Lua Programming

en.wikibooks.org

July 31, 2016

On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. A URI to this license is given in the list of figures on page 67. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 65. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 71, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 67. This PDF was generated by the LaTeX typesetting software. The LaTeX source code is included as an attachment (`source.7z.txt`) in this PDF file. To extract the source from the PDF file, you can use the `pdfdetach` tool including in the `poppler` suite, or the `http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/` utility. Some PDF viewers may also let you save the attachment to a file. After extracting it from the PDF file you have to rename it to `source.7z`. To uncompress the resulting archive we recommend the use of `http://www.7-zip.org/`. The LaTeX source itself was generated by a program written by Dirk Hünniger, which is freely available under an open source license from `http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf`.

Contents

1 Introduction **3**
1.1 Hello, world! . . . 4
1.2 Comments . . . 4
1.3 Syntax . . . 5
1.4 Obtaining Lua . . . 6

2 Expressions **7**
2.1 Types . . . 7
2.2 Literals . . . 11
2.3 Coercion . . . 12
2.4 Bitwise operations . . . 12
2.5 Operator precedence . . . 14
2.6 Quiz . . . 15

3 Statements **17**
3.1 Assignment . . . 17
3.2 Conditional statement . . . 20
3.3 Loops . . . 21
3.4 Blocks . . . 23

4 Functions **27**
4.1 Returning values . . . 29
4.2 Errors . . . 29
4.3 Stack overflow . . . 31
4.4 Variadic functions . . . 32

5 Standard libraries **33**
5.1 Basic library . . . 33
5.2 Coroutines . . . 35
5.3 String matching . . . 36

6 Appendix:Software testing **41**
6.1 Type checking . . . 41
6.2 White-box testing . . . 42
6.3 Further reading . . . 42

7 Glossary **45**

8 Index **53**
8.1 A . . . 53

8.2 B . . . 53
8.3 C . . . 54
8.4 D . . . 55
8.5 E . . . 55
8.6 F . . . 55
8.7 G . . . 56
8.8 H . . . 56
8.9 I . . . 56
8.10 J . . . 57
8.11 K . . . 57
8.12 L . . . 57
8.13 M . . . 57
8.14 N . . . 58
8.15 O . . . 58
8.16 P . . . 58
8.17 Q . . . 59
8.18 R . . . 59
8.19 S . . . 59
8.20 T . . . 60
8.21 U . . . 60
8.22 V . . . 61
8.23 W . . . 61
8.24 X . . . 61
8.25 Y . . . 61
8.26 Z . . . 61
8.27 Lua API . . . 61

9 Contributors **65**

List of Figures **67**

10 Licenses **71**
10.1 GNU GENERAL PUBLIC LICENSE . . . 71
10.2 GNU Free Documentation License . . . 72
10.3 GNU Lesser General Public License . . . 73

1 Introduction

Lua (not ”LUA”, which is incorrect although common) is a powerful, fast, lightweight and embeddable programming language. It is used by many frameworks, games and other applications. While it can be used by itself, it has been designed to be easy to embed in another application. It is implemented in ANSI C, a subset of the C programming language that is very portable, which means it can run on many systems and many devices where most other scripting languages would not be able to run. The purpose of this book is to teach Lua programming to anyone regardless of previous programming experience. The book can be used as an introduction to programming, for someone who has never programmed before, or as an introduction to Lua, for people who have programmed before but not in Lua. Since there are many development platforms and games that use Lua, this book can also be used to learn to use Lua and then to use it in that development platform.

This book aims to teach usage of the latest version of Lua. This means it will be attempted to regularly update it as new versions of Lua come out (Lua releases are infrequent enough that this should not be too difficult). Currently, the book is up-to-date for Lua 5.2, which is the previous version. If you are using Lua in an embedded environment that uses an older version of Lua in the 5.x branch (Lua 5.0 and Lua 5.1), the material should still be sufficiently relevant for you.

Lua was designed and is being maintained at the Pontifical Catholic University of Rio de Janeiro, which is located in Brazil. Its creators are Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo.

> ”Lua” (pronounced LOO-ah) means ”Moon” in Portuguese. As such, it is neither an acronym nor an abbreviation, but a noun. More specifically, ”Lua” is a name, the name of the Earth’s moon and the name of the language. Like most names, it should be written in lower case with an initial capital, that is, ”Lua”. Please do not write it as ”LUA”, which is both ugly and confusing, because then it becomes an acronym with different meanings for different people. So, please, write ”Lua” right!

Lua comes from two languages that were designed by TeCGraf (a laboratory at the Pontifical Catholic University of Rio de Janeiro): DEL and Sol. DEL means ”data entry language”, while Sol means ”simple object language” and also means sun in Portuguese, which is why the name Lua was chosen, since it means ”moon” in Portuguese. It was created for Petrobras, a Brazilian oil company, but was also used in many other projects in TeCGraf, and is now used in a multitude of projects world-wide. Lua is one of the leading languages in the field of embedded game development.

One of the main advantages of Lua is its simplicity. Some companies use it exclusively because of that advantage: they think their employees would be able to work better if they could use a programming language to perform certain tasks, but they cannot afford to

give to their employees a full course on a complicated programming language. Some very simple languages like Bash or Batch here would not be powerful enough to perform these tasks, but Lua is both powerful and simple. Another of the important advantages of Lua is its capability to be embedded, which was one of the most important characteristics of it throughout all of its development. Games like or World of Warcraft or ROBLOX need to be able to embed Lua in their application so users of the application can use it.

Programming, which is also sometimes called scripting in the case of programs that run inside embedded applications, is the process of writing computer programs. A programming language is a language used to give instructions to a computer through computer code that is contained in a computer program. A programming language consists of two things: a syntax, which is like grammar in English, and libraries, basic functions provided with the language. These libraries could be compared with vocabulary in English.

1.1 Hello, world!

Lua can either be used embedded in an application or by itself. This book will not describe the process to install Lua on your computer, but you can execute code using codepad[1] or the Lua demo[2]. The first example of Lua code in this book will be the basic and traditional hello world program.

> A **"Hello world" program** is a computer program that outputs "Hello, world" on a display device. Because it is typically one of the simplest programs possible in most programming languages, it is by tradition often used to illustrate to beginners the most basic syntax of a programming language, or to verify that a language or system is operating correctly.

```
print("Hello, world!")
```

The code above prints the text `Hello, world!` to the output, printing referring to displaying text in the output, not to printing something on paper. It does so by calling the `print` function with the string "Hello, world!" as an argument. This will be explained in the chapter about functions.

Note that Lua is most of the time embedded in a lower level application, which means that the `print` function will not always display text in an area that is visible to the user. The documentation of the programming interface of these applications will generally explain how text may be displayed to users.

1.2 Comments

A comment is a code annotation that is ignored by the programming language. Comments can be used to describe one or many lines of code, to document a program, to temporarily

1 `http://codepad.org`
2 `http://www.lua.org/demo.html`

disable code, or for any other reason. They need to be prefixed by two hyphens to be recognized by Lua and they can be put either on their own line or at the end of another line:

```
print("This is normal code.")
-- This is a comment
print("This is still normal code.") -- This is a comment at the end of a line of
 code.
```

These comments are called short comments. It is also possible to create long comments, which start with a long bracket and can continue on many lines:

```
print("This is normal code")
--[[Line 1
Line 2
]]
```

Long brackets consist of two brackets in the middle of which any number of equality signs may be put. That number is called the level of the long bracket. Long brackets will continue until the next bracket of the same level, if there is one. A long bracket with no equal sign is called a long bracket of level 0. This approach makes it possible to use closing double brackets inside of long comments by adding equal signs in the middle of the two brackets. It is often useful to do this when using comments to disable blocks of code.

```
--[==[
This is a comment that contains a closing long bracket of level 0 which is here:
 ]]
However, the closing double bracket doesn't make the comment end, because the
 comment was opened with an opening long bracket of level 2, and only a closing
 long bracket of level 2 can close it.
]==]
```

In the example above, the closing long bracket of level 0 (`]]`) does not close the comment, but the closing long bracket of level 2 (`]==]`) does.

1.3 Syntax

The syntax of a programming language defines how statements and expressions must be written in that programming language, just like grammar defines how sentences and words must be written. Statements and expressions can be respectively compared to sentences and words. Expressions are pieces of code that have a value and that can be evaluated, while statements are pieces of code that can be executed and that contain an instruction and one or many expressions to use that instruction with. For example, `3 + 5` is an expression and `variable = 3 + 5` is a statement that sets the value of *variable* to that expression.

The entire syntax of Lua can be found in extended Backus–Naur form on the Lua website[3], but you wouldn't understand anything if you read it. Extended Backus–Naur Form[4] is a metalanguage, a language used to describe another language, just like a metawebsite is a

3 http://www.lua.org/manual/5.1/manual.html#8
4 https://en.wikipedia.org/wiki/Extended%20Backus%E2%80%93Naur%20Form

website about a website, and just like metatables, in Lua, are tables that define the behavior of other tables (you'll learn about metatables and tables later in this book). But you're not going to have to learn extended Backus–Naur form in this book, because, while a language like Lua can be described using a metalanguage, it can also be described using words and sentences, in English, and this is exactly what this book is going to do.

