takes three arguments: A tag, an event name, and a function. The tag identifies the objects that will use this tag method, so that we can have different tag methods for different tables. For now, we will use only the default tag for tables, which is given by the call `tag` (that is, `tag` applied over a table); this tag refers to all "regular" tables in a program. The event name identifies what kind of event this tag method will handle; for now, it will be `"index"`. Finally, the third argument is the function to be called when the given event happens over an object with

the given tag. For the index event, the function is called with two parameters: the table that has been indexed and the (absent) index.

Before using tag methods for our real purpose, let us see a simple example. The next chunk changes the default value for absent indices in a table from **nil** to zero:

```
settagmethod(tag{}, "index", function (t, i) return 0 end)

a = {x=1, y=10}
print(a.x, a.y, a.z)      --> 1    10    0
print(a[-5])              --> 0
a[-5] = "hi"
print(a[-5])              --> hi
```

Now we are equipped to implement delegation. The idea is simple: If we cannot find a value for an index in a given object, we will look for this value in some other object, as if the first object delegates the index operation to the second object. There are many ways for an object to choose the other object to which it will delegate its accesses. Here we will adopt a simple solution: When an object wants to delegate its accesses to another object, it will keep this other object in a pre-defined field, that we will call *parent*. So, the task of our tag method is to check whether the indexed object has a "parent" and, if so, to apply the index over this parent. The "magic" code is as follows:

```
function delegate (t, i)
  local parent = rawget(t, "parent")
  if type(parent) == "table" then
    return parent[i]
  else
    return nil
  end
end

settagmethod(tag{}, "index", delegate)
```

The only tricky part is the call to `rawget`. At first, it seems that we could write `t.parent` there; after all, what we want is the field `"parent"` from table `t`. However, if `t` did not have such field, Lua would call the index tag method (which is this function) again, and again it would try to get the field `"parent"` from table `t`. To avoid this loop, we use the `rawget` function: The call `rawget(t, i)` is a "raw" version of `t[i]`, where only the primitive indexing operation is performed, without tag methods.

After that, the `delegate` function is straightforward. It checks whether `parent` is a table and, if it is, it index that table with the original index `i`; otherwise it returns **nil**, as the original behavior. (If `t` does not have a parent, `parent` will be **nil**, and so this test will fail.) Notice that we index the parent table with the usual subscription operation `[i]`, and not with `rawget`. Therefore,

if the parent object also does not have this index, but has a parent, `delegate` will be called again, and the search will continue upward until arriving in an object with a value for that index or without a parent.

14.3 Classes & Inheritance

Finally, we know all we need to know to fully implement objects in Lua. Let us go back to our first example of a bank account. Let us assume that our program already has the definition for an object `Account`; our program also has installed the `delegate` function as its index tag method, as we described previously. Now suppose we define a new object as follows:

```
a1 = {balance = 100.00, parent = Account}
```

and then do a call like

```
a1:deposit(150.00)
```

which is equivalent to

```
a1.deposit(a1, 150.00)
```

Because `a1` does not have a field `"deposit"`, the index tag method (the `delegate` function) will be called. That function will look for a field `"parent"` in `a1`, will find it, and will get the field `"deposit"` from this parent. So, the result of `a1.deposit` will be the `Account.deposit` function. Lua will then call this function, with `self = a1` and $Tv = 150.00$, and finally the value of `a1.balance` will be incremented. The same sequence of actions happens for a call to `a1.withdraw`, or to any other method available in object `Account`.

Now, any object with a `parent` field pointing to `Account` has available all methods defined for `Account`. Therefore, `Account` acts as a *prototype* for these objects; for most practical purposes, it also acts as a *class*. Let us repeat here this class definition:

```
Account = {balance=0}

function Account:withdraw (v)
  self.balance = self.balance - v
end

function Account:deposit (v)
  self.balance = self.balance + v
end
```

Moreover, as a class, `Account` should also have a method to create new instances:

```
function Account:new (balance)
  return {balance=balance, parent=self}
end
```

With those definitions, we can easily create and manipulate multiple accounts:

```
a1 = Account:new(100.00)
a2 = Account:new(500.00)
a1:deposit(50.00)
print(a1.balance)          --> 150
```

What are the costs of this solution? Now, each new object has only one extra field, its `parent`, regardless the number of methods it inherits. If we change or add new methods to the `Account` class, its instances will inherit these changes immediately. On the other hand, method calls are less efficient than before, because each call must go through the `"index"` tag method to find the function to be called.

to do: work in progress

Part III

The Standard Libraries

A great part of the power of Lua comes from its libraries. This is not by chance. One of the main strengths of Lua is its extensibility through new functions. Many features contribute to this strength. Dynamic typing allows a great degree of polymorphism (e.g., functions that operate over any kind of tables, such as `sort` and `tinsert`). Automatic memory management simplifies interfaces, because there is no need to decide who is responsible to allocate and deallocate memory, and how to handle overflows (e.g., functions returning strings, such as `gsub`). First order functions and anonymous functions allow a high degree of parameterization, making functions more versatile (e.g., functions `sort` and `gsub`).

Lua comes with a set of standard libraries. When installing Lua in a strongly limited environment, such as embedded processors, it may be wise to choose carefully which libraries you really need. Moreover, if the limitations are really hard, it is very easy to go inside the libraries source code and choose one by one which functions should be kept. Remember, however, that Lua is rather small (even with all standard libraries), and in most systems you can use the whole package without any concerns.

The *basic library* comprises functions that provide some kind of *reflexive* facility (e.g., `type` and `call`), and functions that could be written in Lua itself, but are so common that deserve to be always at hand (e.g., `sort`, `assert`).

The other standard libraries are a mathematical library, with trigonometric and logarithmic functions, a pseudo-random number generator, and other related functions; a library for string manipulation, comprising pattern matching; and the system library, with functions for time/date, input/output, file manipulation, and other functions related to the operating system.

In the following chapters, we will tour through these libraries. Our purpose here is not to give the complete specification of each function, but to show you what kind of functionality these libraries can provide to you. We may omit some subtle options or behaviors for clarity of exposition. The main idea is to spark your curiosity, which then will be satiated by the manual.

15. Basic Library

We classify the basic functions in five main classes:

- Functions to check and convert types.

- Functions to manipulate tables.

- Functions to manipulate the global environment.

- Functions to execute Lua code.

- Functions to manipulate tags and tag methods. These functions will be explained in Chapter IV, where we discuss tag methods.

15.1 Functions to Check and Convert Types

We have already seen all these functions before. The function `type` returns a string describing the type of its argument. Its result can be `"function"`, `"nil"`, `"number"`, `"string"`, `"table"`, and `"userdata"`.

The function `tostring` returns a string describing a value. Its main use is for human consumption; for instance, the function `print` uses it to know how to show a value. For strings it returns the string itself. For numbers, it returns the number converted to a string in a reasonable format (for a complete control over the conversion from numbers to strings, such as the number of significant digits, you should use the function `format`, from the string library). For other types, it returns the type name plus an internal identification (such as the hardware address of the object).

```
> print(print)
function: 0040A5B0
> print({})
table: 00672B60
> print(_INPUT)
userdata: 00420AB0
> print("Rio")
Rio
```

```
> print(1.0e4)
10000
```

Because the `print` function uses `tostring` for its formatting, you can rede-
fine `tostring` if you want to print values with different formats. As an example,
you can use the following code to make Lua print integer numbers in hexadeci-
mal:

```
function tostring (o)
  if type(o) == "number" and floor(o) == o then
    return format("%X", o)
  else
    return %tostring(o)  -- uses old version for other values
  end
end

print(28)      --> 1C
```

As another example, suppose that your program uses tables to represent
points, and you want to print those points in a more informative way. Then you
can redefine `tostring` as follows:

```
function tostring (o)
  -- value is a table with both fields 'x' and 'y' numeric?
  if type(o) == "table" and type(o.x) == "number" and
     type(o.y) == "number" then
    return format("(%g,%g)", o.x, o.y)
  else
    return %tostring(o)    -- default format
  end
end
```

With this code, your `print` now behaves as illustrated below:

```
a = {x=20, y=10}
print(a)        --> (20,10)
```

Of course, these two previous redefinitions of `tostring` are not incompatible;
when redefining a function to handle new special cases, you can "chain" as many
redefinitions as you need. All you have to do is to call the previous version for
cases that your new definition does not handle. Whenever a function cannot
handle a given argument, it passes that argument to the next function in the
chain (that is, the previous version of that function). Nevertheless, you must
exercise a little restrain when creating such chains, for the sake of both efficiency
and mental sanity.