Since English can be used to describe another language, then it must itself be a metalanguage (because it corresponds to the definition of a metalanguage). This is indeed the case. And since the purpose of a programming language is to describe instructions, and you can do that with English, English must also be a programming language. This, **in a way**, is also the case. In fact, English is a language that can be used for many things. But extended Backus–Naur form is a specialized language, and programming languages are also specialized languages. Specialization is the characteristic of being very good at doing something in particular, but not being capable of doing other things. Extended Backus–Naur form is very good at describing other languages, but it cannot be used to write instructions or to communicate a message. Programming languages are very good at giving instructions, but they cannot be used to describe languages or to communicate messages.

English is capable of doing everything: describing languages, giving instructions and communicating messages. But it is not very good at doing some of these. In fact, it is so bad at giving instructions that, if it is used to give instructions to a computer, the computer won't understand anything. That's because computers need the instructions to be very precise and unambiguous.

1.4 Obtaining Lua

Lua can be obtained on the official Lua website, on the download page[5]. Instructions are also available there: the download button is for the source code, which is probably not what you want. You are probably looking for binaries, so you should look around the page to find information about those (what exactly you are looking for depends on the platform you are using). The purpose of this book is only to teach the Lua language, not to teach usage of the Lua tools. It is generally assumed that the reader will be using Lua in an embedded environment, but this does not need to be the case for the book to be useful, only does it mean that the book will not describe the usage of Lua as a standalone language.

5 `http://www.lua.org/download.html`

2 Expressions

As explained before, expressions are pieces of code that have a value and that can be evaluated. They cannot be executed directly (with the exception of function calls), and thus, a script that would contain only the following code, which consists of an expression, would be erroneous:

```
3 + 5
-- The code above is erroneous because all it contains is an expression.
-- The computer cannot execute '3 + 5', since that does not make sense.
```

Code must be comprised of a sequence of statements. These statements can contain expressions which will be values the statement has to manipulate or use to execute the instruction.

Some code examples in this chapter do not constitute valid code, because they consist of only expressions. In the next chapter, statements will be covered and it will be possible to start writing valid code.

2.1 Types

To evaluate an expression is to compute it to find its value. The value a given expression evaluates to might be different from one context to another, since it can depend on the environment and stack level. This value will sometimes be a number, sometimes text and the other times any of many other data types, which is why it is said to have a type.

In Lua, and in programming in general, expressions will usually consist of one or more values with zero or more operators. Some operators can only be used with some types (it would be illogical, for example, to try to divide text, while it makes sense to divide numbers). There are two kinds of operators: unary operators and binary operators. Unary operators are operators that only take one value. For example, the unary - operator only takes one number as a parameter: -5, -3, -6, etc. It takes one number as a parameter and negates that number. The binary - operator, however, which is not the same operator, takes two values and subtracts the second from the first: 5 - 3, 8 - 6, 4 - 9, etc.

It is possible to obtain a number's type as a string with the `type` function:

```
print(type(32425)) --> number
```

2.1.1 Numbers

Numbers generally represent quantities, but they can be used for many other things. The number type in Lua works mostly in the same way as real numbers. Numbers can be

constructed as integers, decimal numbers, decimal exponents or even in hexadecimal[1]. Here are some valid numbers:

- 3
- 3.0
- 3.1416
- 314.16e-2
- 0.31416E1
- 0xff
- 0x56

Arithmetic operations

The operators for numbers in Lua are the following:

Operation	Syntax	Description	Example
Arithmetic negation	-*a*	Changes the sign of *a* and returns the value	-3.14159
Addition	*a* + *b*	Returns the sum of *a* and *b*	5.2 + 3.6
Subtraction	*a* - *b*	Subtracts *b* from *a* and returns the result	6.7 - 1.2
Multiplication	*a* * *b*	Returns the product of *a* and *b*	3.2 * 1.5
Exponentiation	*a* ^ *b*	Returns *a* to the power *b*, or the exponentiation of *a* by *b*	5 ^ 2
Division	*a* / *b*	Divides *a* by *b* and returns the result	6.4 / 2
Modulo operation	*a* % *b*	Returns the remainder of the division of *a* by *b*	5 % 3

You probably already know all of these operators (they are the same as basic mathematical operators) except the last. The last is called the modulo operator, and simply calculates the remainder of the division of one number by another. 5 % 3, for example, would give 2 as a result because 2 is the remainder of the division of 5 by 3. The modulo operator is less common than the other operators, but it has multiple uses.

Integers

A new subtype of numbers, integers, was added in Lua 5.3. Numbers can be either integers or floats. Floats are similar to numbers as described above, while integers are numbers with no decimal part. Float division (`/`) and exponentiation always convert their operands to floats, while all other operators give integers if their two operands were integers. In other cases, with the exception of the floor division operator (`//`), the result is a float.

1 `https://en.wikipedia.org/wiki/hexadecimal`

2.1.2 Nil

Nil is the type of the value nil, whose main property is to be different from any other value; it usually represents the absence of a useful value. A function that would return nil, for example, is a function that has nothing useful to return (we'll talk later about functions).

2.1.3 Booleans

A boolean value can be either true or false, but nothing else. This is literally written in Lua as `true` or `false`, which are reserved keywords. The following operators are often used with boolean values, but can also be used with values of any data type:

Operation	Syntax	Description
Boolean negation	not *a*	If *a* is false or nil, returns true. Otherwise, returns false.
Logical conjunction	*a* and *b*	Returns the first argument if it is false or nil. Otherwise, returns the second argument.
Logical disjunction	*a* or *b*	Returns the first argument if it is neither false nor nil. Otherwise, returns the second argument.

Essentially, the `not` operator just negates the boolean value (makes it false if it is true and makes it true if it is false), the `and` operator returns true if both are true and false if not and the `or` operator returns true if either of arguments is true and false otherwise. This is however not exactly how they work, as the exact way they work is explained in the table above. In Lua, the values false and nil are both considered as false, while everything else is considered as true, and if you do the logic reasoning, you'll realize that the definitions presented in this paragraph correspond with those in the table, although those in the table will not always return a boolean value.

The relational operators introduced in the next chapter (`<`, `>`, `<=`, `>=`, `~=`, `==`) do not necessarily take boolean values as operands, but will always give a boolean value as a result.

2.1.4 Strings

Strings are sequences of characters that can be used to represent text. They can be written in Lua by being contained in double quotes, single quotes or long brackets, which were covered before in the section about comments[2] (it should be noted that comments and strings have nothing in common other than the fact they can both be delimited by long brackets, preceded by two hyphens in the case of comments). Strings that aren't contained in long brackets will only continue for one line. Because of this, the only way to make a string that contains many lines without using long brackets is to use escape sequences. This is also the only way to insert single or double quotes in certain cases. Escape sequences

2 `https://en.wikibooks.org/wiki/%2FIntroduction%23Comments`

consist of two things: an escape character, which will always be a backslash ('\') in Lua, and an identifier that identifies the character to be escaped.

Escape sequences in Lua	
Escape sequence	**Description**
\n	A new line
\"	A double quote
\'	A single quote (or apostrophe)
\\	A backslash
\t	A horizontal tab
\###	### must be a number from 0 to 255. The result will be the corresponding ASCII[3] character.

Escape sequences are used when putting the character directly in the string would cause a problem. For example, if you have a string of text that is enclosed in double quotes and must contain double quotes, then you need to enclose the string in different characters or to escape the double quotes. Escaping characters in strings delimited by long brackets is not necessary, and this is true for all characters. All characters in a string delimited with long brackets will be taken as-is. The % character is used in string patterns to escape magic characters, but the term *escaping* is then used in another context.

```
"This is a valid string."

'This is also a valid string.'

"This is a valid \" string 'that contains unescaped single quotes and escaped
 double quotes."

[[
This is a line that can continue
on more than one line.

It can contain single quotes, double quotes and everything else (-- including
 comments). It ignores everything (including escape characters) except closing
 long brackets of the same level as the opening long bracket.
]]

"This is a valid string that contains tabs \t, double quotes \" and backlashes
 \\"

"This is " not a valid string because there is an unescaped double quote in the
 middle of it."
```

For convenience, if an opening long string bracket is immediately followed by a new line, that new line will be ignored. Therefore, the two following strings are equivalent:

```
[[This is a string
that can continue on many lines.]]

[[
This is a string
that can continue on many lines.]]
```

3 https://en.wikipedia.org/wiki/ASCII

```
-- Since the opening long bracket of the second string is immediately followed
 by a new line, that new line is ignored.
```

It is possible to get the length of a string, as a number, by using the unary length operator ('#'):

```
print(#("This is a string")) --> 16
```

Concatenation

In formal language theory[a] and computer programming[b], **string concatenation** is the operation of joining two character strings[c] end-to-end. For example, the concatenation of "snow" and "ball" is "snowball".

a https://en.wikipedia.org/wiki/formal%20language
b https://en.wikipedia.org/wiki/computer%20programming
c https://en.wikipedia.org/wiki/character%20string%20%28computer%20science%29

The string concatenation operator in Lua is denoted by two dots ('..'). Here is an example of concatenation that concatenates "snow" and "ball" and prints the result:

```
print("snow" .. "ball") --> snowball
```

This code will concatenate "snow" and "ball" and will print the result.

2.1.5 Other types

The four basic types in Lua (numbers, booleans, nil and strings) have been described in the previous sections, but four types are missing: functions, tables, userdata and threads. *Functions* are pieces of code that can be called, receive values and return values back. *Tables* are data structures that can be used for data manipulation. *Userdata* are used internally by applications Lua is embedded in to allow Lua to communicate with that program through objects controlled by the application. Finally, *threads* are used by coroutines, which allow many functions to run at the same time. These will all be described later, so you only need to keep in mind that there are other data types.

2.2 Literals

Literals are notations for representing fixed values in source code. All values can be represented as literals in Lua except threads and userdata. String literals (literals that evaluate to strings), for example, consist of the text that the string must represent enclosed into single quotes, double quotes or long brackets. Number literals, on the other hand, consist the number they represent expressed using decimal notation (ex: `12.43`), scientific notation (ex: `3.1416e-2` and `0.31416E1`) or hexadecimal notation (ex: `0xff`).

2.3 Coercion

Coercion is the conversion of a value of one data type to a value of another data type. Lua provides automatic coercion between string and number values. Any arithmetic operation applied to a string will attempt to convert this string to a number. Conversely, whenever a string is expected and a number is used instead, the number will be converted to a string. This applies both to Lua operators and to default functions (functions that are provided with the language).

```
print("122" + 1) --> 123
print("The number is " .. 5 .. ".") --> The number is 5.
```

Coercion of numbers to strings and strings to numbers can also be done manually with the `tostring` and `tonumber` functions. The former accepts a number as an argument and converts it to a string, while the second accepts a string as an argument and converts it to a number (a different base than the default decimal one can optionally be given in the second argument).

2.4 Bitwise operations

Since Lua 5.3, bitwise operators are provided to operate on binary numerals (bit patterns). These operators are not used as frequently as the others, so you may skip this section if you do not need them.

The bitwise operators in Lua always operate on integers, converting their operands if this is necessary. They also give integers.

The *bitwise AND* operation (with operator `&`) performs logical conjunction on each pair of bits of two binary representations of equal length. For example, `5 & 3` evaluates to 1. We can explain this by looking at the binary representation of these numbers (the subscripts are used to denote the base):

$(5)_{10} = (0101)_2$

$(3)_{10} = (0011)_2$

$(1)_{10} = (0001)_2$

If the bits in a given position in the binary representation of both 5 and 3 are 1 (as is the case for the last bit), then the bit at that position will be 1 in the result; in all other cases, it will be 0.

The *bitwise OR* operation (with operator `|`) works in the same way as the bitwise AND, performing logical disjunction instead where it performs logical conjunction. Thus, `5 | 3` will evaluate to 7:

$(5)_{10} = (0101)_2$

$(3)_{10} = (0011)_2$

$(7)_{10} = (0111)_2$

Here, we can see that the bit in each position in the final result was 0 only when the binary representations of the two operands had a 0-bit at that position.

The *bitwise XOR* operation (with operator ˜) works like two others, but at a given position, the final bit is only 1 if one, and not both, of the bits in the operands are 1.

$(5)_{10} = (0101)_2$

$(3)_{10} = (0011)_2$

$(6)_{10} = (0110)_2$

This is the same as the previous example, but we can see that the last bit in the result is 0 instead of 1, since the last bit of both operands was 1.

The *bitwise NOT* operation (with operator ˜) performs logical negation on each bit of its unique operand, which means that each 0 becomes 1 and that each 1 becomes 0. Thusly, `~7` will evaluate to -8:

$(7)_{10} = (0111)_2$

$(8)_{10} = (1000)_2$

Here, the first bit became 1 in the result because it was 0 in the operand, and the other bits became 0 because they were all 1.

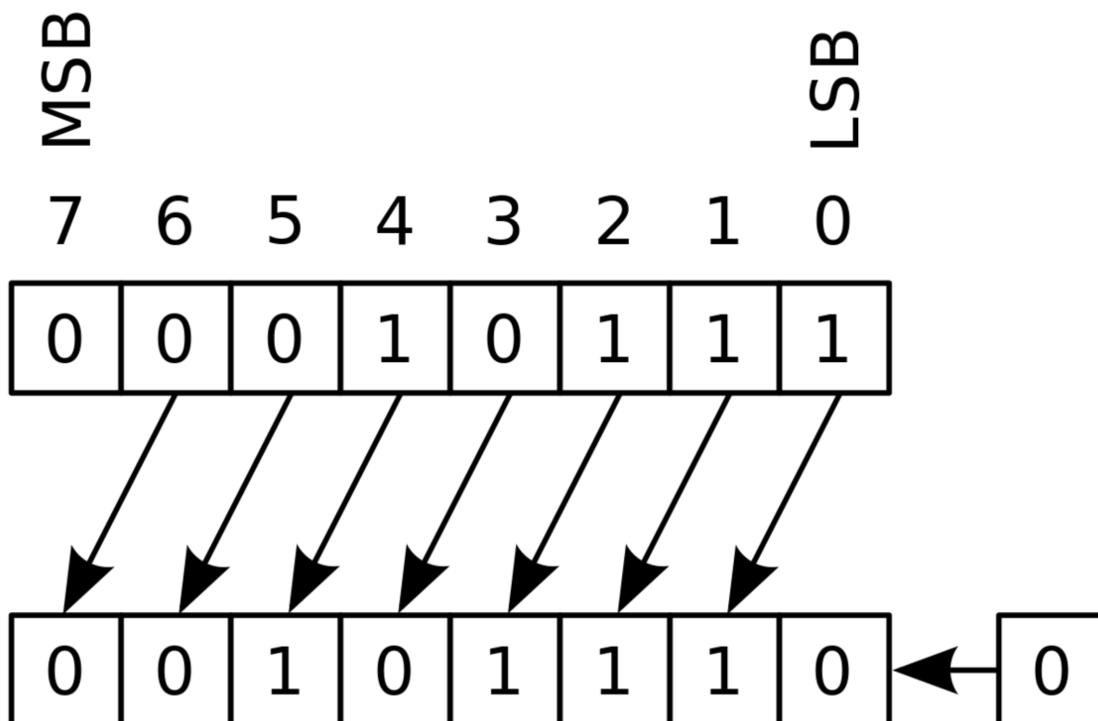

Figure 1 Left shift

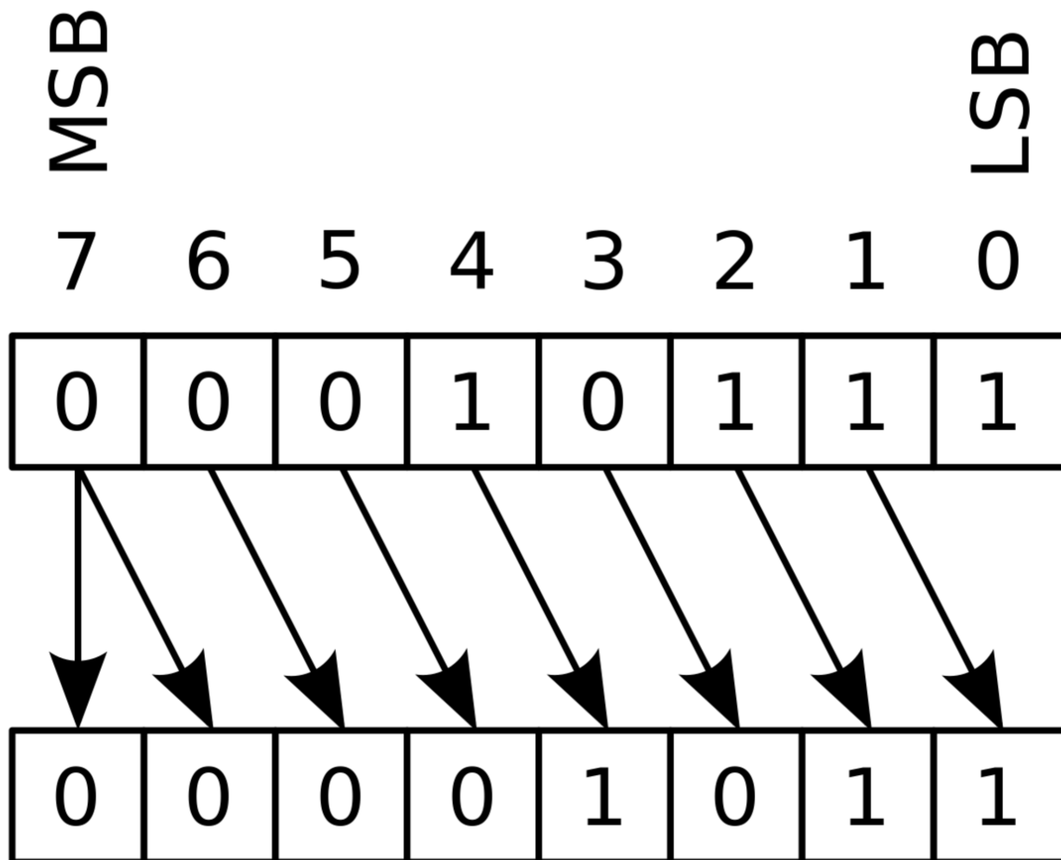

Figure 2 Right shift

In addition to these bitwise operators, Lua 5.3 also supports arithmetic bit shifts. The *left shift*, with operator << and illustrated on left, consists in shifting all bits to the left, by a number of bits that corresponds to the second operand. The *right shift*, denoted by operator >> and illustrated on right, does the same but in the opposite direction.