The last function of the class is `tonumber`. Besides simply converting strings
to numbers, this function has two other uses. First, as we have already seen,
it can be used to check whether a string is a valid numeral. Second, it can be
used to convert numerals written in other bases; in this case, a second optional
argument specifies the base:

```
> print(tonumber("10010", 2))
18
> print(tonumber("1EF", 16))
495
> print(tonumber("13", 2))        -- not a valid binary number
nil
> print(tonumber("j0", 20))       -- 19*20
380
```

The base can be any number between 2 and 36; the letters correspond to the digits from 10 (`A` or `a`) to 35 (`Z` or `z`).

15.2   Functions to manipulate tables

Frequently we use tables to represent arrays or lists. In those cases, the table has indices in the range $1 \ldots n$. The Lua libraries offer special support for this use of tables. For most functions that manipulate arrays (that is, that manipulate tables that represent arrays), we need to define the size of an array. As we have seen, tables in Lua have no fixed size. So, if you are going to see a table as a list or an array, what is the list length, or the array size? The `getn` function defines the size of an array: If the table has a field `n` with a numeric value, that value is the size of the array. Otherwise, the size of the array is its greatest numeric index with a non **nil** value.

```
print(getn{n=20})               --> 20
print(getn{"a", "b", "c"})      --> 3
print(getn{"a", nil, "c"})      --> 3
print(getn{"a", nil, nil})      --> 1
print(getn{"a", "b", "c"; n=2}) --> 2
print(getn{[1000] = "x"})       --> 1000
```

Two useful functions to manipulate arrays are `tinsert` and `tremove`. The first one inserts an element in a given position of an array, moving up other elements to open space. Moreover, `tinsert` increments (or creates) the field `n` of the array, to reflect its size. For instance, if `a` is the array `{10 20 30}`, after the call `tinsert(a, 1, 15)` a will be `{15 10 20 30; n=4}`. As a special (and frequent) case, if we call `tinsert` without a position, the element is inserted in the last position of the array (and, therefore, no elements are moved). As an example, the following code reads a file line by line, storing all lines in an array:

```
a = {n=0}
while 1 do
  local line = read()
  if line == nil then break end
  tinsert(a, line)
end
print(a.n)         --> (number of lines read)
```

The function `tremove` is a kind of inverse of `tinsert`. It removes (and returns) an element from a given position in an array, moving down other elements to close space, and decrementing the field `n`. When called without a position, it removes the last element of the array.

With those two functions, it is trivial to implement stacks, queues and double queues. Such structures can be initialized as `a = {n=0}` (or simply `a = {}`). A push operation is equivalent to `tinsert(a, x)`, while a pop is `tremove(a)`. To insert at the other end of the structure, you use `tinsert(a, 1, x)`; the same with `tremove`. When we manipulate one side of the structure, such operations are not very efficient for big structures (more than one hundred elements, say), as elements must be moved up and down. However, because these functions are implemented in C, these loops are not too expensive, and frequently this implementation is more than enough. In ?? we will see better implementations of queues.

To iterate over the elements of an array, you should use a numeric `for`, with a range from 1 to `getn` of the array. For instance, after reading all file lines in an array `a`, we can print the whole file with

```
for i=1,getn(a) do
  write(a[i], "\n")
end
```

You may be tempted to use the other `for` in such loops:

```
-- bad code
for i,l in a do
  write(l, "\n")
end
```

This is a bad idea. First, such loop may print the lines out of order. Moreover, it will print the field `"n"`, too.

Another useful function over arrays is `sort`, as we have seen before. It receives the array to be sorted, plus an optional order function. This function should receive two arguments and returns true if the first argument has to come first in the sorted array. If this function is not provided, the default less-than operation (corresponding to the `<` operator) is used.

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 built 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. Because these names are in the keys of the table, and not in the values, you cannot sort them directly. First, you must create an array with those names, then sort it, and finally print the result:

```
a = {}
for n, _ in lines do tinsert(a, n) end
sort(a)
for i=1,getn(a) do print(a[i]) end
```

(The _ in the first `for` is just a dummy variable.)

As a more advanced solution, we can write a higher-order function that encapsulates these previous operations:

```
function foreachsorted (t, f)
  local a = {}
  for n, _ in t do tinsert(a, n) end
  sort(a)
  for i=1,getn(a) do f(a[i], t[a[i]]) end
end
```

Such function will apply `f` over each element of table `t`, following an alphabetical order. For instance, the call `foreachsorted(line, print)` will print

```
luaH_get        24
luaH_present    48
luaH_set        10
```

15.3   Functions to manipulate the global environment

Our next class of functions deals with global variables. Usually, assignment is enough for getting and setting global variables. However, often we need a form of meta-programming. A typical case is 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 such a variable, many programmers are tempted by something like

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

or

```
value = dostring("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 will achieve the desired result. However, such code involves 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 = getglobal(varname)
```

In a similar way, you can assign to a global variable whose name is dynamically computed:

```
setglobal(x, value)
```

Beware, however. Some programmers get a little excited with these functions, and end up writing code like

```
setglobal("a", getglobal("var1"))
```

which is just a complicated and inefficient way to write `a = var1`.

The `globals` function returns the table that keeps all global variables of your program. The indices of this table are the global variable names. Therefore, an alternative way to write `getglobal(varname)` is `globals()[varname]`; and you can write `setglobal(varname, newvalue)` as `globals()[varname] = newvalue`. (As we will see in Chapter ?, these forms are not exactly equivalent, as they may invoke different tag methods.)

The following code prints the names of all globals defined in your program:

```
for n, _ in globals() do print(n) end
```

In Chapter ? we will see more powerful uses for the `globals` function.

15.4   Functions to execute Lua code

Our last group of builtin functions deals with the execution of Lua code. Two of these functions are `dofile` and `dostring`, which we have seen already.

The function `dostring` is very powerful, and therefore must be used with care. It is an expensive function (when compared to some alternatives), and may result in incomprehensible code. Before you use it, make sure that there is no better way to solve the problem at hand (for instance with `getglobal`, `setglobal`, or `call`).

The most typical use of `dostring` 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 `dostring` to evaluate it. Note that `dostring` 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 = read()
result = dostring("return "..l)
print("the value of your expression is "..result)
```

Sometimes, you will need to evaluate a piece of external code many times (as in our example of plotting a function). Instead of repeating the `dostring`, you can make your program more efficient and cleaner by using `dostring` only once, to create a function that encapsulates the external code; for instance

```
print "enter the function to be plotted (with variable 'x'):"
local l = read()
local f = dostring("return function (x) return "..l.." end")
local i = 1
while i<=50 do
  print(strrep("*", f(i)))
  i = i+1
end
```

So, if the user writes `(x-20)^2`, `f` will be the function

```
function (x) return (x-20)^2 end
```

As an extra bonus of this approach, this single `dostring` will detect any syntactic errors in the external code; if there are any errors, `f` will get **nil** (the result of `dostring` when there is an error).

For a more professional program that runs external code, you may want to redefine the `_ERRORMESSAGE` function during the `dostring`, to present the user with a more user-friendly message in case of errors in the external code. (We have shown an example of such redefinition in Section 8.1.) Also, if the code cannot be trusted, you may want to temporarily redefine your global environment to avoid unpleasant side effects when running the code (more about this later??).

The third function to execute Lua code, `call`, calls a function with an array of arguments. It gets two mandatory parameters, the function to be called and the array of arguments. As a general rule, you can always write a function call such as

```
f(a1,a2)
```

as

```
call(f, {a1, a2; n=2})
```

although the last form is worse in all aspects. What makes `call` useful is that the array of arguments can be build dynamically, and can have a variable number of arguments.

For a realistic example, suppose you are debugging a C function, `foo` let us call it, and you want to print its arguments every time the function is called. A simple solution is to write a new function `foo`, that prints its arguments and then calls the old one:

```
function foo (a,b)
  print(a,b)
  return %foo(a,b)
end
```

Now, suppose that the function `foo` has optional arguments, so that you do not know how many arguments it gets in each call. You can define the new `foo` as a function with a variable number of arguments, but you cannot call directly the old function with those arguments. There is where the function `call` has its typical use:

```
function foo (...)
  call(print, arg)
  return call(%foo, arg)
end
```

Whatever are the arguments to `foo`, they are collected in table `arg` (remember the meaning of `...`). Then, `call` is used first to call `print`, and then to call the original `foo` with that arguments.

The function `call` has another typical use, as we discussed in Chapter **??**. Usually, `call` handles errors the same way that a normal function call: If an error occurs during the function call, the error is propagated, and the whole chunk ends. However, with the string `"x"` as its third argument, `call` works in a *protected* mode. In this mode, when an error occurs during the call, it is not propagated; instead, `call` returns **nil** to signal the error, besides calling the appropriated error handler. Moreover, a forth optional argument can be passed to `call`, which is used as a temporary error handler during the call.

16. The Mathematical Library

The math library comprises a standard set of math functions, such as trigonometric functions (`sin`, `cos`, `tan`, `asin`, `acos`, etc.), exponential and logarithms (`exp`, `log`, `log10`), round functions (`floor`, `ceil`), `max`, `min`, plus a global variable PI. The math library also defines the operator `^` (using tag methods) to work as the exponential operator.

All trigonometric functions work in degrees (and not radians). You can use the functions `deg` and `rad` to convert between degrees and radians. If you really want to work in radians, you can always redefine these functions:

```
sin = function (x) return %sin(deg(x)) end
asin = function (x) return rad(%asin(x)) end
...
```

The function to generate pseudo-random numbers, `random`, can be called in three ways. When called without arguments, it returns a pseudo-random real number with uniform distribution in the interval $[0, 1)$. When called with only one argument, an integer n, it returns an integer pseudo-random number x such that $1 \leq x \leq n$. For instance, you can simulate the result of a dice with `random(6)`. Finally, `random` can be called with two integer arguments, l and u, to return an integer pseudo-random number x such that $l \leq x \leq u$.

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 this 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 time of the day as a seed:

```
randomseed(date"%d%H%M%S")
```

The call to function `date` will return the concatenation of the current day, hour, minute, and second. (So, you will repeat your game scenario only if you start to play at exactly the same second of the same day, at least one month later.)

17. 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, nor examine its contents. The full power to manipulate strings in Lua is provided by the string library.

Some functions in the string library are quite simple: `strlen(s)` returns the length of the string `s`. `strrep(s, n)` returns the string `s` repeated `n` times; for instance, `strrep(".\n", 3)` is `".\n.\n.\n"`. You can create a string with 1M bytes (for tests, for instance) with `strrep("a", 2^20)`. `strlower(s)` returns a copy of `s` with the upper case letters converted to lower case; all other characters in the string are not changed (`strupper` converts to upper case). As a typical use, if you want to sort an array of strings regardless case, you may write something like

```
sort(a, function (a, b) return strlower(a) < strlower(b) end)
```

Both `strupper` and `strlower` follow the current locale. Therefore, if you work with the European Latin-1 locale, the expression `strupper("ação")` will result in `"AÇÃO"`.

The `strsub(s,i,j)` function 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 use also 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 `strsub(s, 1, j)` gets a *prefix* of the string `s` with length j; `strsub(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 the last call could be written as `strsub(s, j)`); and `strsub(s, 2, -2)` removes the first and last characters of a string: If `s` is `"[in brackets]"`, `strsub(s, 2, -2)` will be `"in brackets"`.

The conversion between characters and their internal numeric representations is done with `strchar` and `strbyte`. The function `strchar` gets zero or more integers, converts each one to a character, and returns a string concatenating all those characters. The function `strbyte(s, i)` returns the internal numeric representation of the i-th character of the string `s`; the second argument

111

is optional, so that a call `strbyte(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(strchar(97))                  -->  a
i = 99; print(strchar(i, i+1, i+2)) -->  cde
print(strbyte("abc"))               -->  97
print(strbyte("abc", 2))            -->  98
print(strbyte("abc", -1))           -->  99
```

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

The function `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 formated string. A simple directive is the character `%` plus a letter that tells how to format the argument: `d` for a decimal number, `h` for hexadecimal, `o` for octal, `f` for a floating point number, `s` for strings, plus other variants. Between the `%` and this letter, a directive can have other options, which control the details of the format. For a complete description of these directives, see the manual.

```
print(format("pi = %.4f", PI))     --> pi = 3.1416
d = 5; m = 11; y = 1990
print(format("%02d/%02d/%04d", d, m, y))  --> 05/11/1990
tag, title = "h1", "a title"
print(format("<%s>%s</%s>", tag, title, tag))
--> <h1>a title</h1>
```

In the first example, the `%.4f` means a floating point number with 4 digits after the decimal point. In the second example, the `%02d` means a decimal number (d), with at least 2 digits and zero padding; the directive `%2d`, without the 0, would use blanks for padding.

17.1   Pattern Matching

The most powerful functions in the string library are `strfind` (*String Find*) and `gsub` (*Global Substitution*). Both have multiple uses besides those implied by their names, and are based on *patterns*.

to do: brief comparison with POSIX regexp (why Lua does not use them?)

The basic use of `strfind` is to search for a pattern inside a given string (called the *subject* string). The function returns where it found the pattern, or **nil** if it could not find it. The simplest form of a pattern is a word, that matches only a copy of itself. For instance, the pattern `hello` will search for the substring `"hello"` inside the subject string. When the `strfind` function

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 = strfind(s, "hello")
print(i, j)                    --> 1    5
print(strsub(s, i, j))         --> hello
print(strfind(s, "world"))     --> 7    11
i, j = strfind(s, "l")
print(i, j)                    --> 3    3
print(strfind(s, "lll"))       --> nil
```

If a match succeed, a `strsub` of the values returned by `strfind` will return the part of the subject string that matched the pattern.

The `strfind` 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 repeatedly search for a new pattern, starting at 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 1 do
  i = strfind(s, "\n", i+1)    -- find 'next' newline
  if i == nil then break end
  tinsert(t, i)
end
```

The basic use of `gsub` is to substitute all occurrences of a given pattern inside a string by another string. This function gets three arguments: The subject string, the pattern, and the substitution string:

```
s = gsub("Lua is cute", "cute", "great")
print(s)         --> Lua is great
s = gsub("all lii", "l", "x")
print(s)         --> axx xii
s = gsub("Lua is great", "perl", "tcl")
print(s)         --> Lua is great
```

An optional fourth argument limits the number of substitutions to be made:

```
s = gsub("all lii", "l", "x", 1)
print(s)           --> axl lii
s = gsub("all lii", "l", "x", 2)
print(s)           --> axx lii
```

Remember that strings in Lua are immutable. The `gsub` 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 `gsub(s, "a", "b")` and then 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 = gsub(s, "a", "b")
```

The `gsub` function also returns as a second result the number of times it made the substitution. So, an easy way to count the number of spaces in a string is

```
_, count = gsub(s, " ", " ")
```

(Remember, the `_` is just a dummy variable name.)

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(strsub(s, strfind(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(gsub("hello, up-down!", "%A", "."))  --> hello..up.down.
```

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, and `%%` matches the `%` itself. You can use the escape `%` not only for the magic characters, but for any non alphanumeric character. When in doubt, play safe and put an escape.

For Lua, patterns are regular strings. They have no special treatment, and follow exactly the same rules as other strings. Only inside the `strfind` and

`gsub` functions they are interpreted as patterns, and only then the `%` works 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.

Char-sets allow 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 class `[01]` matches binary digits, and the class `[%[%]]` matches square brackets. To count the number of vowels in a text, you can write

```
_, nvow = gsub(text, "[aeiou]", "")
```

You can also include character ranges in a char-set, by writing the first and the last characters of the range separated by an hyphen. You seldom will need this facility, since most useful ranges are already pre-defined; for instance, `[0-9]` is better written as `%d`, and `[0-9a-fA-F]` is the same as `%x`. But, if you need to find an octal digit, then you may prefer `[0-7]`, instead of an explicit enumeration (`[01234567]`). You can also 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 single classes with its upper version; `%S` is better than `[^%s]` to find a non-space character.

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 "repetitions")

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(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 = strfind("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*%)` (parentheses have a special meaning in a pattern, so we must escape them with a `%`); the `%s*` matches zero or more spaces. 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.