2.5 Operator precedence

Operator precedence works the same way in Lua as it typically does in mathematics. Certain operators will be evaluated before others, and parentheses can be used to arbitrarily change the order in which operations should be executed. The priority in which operators are evaluated is in the list below, from higher to lower priority. Some of these operators were not discussed yet, but they will all be covered at some point in this book.

1. Exponentiation: `^`
2. Unary operators: `not`, `#`, `-`, `~`
3. Level 2 mathematical operators: `*`, `/`, `//`, `%`
4. Level 1 mathematical operators: `+`, `-`
5. Concatenation: `..`

6. Bit shifts: <<, >>
7. Bitwise AND: `&`
8. Bitwise XOR: `~`
9. Bitwise OR: `|`
10. Relational operators: <, >, <=, >=, `~=`, `==`
11. Boolean and: `and`
12. Boolean or: `or`

2.6 Quiz

There are some questions you can answer to verify that you have understood the material in this chapter. Note that finding the answer to some of those questions could require having knowledge that is not presented in this chapter. This is normal: the quizzes are part of the learning experience, and they can introduce information that is not available elsewhere in the book.

<quiz display="simple"> {What will `print(type(type(5.2)))` output? | type="{}"} { string (i) }

{What will the expression `0 or 8` return? | type="()"} { -

```
true
```

|| `and` and `or` return their arguments rather than a boolean value. -

```
false
```

|| All values different than `false` and `nil` are considered as 'true' (even 0). +

```
0
```

|| `or` returns the first argument, since it is neither `false` nor `nil`. -

```
8
```

|| `or` returns the first argument, since it is neither `false` nor `nil`. 0 is considered as 'true'. }

{Which strings are valid? | type="[]"} { + `"test's test"`|| No escaping for the single quote required, since the string is limited by double quotes. + `'test\'s test'`|| Escaping is necessary here. - `"test"s test"`|| There is a double quote in the middle of a double quoted string. + `'test"s test'`|| No escaping for the double quote required, since the string is limited by single quotes. + `"test\'s test"`|| Escaping is optional here. - `'test's test'`|| There is a single quote in the middle of a single quoted string. }

{Which expressions give the string `"1223"`? | type="[]"} { - `"122" + 3`|| Yields the number 125. + `"122" .. 3`- `"12" + "23"`|| Yields the number 35. + `12 .. 23`}

{True or false? `not 5^3 == 5`| type="()"} { - `true`|| Beware the operator precedence (`not` is higher than `==`)! The expression is equivalent to `(not 5^3) == 5`, which evaluates first to `false == 5` and thus to `false`. + `false`} </quiz>

3 Statements

Statements are pieces of code that can be executed and that contain an instruction and expressions to use with it. Some statements will also contain code inside of themselves that may, for example, be run under certain conditions. Dissimilarly to expressions, they can be put directly in code and will execute. Lua has few instructions, but these instructions, combined with other instructions and with complex expressions, give a good amount of control and flexibility to the user.

3.1 Assignment

Programmers frequently need to be able to store values in the memory to be able to use them later. This is done using variables. *Variables* are references to a value which is stored in the computer's memory. They can be used to access a number later after storing it in the memory. *Assignment* is the instruction that is used to assign a value to a variable. It consists of the name of the variable the value should be stored in, an equal sign, and the value that should be stored in the variable:

```
variable = 43
print(variable) --> 43
```

As demonstrated in the above code, the value of a variable can be accessed by putting the variable's name where the value should be accessed.

3.1.1 Identifiers

Identifiers[1], in Lua, are also called names. They can be any text composed of letters, digits, and underscores and not beginning with a digit. They are used to name variables and table fields, which will be covered in the chapter about tables.

Here are some valid names:

- `name`
- `hello`
- `_`
- `_tomatoes`
- `me41`
- `__`
- `_thisIs_StillaValid23name`

1 https://en.wikipedia.org/wiki/Identifier%23In_computer_science

Here are some invalid names:

- `2hello` : starts with a digit
- `th$i` : contains a character that isn't a letter, a digit or an underscore
- `hel!o` : contains a character that isn't a letter, a digit or an underscore
- `563text` : starts with a digit
- `82_something` : starts with a digit

Also, the following keywords are reserved by Lua and can not be used as names: `and`, `end`, `in`, `repeat`, `break`, `false`, `local`, `return`, `do`, `for`, `nil`, `then`, `else`, `function`, `not`, `true`, `elseif`, `if`, `or`, `until`, `while`.

When naming a variable or a table field, you must choose a valid name for it. It must therefore start with a letter or an underscore and only contain letters, underscores and digits. Note that Lua is case sensitive. This means that `Hello` and `hello` are two different names.

3.1.2 Scope

The scope of a variable[2] is the region of the code of the program where that variable is meaningful. The examples of variables you have seen before are all examples of global variables, variables which can be accessed from anywhere in the program. Local variables, on the other hand, can only be used from the region of the program in which they were defined and in regions of the program that are located inside that region of the program. They are created exactly in the same way as global variables, but they must be prefixed with the `local` keyword.

The `do` statement will be used to describe them. The `do` statement is a statement that has no other purpose than to create a new block of code, and therefore a new scope. It ends with the `end` keyword:

```
local variable = 13 -- This defines a local variable that can be accessed from
 anywhere in the script since it was defined in the main region.
do
	-- This statement creates a new block and also a new scope.
	variable = variable + 5 -- This adds 5 to the variable, which now equals 18.
	local variable = 17 -- This creates a variable with the same name as the
 previous variable, but this one is local to the scope created by the do
 statement.
	variable = variable - 1 -- This subtracts 1 from the local variable, which
 now equals 16.
	print(variable) --> 16
end
print(variable) --> 18
```

When a scope ends, all the variables in it are gotten rid of. Regions of code can use variables defined in regions of code they are included in, but if they "overwrite" them by defining a local variable with the same name, that local variable will be used instead of the one defined in the other region of code. This is why the first call to the print function prints `16` while the second, which is outside the scope created by the `do` statement, prints `18`.

2 https://en.wikipedia.org/wiki/Scope%20%28computer%20science%29

In practice, only local variables should be used because they can be defined and accessed faster than global variables, since they are stored in registers instead of being stored in the environment of the current function, like global variables. Registers are areas that Lua uses to store local variables to access them quickly, and can only usually contain up to 200 local variables. The processor, an important component of all computers, also has registers, but these are not related to Lua's registers. Each function (including the main thread, the core of the program, which is also a function) also has its own environment, which is a table that uses indices for the variable names and stores the values of these variables in the values that correspond to these indices.

3.1.3 Forms of assignment

Some assignment patterns are sufficiently common for syntactic sugar to have been introduced to make their use simpler.

Augmented assignment

Augmented assignment[3], which is also called *compound assignment*, is a type of assignment that gives a variable a value that is relative to its previous value. It is used when it is necessary to change the value of a variable in a way that is relative to its previous value, such as when that variable's value must be incremented. In C[4], JavaScript[5], Ruby[6], Python[7] and some other languages, the code `a += 8` will increment the value of *a* by 8. Lua does not have syntactic sugar for augmented assignment, which means that it is necessary to write `a = a + 8`.

Chained assignment

Chained assignment[8] is a type of assignment that gives a single value to many variables. The code `a = b = c = d = 0`, for example, would set the values of *a*, *b*, *c* and *d* to 0 in C and Python. In Lua, this code will raise an error because Lua does not have syntactic sugar for chained assignment, so it is necessary to write the previous example like this:

```
d = 0
c = d -- or c = 0
b = c -- or b = 0
a = b -- or a = 0
```

3 https://en.wikipedia.org/wiki/augmented%20assignment
4 https://en.wikipedia.org/wiki/C%20%28programming%20language%29
5 https://en.wikipedia.org/wiki/JavaScript
6 https://en.wikipedia.org/wiki/Ruby%20%28programming%20language%29
7 https://en.wikipedia.org/wiki/Python%20%28programming%20language%29
8 https://en.wikipedia.org/wiki/chained%20assignment

Parallel assignment

Parallel assignment[9], which is also called *simultaneous assignment* and *multiple assignment*, is a type of assignment that simultaneously assigns different values (they can also be the same value) to different variables. Unlike chained assignment and augmented assignment, parallel assignment is available in Lua.

The example in the previous section can be rewritten to use parallel assignment:

```
a, b, c, d = 0, 0, 0, 0
```

If you provide more variables than values, some variables will be not be assigned any value. If you provide more values than variables, the extra values will be ignored. More technically, the list of values is adjusted to the length of list of variables before the assignment takes place, which means that excess values are removed and that extra nil values are added at its end to make it have the same length as the list of variables. If a function call is present *at the end of the values list*, the values it returns will be added at the end of that list, unless the function call is put between parentheses.