As `*`, the modifier `-` also matches zero or more occurrences of characters of the original class. But, instead of matching the longest sequence, it matches the shortest one. Sometimes, you can choose 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(gsub(test, "/%*.*%*/", "<COMMENT>"))  --> int x; <COMMENT>
```

The pattern `.-`, instead, will expand the minimum enough to find a `*/`, and so you get your desired result:

```
test = "int x; /* x */  int y; /* y */"
print(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 numbers like `-12`, `23` and `+1009`. The `[+-]` is a character class that matches both a `+` or a `-` sign; the following `?` makes this 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 such 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 strfind(s, "^%d") then ...
```

checks whether the string `s` starts with a digit, and the test

```
if strfind(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 `%bxy`, 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()` will match parts of the string that start with a `(` and finish at the respective `)`:

```
print(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.

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 you want to capture between parentheses.

When you specify captures with the `strfind` function, the capture values are returned as extra results from the call. A typical use of this facility is to break a string in parts:

```
pair = "name = Anna"
_, _, key, value = strfind(pair, "(%a+)%s*=%s*(%a+)")
print(key, value)  --> name  Anna
```

In the above example, the pattern specifies a non empty sequence of letters (`%a+`), followed by a possibly empty sequence of spaces (`%s*`), followed by `=`, again followed by spaces and another sequence of letters. Both sequences of letters are written between parentheses, and therefore they will be captured if a match occurs. The `strfind` function always returns first the indices where the matching happened (which we store in the dummy variable `_`), and then the captures made during the pattern matching. Below is a similar example:

```
date = "17/7/1990"
_, _, d, m, y = strfind(date, "(%d+)/(%d+)/(%d+)")
print(d, m, y)  --> 17  7  1990
```

Captures can also be used in the pattern itself. In a pattern, an item like `%i`, where `i` is a single digit, matches only a copy of the `i`-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 this 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 = strfind(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 capture values is in the replacement string of `gsub`. Like the pattern, the replacement string may contain items like `%i`, which are changed by the respective captures when the substitution is made. (By the way, because of that changes, a `%` in the replacement string must be escaped as `%%`.) As an example, the following command duplicates every letter in a string, with an hyphen between the copies:

```
print(gsub("hello Lua!", "(%a)", "%1-%1"))
-->  h-he-el-ll-lo-o L-Lu-ua-a!
```

And this one interchanges adjacent characters:

```
print(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{string}`, and change them to a format in XML style, `<command>string</command>`. For such specification, the following line does the whole job:

```
s = gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
```

If `s` is the string

```
the \quote{task} is to \em{change} that.
```

that command will change it to

```
the <quote>task</quote> is to <em>change</em> that.
```

Another interesting example is how to trim a string:

```
function trim (s)
  return 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, all spaces at both extremities will be matched by the `%s*`.

The last use of capture values is perhaps the most powerful. We can call `gsub` with a function as its third argument, instead of a replacement string. When invoked this way, `gsub` calls the given function every time it makes a match; the arguments to this function are the captures, and the value the function returns is used as the replacement string. As a first example, the following function will change every occurrence of `$varname` in a string by the value of the global variable `varname`:

```
function expand (s)
  return gsub(s, "$(%w+)", getglobal)
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 gsub(s, "$(%w+)", function (v)
           return tostring(getglobal(v))
         end)
end

print(expand("print = $print; a = $a"))
--> print = function: 0x8050ce0; a = nil
```

A more powerful example uses `dostring` to evaluate whole expressions, written in the text between square brackets preceded by a dollar sign:

```
s = "sin(3) = $[sin(3)]; 2^5 = $[2^5]"

print(gsub(s, "$(%b[])", function (x)
        return dostring ("return "..strsub(x, 2, -2))
      end))
-->  sin(3) = 0.05233595624294383; 2^5 = 32
```

The first match is the string `"$[sin(3)]"`, and the capture is `"[sin(3)]"`; `strsub` removes the brackets from the captured string, so the string to be executed will be `"return sin(3)"`. Then `dostring` will return this value to the anonymous function, which will return it again as the replacement string. The same happens for the match `"$[2^5]"`.

Frequently, we use this kind of `gsub` only to iterate over a string, without any interest in the resulting string. For instance, we can collect the words of a string into a table with the following code:

```
words = {}
gsub(s, "(%a+)", function (w) tinsert(words, w) end)
```

If `s` is the string `"hello hi, again!"`, after that command the `word` table will be

```
{"hello", "hi", "again"}
```

For that kind of use, the "replacement" function does not need to return any value.

For our next example, we will use the encoding used by HTTP to send parameters in a URL. In this encoding, special characters (such as `=`, `&` and `+`) are encoded as `%XX`, where `XX` is the hexadecimal representation of the character. Then, spaces are changed to `+`. For instance, the string `"a+b = c"` is encoded as `"a%2Bb+%3D+c"`. Finally, each parameter name and parameter value are written separated by an `=`, and all pairs (name,value) are appended with an ampersand between them. 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, in a field with its corresponding name. The following function can be used to do the basic decoding:

```
function unescape (s)
  s = gsub(s, "+", " ")
  s = gsub(s, "%%(%x%x)", function (h)
        return strchar(tonumber(h, 16))
      end)
  return s
end
```

The first statement changes all `+` in the string to spaces. The second `gsub` matches all two-digit hexadecimal numerals preceded by `%`, and calls an anonymous function, which converts the hexadecimal numeral into a number (`tonumber`, with base 16), and returns the corresponding character (`strchar`).

```
  print(unescape("a%2Bb+%3D+c"))  --> a+b = c
```

The pairs (name,value) are collected by another `gsub`. Because each name or value cannot contain `&` nor `=`, these strings can be matched by the pattern `[^&=]+`:

```
cgi = {}
function decode (s)
  gsub(s, "([^&=]+)=([^&=]+)", function (name, value)
    name = unescape(name)
    value = unescape(value)
    cgi[name] = value
  end)
end
```

This `gsub` matches all pairs in the form `name=value`, and for each pair it calls the local function with the strings `name` and `value` (as marked by the parentheses in the matching string). The local function simply "unescapes" both strings and stores the pair in the `cgi` table.

The corresponding enconding 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 = gsub(s, "([&=+%c])", function (c)
        return format("%%%02X", strbyte(c))
      end)
  s = 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 t do
    s = s.."&"..escape(k).."="..escape(v)
  end
  return strsub(s, 2)     -- remove first '&'
end
```

```
  print(encode{name = "al",  query = "a+b = c", q="yes or no"})
  --> q=yes+or+no&query=a%2Bb+%3D+c&name=al
```

Tricks of the Trade

Pattern matching is a powerful tool for manipulating strings. Many complex operations can be executed with only one or a few calls to `gsub` and `strfind`. However, as with any power, you must use it carefully.

Pattern matching is not a substitute for a proper parser. For "quick-and-dirty" programs, you can do useful manipulations over 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(gsub(test, "/%*.-%*/", "<COMMENT>"))
--> char s[] = "a <COMMENT>
```

Strings with such contents are rare, and, for your own use, that pattern probably will 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 write the pattern as specific as possible; loose patterns are slower than specific ones. An extreme example is `(.-)%$`, to get all text in a string up to the first dollar sign. If the subject string has a dollar sign, everything goes fine; but suppose that the string does not contain any dollar signs. The algorithm will first try to match the pattern starting at the first position of the string. It will go through all the string, looking for a dollar. When the string ends, the pattern fails *for the first position* of the string. Then, the algorithm will do the whole thing again, starting at the second position of the string, only to discover that the pattern does not match there, too; and so on. This will take a quadratic time, which means more than three hours in a Pentium 333MHz for a text with 200K characters. You can correct this problem simply by anchoring the pattern at the first position of the string, with `^(.-)%$`. 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 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 = strfind(";$%  **#$hello13", "%a*")
print(i,j)    --> 1  0
```

In this example, the call to `strfind` 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. By the same token, 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 such pattern by hand, we can create it with the `strrep` function:

```
longlen = 70
p = strrep("[^\n]", longlen) .. "[^\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 character. We can automate that conversion with a function:

```
function nocase (s)
  s = gsub(s, "(%l)", function (c)
        return format("[%s%s]", c, strupper(c))
      end)
  return s
end

print(nocase("hi there!"))
-->  [hH][iI] [tT][hH][eE][rR][eE]!
```

Sometimes, you want to change every occurrence of s1 to s2, without special characters, repetitions, or classes. If the strings s1 and s2 are literals, it is easy to escape the magic characters. But if those strings are variable values, you can use another gsub to put the escapes for you:

```
s1 = gsub(s1, "(%W)", "%%%1")
s2 = gsub(s2, "%%", "%%%%")
```

In the search string, we escape all non-alphanumeric characters. In the replacement string, we need to 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 handle 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 gsub(s, '\\(.)', function (x)
          return format("\\%03d", strbyte(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 gsub(s, '\\(%d%d%d)', function (d)
          return "\\" .. strchar(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 = gsub(s, '(".-")', strupper)
s = decode(s)
print(s)
--> follows a typical string: "THIS IS \"GREAT\"!".
```

or, in a more compact notation,

```
print(decode(gsub(code(s), '(".-")', strupper)))
```

As a more complex task, let us return to our example of a primitive format converter, that changes format commands written as \command{string} to XML style, <command>string</command>. But now our original format is more powerful, and uses \ as a general escape, so that we can represent the characters \, { and }, writing \\, \{ and \}. To avoid our pattern matching to mix up commands and escaped characters, we should recode these sequences in the original string. However, this time we cannot code all sequences \x, because that would code also our commands (written as \command). Instead, we code \x only when x is not a letter:

```
function code (s)
  return gsub(s, '\\(%A)', function (x)
          return format("\\%03d", strbyte(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 `strchar` directly.

```
function decode (s)
  return gsub(s, '\\(%d%d%d)', strchar)
end
```

```
s = [[a \emph{command} is written as \\comand\{text\}.]]
s = code(s)
s = gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
print(decode(s))
-->  a <emph>command</emph> is written as \comand{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 value also has quotes, they 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 comma between them:

```
function toCSV (t)
  local s = ""
  for i=1,getn(t) do
    s = s..","..escapeCSV(t[i])
  end
  return strsub(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 strfind(s, '[,"]') then     -- s contain commas or quotes?
    s = format('"%s"', gsub(s, '"', '""'))
  end
  return s
end
```

To break a CSV into an array is more difficult, because we must avoid to mix 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 double quotes), and the last field is an empty string between quotes,

We could try to use myriads of `gsub` calls to handle all these 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 content 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 `"("?`: if the found quote is followed by another quote, the second quote is captured and assigned to the `c` variable, and we repeat the loop.

```
function fromCSV (s)
  s = s..','           -- ending comma
  local t = {n=0}      -- table to collect fields
  local fieldstart = 1
  repeat
    if strfind(s, '^"', fieldstart) then    -- quoted field?
      local a, c
      local i  = fieldstart
      repeat
        a, i, c = strfind(s, '"("?)', i+1)  -- find closing quote
      until c ~= '"'     -- repeat if quote is followed by quote
      if not i then error('unmatched "') end
      tinsert(t, gsub(strsub(s, fieldstart+1, i-1), '""', '"'))
      fieldstart = strfind(s, ',', i) + 1
    else                 -- unquoted; find next comma
      local nexti = strfind(s, ',', fieldstart)
      tinsert(t, strsub(s, fieldstart, nexti-1))
      fieldstart = nexti+1
    end
  until fieldstart > strlen(s)
  return t
end

t = fromCSV('"hello "" hello", "",""')
for i=1,getn(t) do print(i, t[i]) end
--> 1      hello " hello
--> 2            ""
--> 3
```

18.   The System Library

The system library is the interface to the operating system. It includes functions for file manipulation, for getting current date and time, and other related facilities. This library pays a little 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 standard, and therefore are not provided by the system library. There are other Lua libraries, not included in the main distribution, that provide extended OS access. An example is the `poslib` library, which offers all functionality of the POSIX standard to Lua.

18.1   Input & Output

The `readfrom` function opens a file for input. It receives the file name, and returns a *file handle*, which is a userdata that Lua uses to represent the file. Moreover, `readfrom` stores this new file handle in the global variable `_INPUT`, so that this file becomes the *current input file*, which will be used as the default file for subsequent read operations.

Similarly, the `writeto` function opens a file for output. It stores its result in the variable `_OUTPUT`, as the *current output file*. Remember that, when you open a file for writing, you erase all its previous content. To append to a file, you should open it with the `appendto` function.

The system library initializes the global variables `_STDIN`, `_STDOUT`, and `_STDERR` with handles to the standard input, standard output, and standard error, respectively. It also initializes the variables `_INPUT` and `_OUTPUT` with the values of `_STDIN` and `_STDOUT`, respectively.

To close the current input file, you can call `readfrom` without arguments. This call will also reset `_INPUT` to the standard input. Similarly, to close the current output file, simply call `writeto` without arguments (even when you opened the file with `appendto`). This call will also reset `_OUTPUT` to the standard output.

All these functions return **nil** plus an error message and an error number in case of errors:

```
print(readfrom("non existent file"))
--> nil     No such file or directory       2

print(writeto("/etc/passwd"))
--> nil    Permission denied       13
```

The interpretation of the error numbers is system dependent.

As the `write` function is simpler than `read`, we will see it first. The `write` function simply gets an arbitrary number of string arguments, and write them to the current output file. If you want to write to a different file, you can supply the file handle as the first argument to `write`. Numbers are converted to strings following the usual conversion rules; for full control over this conversion, you should use the `format` function:

```
> write("sin (3) = ", sin(3), "\n")
--> sin (3) = 0.05233595624294383
> write(format("sin (3) = %.4f\n", sin(3)))
--> sin (3) = 0.0523
> write(_STDERR, "error message!\n")   -- string goes to stderr
```

Avoid code like `write(a..b..c)`; the call `write(a,b,c)` accomplishes the same effect with less resources, as it avoids the concatenation.

As a general 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

> write("hello", "Lua"); write("Hi", "\n")
--> helloLuaHi
```

Unlike `print`, `write` adds no extra characters to the output, such as tabs or newlines. Moreover, `write` writes to the current output file, while `print` always writes to the standard output. Finally, `print` automatically applies the `tostring` function over its arguments, so it can also show tables, functions, and **nil**.

The following code shows how we can implement `print` using `write`:

```
function print (...)
  for i = 1, arg.n do
    write(_STDOUT, tostring(arg[i]), "\t")
  end
  write(_STDOUT, "\n")
end
```

On the other hand, we cannot define `write` using `print`.

The `read` function reads strings from a file. Like `write`, `read` accepts a file handle as its optional first argument, which tells from which file to read. Without this argument, it reads from the current input file (`_INPUT`). The other arguments control what `read` reads:

`"*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 `read("*all")` (or simply `read("*a")`) reads the whole file, starting at the 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 very simple technique for writing filters in Lua is to read the whole file into a string, do the processing over the string (typically with `gsub`), and then write the string to the output:

```
t = read("*a")         -- read the whole file
t = gsub(t, ...)       -- do the job
write(t)               -- write the file
```

As an example, the following code is a complete program to code a file content using the *quoted-printable* encoding of mime. In this encoding, non-ASCII characters are coded as =*XX*, where *XX* is the ASCII 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 = read("*a")
t = gsub(t, "([\128-\255=])", function (c)
      return format("=%02X", ascii(c))
    end)
write(t)
```

or simply

```
write(gsub(read("*a"), "([\128-\255=])", function (c)
        return format("=%02X", ascii(c))
      end))
```

On a Pentium 333MHz, this program takes 0.2 seconds to convert a file with 200K characters.