Moreover, unlike most programming languages Lua enables reassignment of variables' values through permutation[10]. For example:

```
first_variable, second_variable = 54, 87
first_variable, second_variable = second_variable, first_variable
print(first_variable, second_variable) --> 87 54
```

This works because the assignment statement evaluates all the variables and values before assigning anything. Assignments are performed as if they were really simultaneous, which means you can assign at the same time a value to a variable and to a table field indexed with that variable's value before it is assigned a new value. In other words, the following code will set `dictionary[2]`, and not `dictionary[1]`, to 12.

```
dictionary = {}
index = 2
index, dictionary[index] = index - 1, 12
```

3.2 Conditional statement

Conditional statements are instructions that check whether an expression is true and execute a certain piece of code if it is. If the expression is not true, they just skip over that piece of code and the program continues. In Lua, the only conditional statement uses the `if` instruction. False and nil are both considered as false, while everything else is considered as true.

```
local number = 6

if number < 10 then
	print("The number " .. number .. " is smaller than ten.")
```

9 https://en.wikipedia.org/wiki/parallel%20assignment
10 https://en.wikipedia.org/wiki/permutation

```
end

-- Other code can be here and it will execute regardless of whether the code in
 the conditional statement executed.
```

In the code above, the variable *number* is assigned the number 6 with an assignment statement. Then, a conditional statement checks if the value stored in the variable *number* is smaller than ten, which is the case here. If it is, it prints `"The number 6 is smaller than ten."`.

It is also possible to execute a certain piece of code *only* if the expression was not true by using the `else` keyword and to chain conditional statements with the `elseif` keyword:

```
local number = 15

if number < 10 then
	print("The number is smaller than ten.")
elseif number < 100 then
	print("The number is bigger than or equal to ten, but smaller than one
 hundred.")
elseif number ~= 1000 and number < 3000 then
	print("The number is bigger than or equal to one hundred, smaller than three
 thousands and is not exactly one thousand.")
else
	print("The number is either 1000 or bigger than 2999.")
end
```

Note that the `else` block must always be the last one. There cannot be an `elseif` block after the `else` block. The `elseif` blocks are only meaningful if none of the blocks that preceded them was executed.

Operators used to compare two values, some of which are used in the code above, are called relational operators. If the relation is true, they return the boolean value `true`. Otherwise, they return the boolean value `false`.

	equal to	not equal to	greater than	less than	greater than or equal to	less than or equal to
Mathematical notation	$=$	$\neq$	$>$	$<$	$\geq$	$\leq$
Lua operator	==	~=	>	<	>=	<=

The code above also demonstrates how the `and` keyword can be used to combine many boolean expressions in a conditional expression.

3.3 Loops

Frequently, programmers will need to run a certain piece of code or a similar piece of code many times, or to run a certain piece of code a number of times that may depend on user input. A loop is a sequence of statements which is specified once but which may be carried out several times in succession.

3.3.1 Condition-controlled loops

Condition-controlled loops are loops that are controlled by a condition. They are very similar to conditional statements, but instead of executing the code if the condition is true

and skipping it otherwise, they will keep running it while the condition is true, or until the condition is false. Lua has two statements for condition-controlled loops: the `while` loop and the `repeat` loop. Such loops will run code, then check if the condition is true. If it is true, then they run the code again, and they repeat until the condition is false. When the condition is false, they stop repeating the code and the program flow continues. Each execution of the code is called an iteration. The difference between `while` and `repeat` loops is that `repeat` loops will check the condition at the end of the loop while `while` loops will check it at the start of the loop. This only makes a difference for the first iteration: `repeat` loops will always execute the code at least once, even if the condition is false at the first time the code is executed. This is not the case for `while` loops, which will only execute the code the first time if the condition is actually true.

```
local number = 0

while number < 10 do
	print(number)
	number = number + 1 -- Increase the value of the number by one.
end
```

The code above will print `0`, then `1`, then `2`, then `3`, and so on, until `9`. After the tenth iteration, *number* will no longer be smaller than ten, and therefore the loop will stop executing. Sometimes, loops will be meant to run forever, in which case they are called infinite loops. Renderers, software processes that draw things on the screen, for example, will often loop constantly to redraw the screen to update the image that is shown to the user. This is frequently the case in video games, where the game view must be updated constantly to make sure what the user sees is kept up-to-date. However, cases where loops need to run forever are rare and such loops will often be the result of errors. Infinite loops can take a lot of computer resources, so it is important to make sure that loops will always end even if unexpected input is received from the user.

```
local number = 0

repeat
	print(number)
	number = number + 1
until number >= 10
```

The code above will do exactly the same thing as the code that used a `while` loop above. The main differences is that, unlike `while` loops, where the condition is put between the `while` keyword and the `do` keyword, the condition is put at the end of the loop, after the `until` keyword. The `repeat` loop is the only statement in Lua that creates a block and that is not closed by the `end` keyword.

3.3.2 Count-controlled loops

Incrementing a variable is increasing its value by steps, especially by steps of one. The two loops in the previous section incremented the variable *number* and used it to run the code a certain number of times. This kind of loop is so common that most languages, including Lua, have a built-in control structure for it. This control structure is called a count-controlled loop, and, in Lua and most languages, is defined by the `for` statement. The variable used in such loops is called the loop counter.

```
for number = 0, 9, 1 do
        print(number)
end
```

The code above does exactly the same thing as the two loops presented in the previous section, but the *number* variable can only be accessed from inside the loop because it is local to it. The first number following the variable name and the equality symbol is the initialization. It is the value the loop counter is initialized to. The second number is the number the loop stops at. It will increment the variable and repeat the code until the variable reaches this number. Finally, the third number is the increment: it is the value the loop counter is increased of at each iteration. If the increment is not given, it will be assumed to be 1 by Lua. The code below would therefore print `1`, `1.1`, `1.2`, `1.3`, `1.4` and `1.5`.

```
for n = 1, 2, 0.1 do
        print(n)
        if n >= 1.5 then
                break -- Terminate the loop instantly and do not repeat.
        end
end
```

The reason the code above does not go up to 2 and only up to 1.5 is because of the `break` statement, which instantly terminates the loop. This statement can be used with any loop, including `while` loops and `repeat` loops. Note that the >= operator was used here, although the == operator would theoretically have done the job as well. This is because of decimal precision errors. Lua represents numbers with the double-precision floating-point format[11], which stores numbers in the memory as an approximation of their actual value. In some cases, the approximation will match the number exactly, but in some cases, it will only be an approximation. Usually, these approximations will be close enough to the number for it to not make a difference, but this system can cause errors when using the equality operator. This is why it is generally safer when working with decimal numbers to avoid using the equality operator. In this specific case, the code would not have worked if the equality operator had been used[12] (it would have continued going up until 1.9), but it works with the >= operator.

3.4 Blocks

A block is a list of statements that are executed sequentially. These statements can include empty statements, that do not contain any instruction. Empty statements can be used to start a block with a semicolon or write two semicolons in sequence.

Function calls and assignments may start with a parenthesis, which can lead to an ambiguity. This fragment is an example of this:

```
a = b + c
(print or io.write)('done')
```

11 https://en.wikipedia.org/wiki/Double-precision%20floating-point%20format
12 http://codepad.org/kYHPSvqx

This code could be interpreted in two ways:

```
a = b + c(print or io.write)('done')
a = b + c; (print or io.write)('done')
```

The current parser always sees such constructions in the first way, interpreting the opening parenthesis as the start of the arguments to a call. To avoid this ambiguity, it is a good practice to always precede with a semicolon statements that start with a parenthesis:

```
;(print or io.write)('done')
```

3.4.1 Chunks

The unit of compilation of Lua is called a *chunk*. A chunk can be stored in a file or in a string inside the host program. To execute a chunk, Lua first precompiles the chunk into instructions for a virtual machine, and then it executes the compiled code with an interpreter for the virtual machine. Chunks can also be precompiled into binary form (bytecode) using `luac`, the compilation program that comes with Lua, or the `string.dump` function, which returns a string containing a binary representation of the function it is given.

The `load` function can be used to load a chunk. If the first parameter given to the `load` function is a string, the chunk is that string. In this case, the string may be either Lua code or Lua bytecode. If the first parameter is a function, `load` will call that function repeatedly to get the pieces of the chunk, each piece being a string that will be concatenated with the previous strings. It is then considered that the chunk is complete when nothing or the empty string is returned.

The `load` function will return the compiled chunk as a function if there is no syntactic error. Otherwise, it will return nil and the error message.

The second parameter of the `load` function is used to set the source of the chunk. All chunks keep a copy of their source within them, in order to be able to give appropriate error messages and debugging information. By default, that copy of their source will be the code given to `load` (if code was given; if a function was given instead, it will be ”=(load)”). This parameter can be used to change it. This is mostly useful when compiling code to prevent people from getting the original source back. It is then necessary to remove the source included with the binary representation because otherwise the original code can be obtained there.

The third parameter of the `load` function can be used to set the environment of the generated function and the fourth parameter controls whether the chunk can be in text or binary. It may be the string ”b” (only binary chunks), ”t” (only text chunks), or ”bt” (both binary and text). The default is ”bt”.

There is also a `loadfile` function that works exactly like `load`, but instead gets the code from a file. The first parameter is the name of the file from which to get the code. There is no parameter to modify the source stored in the binary representation, and the third and fourth parameters of the `load` function correspond to the second and third parameters of this function. The `loadfile` function can also be used to load code from the standard input, which will be done if no file name is given.

The `dofile` function is similar to the `loadfile` function, but instead of loading the code in a file as a function, it immediately executes the code contained in a source code file as a Lua chunk. Its only parameter is used to specify the name of the file it should execute the contents of; if no argument is given, it will execute the contents of the standard input. If the chunk returns values, they will be provided by the call to the `dofile` function. Because `dofile` does not run in protected mode, all errors in chunks executed through it will propagate.