The call `read("*line")` (or `read("*l")`) 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 `read()` has the same effect as `read("*l")`. 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. 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 1 do
  local line = read()
```

```
  if line == nil then break end
  write(format("%6d  %s\n", count, line))
  count = count+1
end
```

A complete program to sort the lines of a file can be written as

```
-- read the lines in table 'lines'
local lines = {}
while 1 do
  local line = read()
  if line == nil then break end
  tinsert(lines, line)
end
-- do the sort
sort(lines)
-- write all the lines
for i=1,getn(lines) do write(lines[i], "\n") end
```

This program sorts a file with 4.5 Mbytes (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 `read("*number")` (or `read("*n")`) reads a number from the current input file. This is the only case where `read` returns a number, and not a string. When you need to read many numbers from a file (say, tens of thousands), 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), the call 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, such as

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

and you want to print the maximum of each line. You can read all three numbers in a single call to `read`:

```
while 1 do
  local n1, n2, n3 = read("*n", "*n", "*n")
  if not n1 then break end
  print(max(n1, n2, n3))
end
```

In any case, you should always consider the alternative of reading the whole file with `read("*a")` and then using `gsub` to break it:

```
gsub(read("*a"), "(%S+)%s+(%S+)%s+(%S+)%s+", function (n1, n2, n3)
  print(max(n1, n2, n3))
end)
```

Besides the basic read patterns, you can call **read** with a number n as argument: in such 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. For instance, the following program is a very efficient way (in Lua, of course) to copy a file:

```
local size = 2^14
while 1 do
  local block = read(size)
  if not block then break end
  write(block)
end
```

This read format is also useful for binary files, as we will see now.

Binary Files

The **readfrom**, **writeto**, and **appendto** functions always open a file in text mode (the default). In a Unix platform, there is no difference between binary files and text files. But in some systems, noticeable DOS, Windows and Mac, binary files must be open with a special flag. You can open such files with the **openfile** function, which mimics the **fopen** function in C. It gets the name of the file to open, plus an explicit *mode* string. This string can contain an **r** for reading, a **w** for writing, or an **a** for appending, plus an optional **b** for binary files. The **openfile** function returns a handle for the file, but it does not update neither **_INPUT** nor **_OUTPUT**. There is also a **closefile** function, that gets a file handle and closes the respective file, again without updating neither **_INPUT** nor **_OUTPUT**. Like their cousins, **openfile** and **closefile** return **nil** plus an error message in case of errors.

As an interesting exercise, we can write the **readfrom**, **writeto**, and **appendto** functions using **openfile** and **closefile**:

```
function readfrom (...)
  if arg.n == 0 then      -- no arguments?
    local f = _INPUT
    _INPUT = _STDIN
    return closefile(f)
  else
    local f, msg = openfile(arg[1], "r")
    if f then _INPUT = f end
    return f, msg
  end
```

```
function writeto (...)
  if arg.n == 0 then      -- no arguments?
    local f = _OUTPUT
    _OUTPUT = _STDOUT
    return closefile(f)
  else
    local f, msg = openfile(arg[1], "w")
    if f then _OUTPUT = f end
    return f, msg
  end

function appendto (filename)
  local f, msg = openfile(filename, "a")
  if f then _OUTPUT = f end
  return f, msg
end
```

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 (the only exception is the `format` function). You can even do pattern matching over binary data, as long as the pattern does not contain a byte zero. If you want to match the byte zero, you can use the class `%z` instead.

Typically, you read binary data either with the `*a` 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 assumes that the variables `inp` and `out` have the names of the input file and the output file, respectively.

```
_INPUT = openfile(inp, "rb")
_OUTPUT = openfile(out, "wb")
data = read("*a")                    -- read the whole file
data = gsub(data, "\r\n", "\n")
write(data)
```

You can call this program with the following command line:

```
> lua inp=file.dos out=file.unix prog.lua
```

As another example, the following program prints all "strings" found in a binary file. The program assumes that a string is a 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 the `strrep` function 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 = openfile(inp, "rb")
data = read(f, "*a")
closefile(f)
validchars = "[%w%p%s]"
pattern = "(" .. strrep(validchars, 6) .. "+)%z"
gsub(data, pattern, print)
```

The following program makes a dump of a binary file. Again, the variable `inp` has the input file name; the output goes to the standard output. The program reads the file in chunks of 16 bytes; for each chunk, it writes the hexadecimal representation of each byte, then it writes the chunk as text, changing control characters to dots.

```
_INPUT = openfile(inp, "rb")
while 1 do
  local bytes = read(16)
  if bytes == nil then break end
  gsub(bytes, "(.)", function (b)
    write(format("%02X ", strbyte(b)))
  end)
  write(strrep("   ", 16-strlen(bytes)+1))    -- align strings
  write(gsub(bytes, "%c", "."), "\n")
end
```

Suppose that this program is stored in a file named `vip`; if we apply the program over itself, with the call

```
C:\home>lua inp=vip vip
```

it will produce the following output (in a Unix machine):

```
5F 49 4E 50 55 54 20 3D 20 6F 70 65 6E 66 69 6C    _INPUT = openfil
65 28 69 6E 70 2C 20 22 72 62 22 29 20 0A 77 68    e(inp, "rb") .wh
69 6C 65 20 31 20 64 6F 0A 20 20 6C 6F 63 61 6C    ile 1 do.  local
20 62 79 74 65 73 20 3D 20 72 65 61 64 28 31 36     bytes = read(16
29 0A 20 20 69 66 20 62 79 74 65 73 20 3D 3D 20    ).  if bytes ==
6E 69 6C 20 74 68 65 6E 20 62 72 65 61 6B 20 65    nil then break e
6E 64 0A 20 20 67 73 75 62 28 62 79 74 65 73 2C    nd.  gsub(bytes,
20 22 28 2E 29 22 2C 20 66 75 6E 63 74 69 6F 6E     "(.)", function
20 28 62 29 0A 20 20 20 20 77 72 69 74 65 28 66     (b).    write(f
6F 72 6D 61 74 28 22 25 30 32 58 20 22 2C 20 73    ormat("%02X ", s
74 72 62 79 74 65 28 62 29 29 29 0A 20 20 65 6E    trbyte(b))).  en
64 29 0A 20 20 77 72 69 74 65 28 73 74 72 72 65    d).  write(strre
70 28 22 20 20 20 22 2C 20 31 36 2D 73 74 72 6C    p("   ", 16-strl
65 6E 28 62 79 74 65 73 29 2B 31 29 29 20 20 20    en(bytes)+1))
20 2D 2D 20 61 6C 69 67 6E 20 73 74 72 69 6E 67     -- align string
73 0A 20 20 77 72 69 74 65 28 67 73 75 62 28 62    s.  write(gsub(b
79 74 65 73 2C 20 22 25 63 22 2C 20 22 2E 22 29    ytes, "%c", ".")
2C 20 22 5C 6E 22 29 20 0A 65 6E 64 0A 0A          , "\n") .end..
```

18.2   Other Operations on Files

The `rename` function changes the name of a file; `remove` removes (deletes) a file. The `tmpname` function returns a name that you can use for a temporary file; it is your responsibility to open, close, and remove such files. The `flush` function executes all pending writes to a file. It can be called with a file handle, to flush that specific file, or without arguments, to flush all open files.

The `seek` function can be used both to get and to set the current position of a file. Its general form is `seek(filehandle, whence, offset)`. The `whence` argument 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 0. Therefore, the call `seek(file)` returns the current file position, without changing it; the call `seek(file, "set")` resets the position to the beginning of the file (and returns 0); and the call `seek(file, "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 = seek(file)      -- get current position
  local size = seek(file, "end")  -- get file size
  seek(file, "set", current)      -- restore position
  return size
end
```

All the previous functions return **nil** plus an error message in case of errors.

18.3   Date and Time

All date and time queries in Lua are done through a single function, called `date`. This function gets as argument a format string, and returns a copy of this string where specific tags have been replaced by information about time and date. All tags are represented by a `%` followed by a letter; for instance

```
print(date("today is %A; minutes = %M"))
--> today is Tuesday; minutes = 53
```

The following table shows each tag, its meaning, and an example of its value for September 16, 1998 (a Wednesday), at 23:48:10. For numeric values, the table also shows the range of possible values.

`%Y`	full year (`1998`)
`%y`	year, with two digits (`98`) [00–99]
`%m`	month (`09`) [01–12]
`%B`	full month name (`September`)
`%b`	abbreviated month name (`Sep`)
`%d`	day of the month (`16`) [01–31]
`%w`	weekday (`3`) [0–6 = Sunday–Saturday]
`%A`	full weekday name (`Wednesday`)
`%a`	abbreviated weekday name (`Wed`)
`%H`	hour, using a 24-hour clock (`23`) [00–23]
`%I`	hour, using a 12-hour clock (`11`) [01–12]
`%p`	either `"am"` or `"pm"` (`pm`)
`%M`	minute (`48`) [00–59]
`%S`	second (`10`) [00–61]
`%c`	date and time (`09/16/98 23:48:10`)
`%x`	date (`09/16/98`)
`%X`	time (`23:48:10`)
`%%`	the character `%`

All representations follow the current locale. Therefore, in a locale for Brazil–Portuguese, `%B` would result in `"setembro"`, and `%x` in `"16/09/98"`. If you call `date` without 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 system. For instance, one system may show `date"%x"` as `"09/16/98"`, and other as `"Sep.  16, 1998"`. If you want a fixed representation like `mm/dd/yyyy`, use a format string like `"%m/%d/%Y"`.

You should always use a single call to the `date` function to get all information you need. Multiple calls can lead to synchronization errors. For instance, at 10:59:59, a call `date"%H"` returns 10, and then a call `date"%M"`, a moment later, may return 00. Your final result will be 10:00, one full hour wrong. The call `date"%H:%M"` ensures that all information is got at the same time, either 10:59 or 11:00. you can break the result with `strfind`:

```
_, _, h, m = strfind(date("%H:%M"), "(%d+):(%d+)")
```

The `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 = clock()
local i = 1
while i<=100000 do i=i+1 end
print(format("elapsed time: %.2f\n", clock()-x))
```

18.4 Other system calls

The `exit` function terminates the execution of a program. The `getenv` function gets the value of an environment variable. It receives the name of the variable and returns a string with its value.