4 Functions

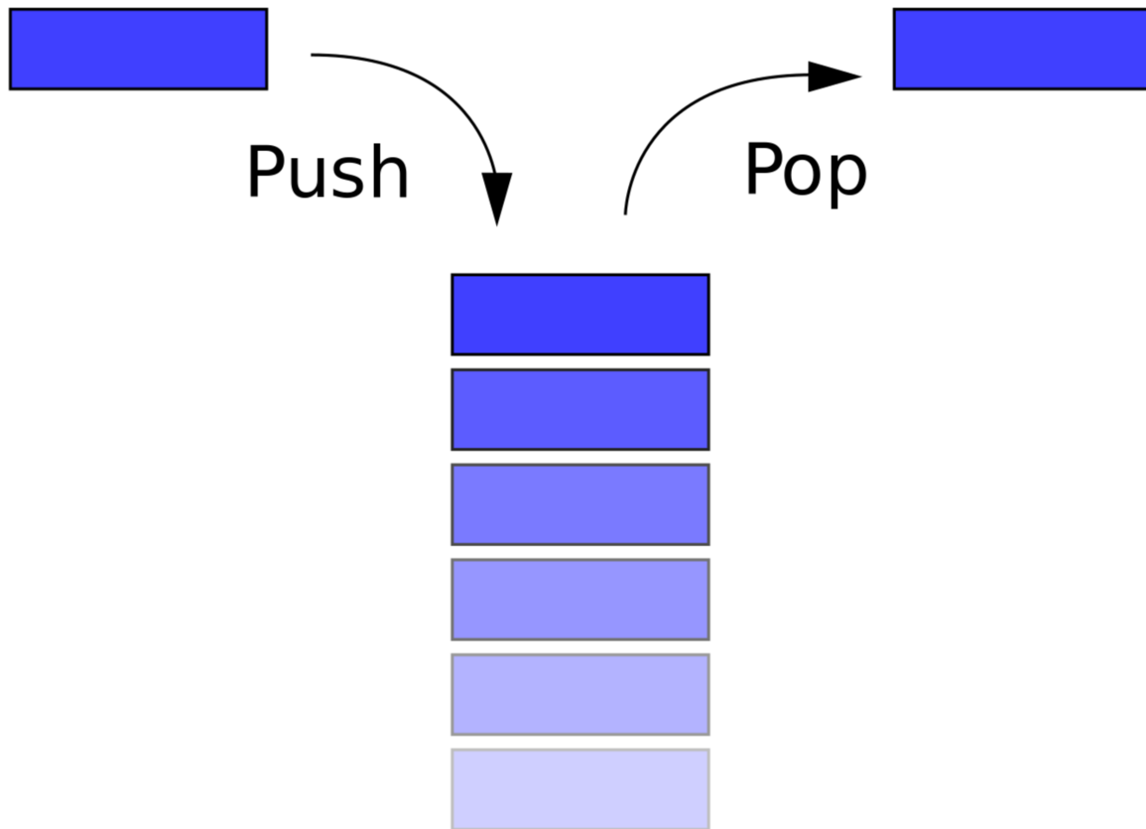

Figure 3 alt=An illustration of a stack and of the operations that can be performed on it.

A *stack* is a list of items where items can be added (*pushed*) or removed (*popped*) that behaves on the last-in-first-out principle, which means that the last item that was added will be the first to be removed. This is why such lists are called stacks: on a stack, you cannot remove an item without first removing the items that are on top of it. All operations therefore happen at the top of the stack. An item is above another if it was added after that item and is below it if it was added before that item.

A **function** (also called a subroutine, a procedure, a routine or a subprogram) is a sequence of instructions that perform a specific task and that can be *called* from elsewhere in the program whenever that sequence of instructions should be executed. Functions can also receive values as input and return an output after potentially manipulating the input or executing a task based on the input. Functions can be defined from anywhere in a program, including inside other functions, and they can also be called from any part of the program that has access to them: functions, just like numbers and strings, are values and can

therefore be stored in variables and have all the properties that are common to variables. These characteristics make functions very useful.

Because functions can be called from other functions, the Lua interpreter (the program that reads and executes Lua code) needs to be able to know what function called the function it is currently executing so that, when the function terminates (when there is no more code to execute), it can return to execution of the right function. This is done with a stack called the call stack: each item in the call stack is a function that called the function that is directly above it in the stack, until the last item in the stack, which is the function currently being executed. When a function terminates, the interpreter uses the stack's pop operation to remove the last function in the list, and it then returns to the previous function.

There are two types of functions: built-in functions and user-defined functions. *Built-in functions* are functions provided with Lua and include functions such as the `print` function, which you already know. Some can be accessed directly, like the `print` function, but others need to be accessed through a library, like the `math.random` function, which returns a random number. *User-defined functions* are functions defined by the user. User-defined functions are defined using a function constructor:

```
local func = function(first_parameter, second_parameter, third_parameter)
        -- function body (a function's body is the code it contains)
end
```

The code above creates a function with three parameters and stores it in the variable *func*. The following code does exactly the same as the above code, but uses syntactic sugar for defining the function:

```
local function func(first_parameter, second_parameter, third_parameter)
        -- function body
end
```

It should be noted that, when using the second form, it is possible to refer to the function from inside itself, which is not possible when using the first form. This is because `local function foo() end` translates to `local foo; foo = function() end` rather than `local foo = function() end`. This means that *foo* is part of the function's environment in the second form and not in the first, which explains why the second form makes it possible to refer to the function itself.

In both cases, it is possible to omit the `local` keyword to store the function in a global variable. Parameters work like variables and allow functions to receive values. When a function is called, arguments may be given to it. The function will then receive them as parameters. Parameters are like local variables defined at the beginning of a function, and will be assigned in order depending on the order of the arguments as they are given in the function call; if an argument is missing, the parameter will have the value `nil`. The function in the following example adds two numbers and prints the result. It would therefore print `5` when the code runs.

```
local function add(first_number, second_number)
        print(first_number + second_number)
end

add(2, 3)
```

Function calls are most of the time under the form `name(arguments)`. However, if there is only one argument and it is either a table or a string, and it isn't in a variable (meaning it is constructed directly in the function call, expressed as a literal), the parentheses can be omitted:

```
print "Hello, world!"
print {4, 5}
```

The second line of code in the previous example would print the memory address of the table. When converting values to strings, which the `print` function does automatically, complex types (functions, tables, userdata and threads) are changed to their memory addresses. Booleans, numbers and the nil value, however, will be converted to corresponding strings.

The terms *parameter* and *argument* are often used interchangeably in practice. In this book, and in their proper meanings, the terms *parameter* and *argument* mean, respectively, a name to which the value of the corresponding argument will be assigned and a value that is passed to a function to be assigned to a parameter.

4.1 Returning values

Functions can receive input, manipulate it and give back output. You already know how they can receive input (parameters) and manipulate it (function body). They can also give output by returning one or many values of any type, which is done using the return statement. This is why function calls are both statements and expressions: they can be executed, but they can also be evaluated.

```
local function add(first_number, second_number)
	return first_number + second_number
end

print(add(5, 6))
```

The code in the above function will first define the function `add`. Then, it will call it with 5 and 6 as values. The function will add them and return the result, which will then be printed. This is why the code above would print `11`. It is also possible for a function to return many values by separating the expressions that evaluate to these values with commas.

4.2 Errors

There are three types of errors: syntactic errors, static semantic errors and semantic errors. Syntactic errors happen when code is plainly invalid. The following code, for example, would be detected by Lua as invalid:

```
print(5 ++ 4 return)
```

The code above doesn't make sense; it is impossible to get a meaning out of it. Similarly, in English, "cat dog tree" is not syntactically valid because it has no meaning. It doesn't follow the rules for creating a sentence.

Static semantic errors happen when code has a meaning, but still doesn't make sense. For example, if you try adding a string with a number, you get a static semantic error because it is impossible to add a string with a number:

```
print("hello" + 5)
```

The code above follows Lua's syntactic rules, but it still doesn't make sense because it is impossible to add a string with a number (except when the string represents a number, in which case it will be coerced into one). This can be compared in English to the sentence "I are big". It follows the rules for creating sentences in English, but it still doesn't make sense because "I" is singular and "are" is plural.

Finally, semantic errors are errors that happen when the meaning of a piece of code is not what its creator thinks it is. Those are the worst errors because they can be very hard to find. Lua will always tell you when there is a syntactic error or a static semantic error (this is called throwing an error), but it cannot tell you when there is a semantic error since it doesn't know what you think the meaning of the code is. These errors happen more often than most people would think they do and finding and correcting them is something many programmers spend a lot of time doing.

The process of finding errors and correcting them is called debugging. Most of the time, programmers will spend more time finding errors than actually correcting them. This is true for all types of errors. Once you know what the problem is, it is usually simple to fix it, but sometimes, a programmer can look at a piece of code for hours without finding what is wrong in it.

4.2.1 Protected calls

Throwing an error is the action of indicating, whether it is done manually or automatically by the interpreter (the program that reads the code and executes it), that something is wrong with the code. It is done automatically by Lua when the code given is invalid, but it can be done manually with the `error` function:

```
local variable = 500
if variable % 5 ~= 0 then
	error("It must be possible to divide the value of the variable by 5 without
 obtaining a decimal number.")
end
```

The `error` function also has a second argument, which indicates the stack level at which the error should be thrown, but this will not be covered in this book. The `assert` function does the same thing as the `error` function, but it will only throw an error if its first argument evaluates to nil or false and it doesn't have an argument that can be used to specify the stack level at which the error should be thrown. The assert function is useful at the start of a script, for example, to check if a library that is required for the script to work is available.

It may be hard to understand why one would desire to voluntarily throw an error, since the code in a program stops running whenever an error is thrown, but, often, throwing errors when functions are used incorrectly or when a program is not running in the right environment can be helpful to help the person who will have to debug the code to find it

immediately without having to stare at the code for a long time without realizing what is wrong.

Sometimes, it can be useful to prevent an error from stopping the code and instead do something like displaying an error message to the user so he can report the bug to the developer. This is called *exception handling* (or *error handling*) and is done by catching the error to prevent its propagation and running an exception handler to handle it. The way it is done in different programming languages varies a lot. In Lua, it is done using protected calls[1]. They are called protected calls because a function called in protected mode will not stop the program if an error happens. There are two functions that can be used to call a function in protected mode:

Function	Description
`pcall(function, ...)`	Calls the function in protected mode and returns a status code (a boolean value whose value depends on if an error was thrown or not) and the values returned by the function, or the error message if the function was stopped by an error. Arguments can be given to the function by passing them to the `pcall` function after the first argument, which is the function that should be called in protected mode.
`xpcall(function, handler, ...)`	Does the same thing as pcall, but, when the function errors, instead of returning the same values as those pcall would return, it calls the handler function with them as parameters. The handler function can then be used, for example, to display an error message. As for the `pcall` function, arguments can be passed to the function by being given to the `xpcall` function.

4.3 Stack overflow

The call stack, the stack that contains all the functions that were called in the order in which they were called, was mentioned earlier. That call stack in most languages, including Lua, has a maximum size. This maximum size is so big that it should not be worried about in most cases, but functions that call themselves (this is called recursivity and such functions are called recursive functions) can reach this limit if there is nothing to prevent them from calling themselves over and over indefinitely. This is called a stack overflow. When the stack overflows, the code stops running and an error is thrown.

1 For more information, see: Programming in Lua. Lua.org, , 2003

4.4 Variadic functions

Variadic functions, which are also called vararg functions, are functions that accept a variable number of arguments. A variadic function is indicated by three dots ("...") at the end of its parameter list. Arguments that do not fit in the parameters in the parameter list, instead of being discarded, are then made available to the function through a vararg expression, which is also indicated by three dots. The value of a vararg expression is a list of values (not a table) which can then be put in a table to be manipulated with more ease with the following expression: `{...}`. In Lua 5.0, instead of being available through a vararg expression, the extra arguments were available in a special parameter called "arg". The following function is an example of a function that would add the first argument to all the arguments it receives, then add all of them together and print the result:

```
function add_one(increment, ...)
	local result = 0
	for _, number in next, {...} do
		result = result + number + increment
	end
end
```

It is not necessary to understand the code above as it is only a demonstration of a variadic function.

The `select` function is useful to manipulate argument lists without needing to use tables. It is itself a variadic function, as it accepts an indefinite number of arguments. It returns all arguments after the argument with the number given as its first argument (if the number given is negative, it indexes starting from the end, meaning -1 is the last argument). It will also return the number of arguments it received, excluding the first one, if the first argument is the string "#". It can be useful to discard all arguments in an argument list before a certain number, and, more originally, to distinguish between nil values being sent as arguments and nothing being sent as an argument. Indeed, `select` will distinguish, when `"#"` is given as its first argument, nil values from no value. Argument lists (and return lists as well) are instances of tuples, which will be explored in the chapter about tables; the `select` function works with all tuples.

```
print((function(...) return select('#', ...) == 1 and "nil" or "no value"
 end)()) --> no value
print((function(...) return select('#', ...) == 1 and "nil" or "no value"
 end)(nil)) --> nil
print((function(...) return select('#', ...) == 1 and "nil" or "no value"
 end)(variable_that_is_not_defined)) --> nil

-- As this code shows, the function is able to detect whether the value nil was
 passed as an argument or whether there was simply no value passed.
-- In normal circumstances, both are considered as nil, and this is the only way
 to distinguish them.
```

5 Standard libraries

Lua is a language that is said to ”not be provided with batteries”. This means that its libraries are kept to the minimum necessary to do some stuff. Lua relies on its community to create libraries that can be used to perform more specific tasks. There are ten libraries available in Lua. The *Lua Reference Manual* provides documentation for all the libraries[1], so they will only be briefly described here[2]. All the libraries except the basic and the package libraries provide their functions and values as fields of a table.

5.1 Basic library

The basic library provides core functionality to Lua. All its functions and values are directly available in the global environment, and all functions and values available directly in the global environment by default are part of the basic library.

5.1.1 Assertion

An *assertion* is a predicate that is assumed by the developer to be true. They are used in programs to ensure that a specific condition is true at a specific moment of the execution of a program. Assertions are used in unit tests[3] to verify that a program works correctly, but are also used in program code, in which case the program will fail when an assertion is false, either to verify that the environment in which the program is correct, or to verify that no error was made in program code and to generate appropriate error messages to make it easier to find bugs in the code when something doesn't happen as expected. In Lua, assertions are made with the `assert` function, which accepts a condition and a message (which will default to ”assertion failed!”) as parameters. When the condition evaluates to false, `assert` throws an error with the message. When it evaluates to true, `assert` returns all its arguments.

5.1.2 Garbage collection

Garbage collection is a form of automatic memory management implemented by Lua and many other languages. When a program needs to store data in a variable, it asks the operating system to allocate space in the computer's memory to store the variable's value. Then, when it doesn't need the space anymore (generally because the variable fell out of

---

1 Lua 5.2 Reference Manual. , ,
2 Functions that were already described elsewhere will not be described in this chapter.
3 `https://en.wikibooks.org/wiki/..%2FUnit%20testing`

scope), it tells the operating system to deallocate the space so that another program may use it. In Lua, the actual process is much more complex, but the basic idea is the same: the program must tell the operating system when it doesn't need a variable's value anymore. In low level languages, allocation is handled by the language, but deallocation is not because the language cannot know when a programmer doesn't need a value anymore: even if a variable that referenced the value fell out of scope or was removed, another variable or a field in a script may still reference it, and deallocating it would cause problems. In higher level languages, deallocation may be handled by various automatic memory management systems, such as garbage collection, which is the system used by Lua. The garbage collector regularly searches through all the values allocated by Lua for values that are not referenced anywhere. It will collect values that the program cannot access anymore (because there is no reference to them) and, since it knows that these values can safely be deallocated, will deallocate them. This is all done transparently and automatically, so the programmer does not generally need to do anything about it. However, sometimes, the developer may want to give instructions to the garbage collector.

Weak references

Weak references are references that are ignored by the garbage collector. These references are indicated to the garbage collector by the developer, using the `mode` metamethod. A table's `mode` metamethod should be a string. If that string contains the letter "k", all the keys of the table's fields are weak, and if it contains the letter "v", all the values of the table's fields are weak. When an array of objects has weak values, the garbage collector will collect these objects even if they are referenced in that array, as long as they are only referenced in that array and in other weak references. This behavior is sometimes useful, but rarely used.

Manipulating the garbage collector

The garbage collector may be manipulated with the `collectgarbage` function, which is part of the basic library and serves as an interface to the garbage collector. Its first argument is a string that indicates to the garbage collector what action should be performed; a second argument is used by some actions. The `collectgarbage` function can be used to stop the garbage collector, manually perform collection cycles and count the memory used by Lua.

5.2 Coroutines

Coroutines are computer program[a] components that generalize subroutine[b]s to allow multiple entry point[c]s for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative task[d]s, exception[e]s, event loop[f], iterator[g]s, infinite list[h]s and pipe[i]s.

a `https://en.wikipedia.org/wiki/computer%20program`
b `https://en.wikipedia.org/wiki/subroutine`
c `https://en.wikipedia.org/wiki/entry%20point`
d `https://en.wikipedia.org/wiki/cooperative%20multitasking`
e `https://en.wikipedia.org/wiki/exception%20handling`
f `https://en.wikipedia.org/wiki/event%20loop`
g `https://en.wikipedia.org/wiki/iterator`
h `https://en.wikipedia.org/wiki/lazy%20evaluation`
i `https://en.wikipedia.org/wiki/pipeline%20%28software%29`

Coroutines are components that can be created and manipulated with the coroutine library in Lua and that allow the yielding and resuming of the execution of a function at specific locations by calling functions that yield the coroutine from inside of itself or that resume it from outside of itself. Example:

1. A function in the main thread creates a coroutine from a function with `coroutine.create` and resumes it with `coroutine.resume`, to which the number 3 is passed.
2. The function in the coroutine executes and gets the number passed to `coroutine.resume` as an argument.
3. The function arrives at a certain point in its execution where it calls `coroutine.yield` with, as an argument, the sum of the argument it received (3) and 2 (hence, 3+2=5).
4. The call to `coroutine.resume` returns 5, because it was passed to `coroutine.yield`, and the main thread, now running again, stores that number in a variable. It resumes the coroutine again after having executed some code, passing to `coroutine.resume` the double of the value it has received from the call to `coroutine.resume` (i.e. it passes 5×2=10).
5. The coroutine gets the value passed to `coroutine.resume` as the result of the call to `coroutine.yield` and terminates after running some more code. It returns the difference between the result of the call to `coroutine.yield` and the value it was given as a parameter initially (i.e. 10−3=7).
6. The main thread gets the value returned by the coroutine as the result of the call to `coroutine.resume` and goes on.