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

If the variable is not defined, the call returns **nil**. The function `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, in DOS-Windows, you can write the following function to create new directories:

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

The `execute` function is very powerful, but it is also highly system dependent.

The `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 gets two string arguments: The locale name and a category, which specifies what features will be influenced by this locale. 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 formated; `"time"` controls how date and time are formated (i.e., function `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(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 numbers are written inside Lua code:

```
-- set locale for Portuguese-Brazil
print(setlocale('pt_BR'))        --> pt_BR
print(3.4)                       --> 3,4
dostring("print(3,4+1)")         --> 3    5
```

Part IV

Tag Methods

to do: This whole part needs lots of work. We have to explain what are tag methods, how we use them, and provide several examples with typical uses, such as different forms of inheritance, proxies, operator overloading, tracking and declaring global variables, autoload, etc.

Part V

The C API

to do: I have written only the first chapters of this part. I still have to cover the auxiliar lib, userdata, tags, gargabe-collector management, upvalues (in C), etc.

What makes Lua different from most other languages is that 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. This 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 other language), and can add facilities that cannot be written directly in Lua. This is what makes Lua an *extensible* language.

The C API is the set of functions that allows a *host* program to interact with Lua. It comprises functions to read and write Lua global variables, to call Lua functions, to run a piece of Lua code, to register a C function so that it can later be called by Lua code, and so on.

The C API follows the C *modus operandi*, which is quite different from Lua's. Most functions 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 easy of use. Later, we will study the *auxiliar library* (`lauxlib`), which uses the basic API to provide a higher abstraction level.

In the following chapters, we are using a "hat" of C programmers. So, when we talk about "you", we mean you when programming in C, or you impersonated by the C code you write.

19.   A First Example

As a first example, the stand-alone Lua interpreter is a simple C program, that takes your files and strings and feed them to the Lua library. We can write a very primitive stand-alone interpreter as follows:

```
#include <stdio.h>
#include <lua.h>

int main (void) {
  char line[BUFSIZ];
  lua_State *L = lua_open(0);          /* opens Lua */
  while (fgets(line, sizeof(line), stdin) != 0)
     lua_dostring(L, line);            /* executes the string */
  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 execute chunks (such as `lua_dostring`), to read and write global variables in the Lua environment, to call Lua functions, to register new functions to be called by Lua, and so on.

The `lua_open` function creates a new environment (or *state*). When `lua_open` creates a fresh environment, this environment contains no pre-defined functions. To keep Lua small, all standard libraries are provided as a separate package, so that you only use them if you need. If you want to include the standard libraries in your little interpreter, your code will look like

```
#include <stdio.h>
#include <lua.h>
#include <lualib.h>

int main (void) {
  char line[BUFSIZ];
  lua_State *L = lua_open(0);
```

```
  lua_baselibopen(L);            /* opens the basic library */
  lua_iolibopen(L);              /* opens the I/O library */
  lua_strlibopen(L);             /* opens the string lib. */
  lua_mathlibopen(L);            /* opens the math lib. */
  while (fgets(line, sizeof(line), stdin) != 0)
    lua_dostring(L, line);
  lua_close(L);
  return 0;
}
```

The header file `lualib.h` defines functions to open the libraries. Each of these functions, when called, registers in the Lua environment the functions of the respective library. So, the call to `lua_iolibopen` will register the functions `read`, `write`, `readfrom`, etc.

The Lua library uses 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 among all functions inside Lua. This implementation makes Lua reentrant, and ready to be used in multi-threaded code. However, many common Lua applications use only one single state, and so it is customary for them to store this state in a global variable. So, unless otherwise stated, we will assume that our program has a global declaration like

```
lua_State *L;
```

and we will use this state `L` in many of our examples.

20. 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 garbage collection.

In Lua, when we write `a[k] = v`, both `k` and `v` can have many different types (even `a` may have different types, because of tag methods). But a single Lua function, such as `settable(a, k, v)`, is enough to represent all possibilities. If we want to offer this funcionallity in C, however, any `settable` function must have fixed types. We would need dozens different functions for this single functionallity (one new function for each combination of types for the three arguments).

We could solve this problem declaring some kind of union type in C, let us call it `lua_Value`, which 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 complex type to other languages (Lua has been designed to interface easily not only with C/C++, but also with Java, Fortran, etc.). 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 ask a value from Lua (such as `lua_getglobal`), you call Lua, which pushes this value in the stack. Whenever you want to pass a value to Lua (such as `lua_setglobal`), you first push this value in the stack, and then you call Lua (which will pop the value). We still need a different function to push each C type into the stack (`lua_pushnumber`, `lua_pushstring`, etc.), and a different function to get each value from the stack (`lua_tonumber`, `lua_tostring`, etc.), but we avoid the combinatorial explosion. Moreover, because this stack is managed by Lua, the garbage collector knows exactly which values are being used by C.

Lua manipulates this stack in a strict FIFO discipline (First In, First Out; that is, always through the top), but your C code can inspect any element inside

it. To refer to an element in the stack, the API uses an `index`. 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. But we can also access elements using the top of the stack as our reference, using negative indices. Then, −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_type(L, -1)` returns the type of the value at the top of the stack. As we will see, there are several occasions where it is quite natural to index the stack from the bottom (that is, with positive indices), and several other occasions where the natural way is to use negative indices.

to do: explain valid and acceptable indices

21. Extending your Application

A main use of Lua is as a *configuration* language. In this section, we will illustrate how Lua is used as a configuration language, starting with a simple example and evolving it to more complex tasks.

As our first task, let us imagine a very simple configuration scenario: Your program (let us call it pp) has a window, and you want the user to be able to give the initial window size. Clearly, for such a simple task, there are many options simpler than using Lua, such as environment variables or files with name-value pairs. But even for 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, such 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 <lualib.h>

int width, height;  /* global variables */
lua_State *L;

void load (char *filename) {
  L = lua_open(0);
  lua_baselibopen(L);
  lua_iolibopen(L);
  lua_strlibopen(L);
  lua_mathlibopen(L);
  if (lua_dofile(L, filename) != 0)
    error("cannot run configuration file");
```

145

```
  else {
    lua_getglobal(L, "width");
    lua_getglobal(L, "height");
    if (!lua_isnumber(L, 1) || !lua_isnumber(L, 2))
      error("invalid values");
    else {
      width = (int)lua_tonumber(L, 1);
      height = (int)lua_tonumber(L, 2);
    }
  }
  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 `lua_dofile` to run the chunk in file `filename`. If there is any error (e.g., a syntax error), Lua will give an error message, and `lua_dofile` will return an error code; otherwise `lua_dofile` returns 0.

After running the chunk, the program needs to get the values of the global variables. For that, it first calls the `lua_getglobal` function, whose single argument (besides the omnipresent `lua_State`) is the global name. Each call pushes the corresponding global value into the API stack. Because the stack was previously empty, the first value will be at index 1, and the second at index 2. Conversely, you could index from the top, using -2 from the first value and -1 from the second.

To check the type of a stack element, the API provides the functions `lua_isnil`, `lua_isnumber`, `lua_isstring`, `lua_isuserdata`, `lua_isfunction`, and `lua_istable`; they receive the index of the element to check. Our example uses `lua_isnumber` to check whether both values are numeric. Finally, it uses `lua_tonumber` to convert such values to `double`, and C does the coercion to `int`.

Is it worth the use of Lua? As we said before, for such a simple task, a simple file with only two numbers in it would be much easier to use than Lua. But even in this scenario, the use of Lua has some advantages. First, your configuration file can have comments. Second, the user already can 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 a trivial configuration scenario, 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 to use Lua is that now it is easy to add new configuration facilities to your program, and this easiness creates an attitude that results in more flexible programs.

22. ??

Let us adopt this attitude: Now, you want to configure a background color for the window, too. We will assume that the final color specification is composed by three numbers, where each number is a color component in RGB. A naive approach is to ask the user to give each component in a different global variable:

```
-- configuration file for program 'pp'
width = 200
height = 300
background_red = 30
background_green = 100
background_blue = 0
```

This approach has several drawbacks; it is too verbose (most real programs need dozens of different colors, for window background, window foreground, menu background, etc.); and there is no way to pre-define some colors, so that, later, you can simply write something like `background = WHITE`. A better option is to use a table to represent a color:

```
background = {r=30, g=100, b=0}
```

The use of tables gives more structure to the script; now it is easy for the user or the application to pre-define colors for later use in her configuration file:

```
BLUE = {r=0, g=0, b=255}
...
background = BLUE
```

To get these values from your C program, you can do as follows:

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
  error("'background' is not a valid color table");
else {
  red = getfield("r");
  green = getfield("g");
  blue = getfield("b");
}
```

149

As usual, you first get the value of the global variable `background`, and then you make sure 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:

```
int getfield (const char *key) {
  int result;
  lua_pushstring(L, key);
  lua_gettable(L, -2);  /* get background[key] */
  if (!lua_isnumber(L, -1))
    error("invalid component in background color");
  result = (int)lua_tonumber(L, -1);
  lua_pop(L, 1);
  return result;
}
```

Again, the problem is polymorphism: There are potentially many versions of `getfield` functions, varying the key type, value type, error handling, etc. The API offers a single function, `lua_gettable`. It receives the index of the table, pops the key from the stack, and pushes the corresponding value. Our private `getfield` assumes that the table is at 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 in the same state it was before the call. (The argument 1 in the call to `lua_pop` is the number of elements to pop.)

We will extend our example a little further, and introduce color names for the user. The user still can use color tables, but she can also use names for the more common colors, such as `background = WHITE`. To implement this feature, you will need a color table in your C application:

```
struct ColorTable {
  char *colorname;
  unsigned char red, green, blue;
} colortable[] = {
  {"WHITE",   255, 255, 255},
  {"RED",     255,   0,   0},
  {"GREEN",     0, 255,   0},
  {"BLUE",      0,   0, 255},
  ...
};

#define NUMCOLORS (sizeof(colortable)/sizeof(colortable[0]))
```

Our first implementation will create global variables with the color names, and initialize these variables using color tables, in the same way a user would do:

```
WHITE = {r=255, g=255, b=255}
RED   = {r=255, g=0,   b=0  }
...
```

The only difference from user-defined colors is that these colors would be defined by the application, before it runs the user script.

The `set_color` function defines a single color. It must create a table, set the appropriate fields, and assign this table to the corresponding global variable:

```
void set_color (struct ColorTable *ct) {
  lua_newtable(L);
  set_field(table, "r", ct->r);          /* table.r = ct->r */
  set_field(table, "g", ct->r);          /* table.g = ct->g */
  set_field(table, "b", ct->r);          /* table.b = ct->b */
  lua_setglobal(ct->colorname);          /* 'colorname' = table */
}
```

The `lua_newtable` function creates an empty table, and pushes it on the stack. The `set_field` calls set the table fields, and finally `lua_setglobal` pops the table and sets it as the value of the global with the given name.

To set the table fields, we define an auxiliary function, `set_field`; it pushes on the stack the index and the field value, and then call the `lua_settable` function.

```
void set_field (const char *index, int value) {
  lua_pushstring(L, index);
  lua_pushnumber(L, value);
  lua_settable(L, -3);
}
```

Like other API functions, `lua_settable` works with many different types, so it gets all its arguments from the stack. `lua_settable` receives the table index as an argument, and pops the key and the value. Before the call the table is at the top of the stack (index -1); after pushing the index and the value, it it at index -3.

With those previous functions, the following loop will register all colors in the application's table:

```
int i;
for (i=0; i<NUMCOLORS; i++)
  set_color(&colortable[i]);
```

Remember that the application must execute this loop before it runs the user script with `lua_dofile`.

Let us see another option for implementing named colors. Instead of global variables, the user can denote color names with strings, writing her settings as `background = "BLUE"`. With this implementation, the application does not need to do anything before running the user's script. However, it needs more work to get a color. When it gets the value of the `background` variable, it has to check 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 *colorname = lua_tostring(L, -1);
  int i;
  for (i=0; i<NUMCOLORS; i++)  /* look up the string */
    if (strcmp(colorname, colortable[i].colorname) == 0)
      break;
  if (i == NUMCOLORS)  /* string not found? */
    error("invalid color name");
  else ... /* use colortable[i] */
} else if (!lua_istable(L, -1))
  /* ... (as before) */
```

The new functions here are `lua_isstring` and `lua_tostring`, which are similar to `lua_isnumber` and `lua_tonumber`, respectively.

Which 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 will not signal any error if the user misspells a color name. If the user writes `background = WITE`, 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` after the statement `background = "WITE"` is the misspelled string, and the application can use this knowledge in the error message; for instance

```
lua error: invalid color name 'WITE'
```

The application can also compare strings regardless 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) for the user to choose only a few.

23.   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
graphic 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 the
`lua_call` to do the actual call; and 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 * sin(y))/(1-x)
end
```

and you want to evaluate, in C, `z = f(x, y)` for given `x` and `y`. You can
encapsulate this call in the following C function, assuming that you have already
opened the Lua library and ran the configuration file:

```
/* call a function 'f' defined in the configuration file */
double f (double x, double y) {
  double z;
  lua_getglobal(L, "f");
  if (!lua_isfunction(L, -1))
    error("function 'f' not defined in configuration file");
  lua_pushnumber(L, x);        /* push 1st argument */
  lua_pushnumber(L, y);        /* push 2nd argument */
  lua_call(L, 2, 1);           /* do the call (2 arguments, 1 result) */
  if (!lua_isnumber(L, -1))
    error("function 'f' must return a number");
  z = lua_tonumber(L, -1);
  lua_pop(L, 1);  /* remove returned value from the stack */
  return z;
}
```

You call `lua_call` with the number of arguments you are passing, and the
number of results that you want. As in Lua, `lua_call` adjusts the actual number

of results to what you asked for, pushing **nil**s or discarding extra values, as needed.

   If there is an error while `lua_call` is running a function, Lua calls the error handler, and then `lua_call` returns a value different from zero, to signal the error; moreover, it pushes no results. In our previous example, we do not need to check the result of `lua_call` because, in case of errors, the result could not be a number, and our code would signal a `"function 'f' must return a number"` error.

24. Registering C Functions

One of the basic means for extending Lua is for the application to *register* new functions, written in C, to Lua. As we saw previously, when C calls a Lua function, it must follow a protocol to pass the arguments and to get the results. Similarly, when Lua calls a C function, this C function must follow a protocol to get its arguments and to return its results.

When Lua calls a C function, it uses the same stack that C uses to call Lua. The C function gets its arguments from the stack, and pushes the results into the stack. As a first example, let us see how to implement a simplified version of a function that returns the sin of a given number (a more professional implementation should check whether its argument is actually a number):

```
int l_sin (lua_State ls) {
  double d = lua_tonumber(ls, 1);
  lua_pushnumber(ls, sin(d));
  return 1;  /* number of results */
}
```

Any function registered into Lua must have this prototype: It gets as its single argument the Lua state, and returns (in C) the number of values it wants to return (in Lua). Therefore, it does not need to clear the stack before pushing its results. Lua removes whatever is in the stack below the resulting values.

Before we can use this function from Lua, we must *register* it. For that, we have to run the following code:

```
lua_pushcfunction(L, l_sin);
lua_setglobal(L, "sin");
```

The first line passes this function to Lua, and the second assigns it to the global variable `sin`. This sequence is so common that `lua.h` has a macro for it:

```
lua_register(L, "sin", l_sin);
```

to do: everything from this point on are just ideas which must be put together.