This example, put in code, gives the following:

```lua
local co = coroutine.create(function(initial_value)
        local value_obtained = coroutine.yield(initial_value + 2) -- 3+2=5
        return value_obtained - initial_value -- 10-3=7
end)

local _, initial_result = coroutine.resume(co, 3) -- initial_result: 5
local _, final_result = coroutine.resume(co, initial_result * 2) -- 5*2=10
print(final_result) --> 7
```

The `coroutine.create` function creates a coroutine from a function; coroutines are values of type "thread". `coroutine.resume` starts or continues the execution of a coroutine. A coroutine is said to be dead when it has encountered an error or has nothing left to run (in which case it has terminated its execution). When a coroutine is dead, it cannot be resumed. The `coroutine.resume` function will return `true` if the execution of the coroutine was successful, along with all the values returned, if the coroutine has terminated, or passed to `coroutine.yield` if it has not. If the execution was not successful, it will return `false` along with an error message. `coroutine.resume` returns the running coroutine and `true` when that coroutine is the main thread, or `false` otherwise.

The `coroutine.status` function returns the status of a coroutine as a string:

- "running" if the coroutine is running, which means it must be the coroutine which called `coroutine.status`
- "suspended" if the coroutine is suspended in a call to yield or if it has not started running yet
- "normal" if the coroutine is active but not running, which means it has resumed another coroutine
- "dead" if the coroutine has finished running or has encountered an error

The `coroutine.wrap` function returns a function that resumes a coroutine every time it is called. Extra arguments given to this function will act as extra arguments to `coroutine.resume` and values returned by the coroutine or passed to `coroutine.yield` will be returned by a call to the function. The `coroutine.wrap` function, unlike `coroutine.resume`, does not call the coroutine in protected mode and propagates errors.

There are many use cases for coroutines, but describing them are outside the scope of this book.

5.3 String matching

When manipulating strings, it is frequently useful to be able to search strings for substrings that follow a certain pattern. Lua has a string manipulation library that offers functions for doing this and a notation for expressing patterns that the functions can search for in strings. The notation offered by Lua is very similar to regular expression[4]s, a notation for expressing patterns used by most languages and tools in the programming world. However, it is more limited and has slightly different syntax.

The `find` function of the string library looks for the first match of a pattern in a string. If it finds an occurrence of the pattern in the string, it returns the indices in the string (integers representing the position of characters in the string starting from the first character, which is at position 1) where the occurrence starts and ends. If it doesn't find an occurrence of the pattern, it returns nothing. The first parameter it accepts is the string, the second being the pattern and the third being an integer indicating the character position where the `find` function should start searching. Finally, the `find` function can be told to perform a simple match without patterns by being given the value `true` as its fourth argument. It

4 https://en.wikipedia.org/wiki/regular%20expression

will then simply search for an occurrence of the second string it is given in the first string. The third argument must be given when a simple match is performed. This example code searches for the word "lazy" in a sentence and prints the start and end positions of the occurrence it finds of the word:

```
local start_position, end_position = string.find("The quick brown fox jumps over
 the lazy dog.", "lazy", 1, true)
print("The word \"lazy\" was found starting at position " .. start_position .. "
 and ending at position " .. end_position .. ".")
```

This code gives the result `The word "lazy" was found starting at position 36 and ending at position 39.`. It is equivalent to the following:

```
local sentence = "The quick brown fox jumps over the lazy dog."
local start_position, end_position = sentence:find("lazy", 1, true)
print("The word \"lazy\" was found starting at position " .. start_position .. "
 and ending at position " .. end_position .. ".")
```

This works because the `index` metamethod of strings is set to the table containing the functions of the string library, making it possible to replace `string.a(b, ...)` by `b:a(...)`.

Functions in the string library that accept indices to indicate character position or that return such indices consider the first character as being at position 1. They accept negative numbers and interpret them as indexing backwards, from the end of the string, with the last character being at position -1.

Patterns are strings that follow a certain notation to indicate a pattern that a string may match or not. For this purpose, patterns contain character classes, combinations that represent sets of characters.

Character combination	Description
.	All characters
%a	Letters (uppercase and lowercase)
%c	Control characters
%d	Digits
%g	Printable characters (except the space character)
%l	Lowercase letters
%p	Punctuation characters
%s	Space characters
%u	Uppercase letters
%w	Alphanumeric characters (digits and letters)
%x	Hexadecimal digits

All characters that are not special represent themselves and special characters (all characters that are not alphanumeric) can be escaped by being prefixed by a percentage sign. Character classes can be combined to create bigger character classes by being put in a set. Sets are noted as character classes noted between square brackets (i.e. `[%xp]` is the set of all hexadecimal characters plus the letter "p"). Ranges of characters can be noted by separating the end characters of the range, in ascending order, with a hyphen (i.e. `[0-9%s]` represents all the digits from 0 to 9 plus space characters). If the caret ("^") character is put at the

start of the set (right after the opening square bracket), the set will contain all characters except those it would have contained if that caret had not been put at the start of the set.

The complement of all classes represented by a letter preceded of a percentage sign can be noted as a percentage sign followed by the corresponding uppercase letter (i.e. `%S` represents all characters except space characters).

Patterns are sequences of pattern items that represent what sequences should be found in the pattern for a string to match it. A pattern item can be a character class, in which case it matches any of the characters in the class once, a character class followed by the "*" character, in which case it matches 0 or more repetitions of characters in the class (these repetition items will always match the longest possible sequence), a character class followed by the addition ("+") character, in which case it matches 1 or more repetitions of characters in the class (these repetition items will also always match the longest possible sequence), a character class followed by the minus ("-") character, in which case it matches 0 or more repetitions of characters in the class, but matches the shortest possible sequence or a character class followed by an interrogation mark, in which case it matches one or no occurrence of a character in the class.

It is possible to match substrings equivalent to previously captured substrings: `%1` will match the first captured substring, `%2` the second, and so on until `%9`. Captures are described below. There are two other functionalities offered by patterns, as described by the reference manual:

> `%bxy`, where *x* and *y* are two distinct characters; such item matches strings that start with *x*, end with *y*, and where the *x* and *y* are balanced. This means that, if one reads the string from left to right, counting +1 for an *x* and -1 for a *y*, the ending *y* is the first *y* where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.
> `%f[set]`, a frontier pattern; such item matches an empty string at any position such that the next character belongs to set and the previous character does not belong to set. The set *set* is interpreted as previously described. The beginning and the end of the subject are handled as if they were the character '\0'.

Patterns are sequences of pattern items, optionally preceded by a caret, which indicates that the pattern can only match at the beginning of the string, and optionally followed by a dollar sign, which indicates that the pattern can only match at the end of the string. These symbols are said to *anchor* the match at the beginning or the end of the string. These two characters only have a special meaning when at the beginning or at the end of a pattern.

Sub-patterns can be enclosed inside parentheses inside patterns to indicate captures. When a match succeeds, the substrings of the string that match captures are stored for future use, for example to be returned by `gmatch`. They are always numbered starting from the position of their left parenthesis. Two empty parentheses denote the *empty capture*, which captures the current string position (which is a number and is not a part of the string).

The `gmatch` function can be used to iterate through the occurrences of a pattern in a string; it is not possible, unlike with the `find` function, to specify an initial position to start searching or to perform simple matching. The `gmatch` function returns an iterator that, when called, returns the next captures from the given pattern in the string. The

whole match is given instead if there are no captures specified in the pattern. The following example shows how to iterate through the words in a sentence and print them one by one:

```
local sentence = "The quick brown fox jumps over the lazy dog."
for word in sentence:gmatch('%a+') do
	print(word)
end
```

In this example, the entire match is given by the only value returned by the iterator, *word*.

The `gsub` function can be used to replace all occurrences of a pattern in a string by something else. Its first two arguments are the string and the pattern, while the third is the string to replace occurrences by and the fourth is the maximum number of occurrences that should be replaced. The third argument, instead of being a string, can also be a table or a function.

When the third argument is a string, it is called the replacement string and it replaces occurrences of the pattern in the string. Captures stored by the pattern can be embedded in the replacement string; they are noted by a percentage sign followed by a digit representing the number of the capture. The match itself can be represented by `%0`. Percentage signs in replacement strings must be escaped as `%%`.

When the third argument is a table, the first capture is used as a key to index that table and the replacement string is the value corresponding to that key in the table. When it is a function, that function is called for every match, with all captures passed as arguments. In both cases, if there is no capture, the entire match is used instead. If the function or table gives the value `false` or `nil`, no replacement is done.

Here are some examples taken directly from the Lua 5.2 Reference Manual:

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return load(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.2"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.2.tar.gz"
```

Lua offers other functions for manipulating strings than those for pattern matching. These include the `reverse` function, which returns a string with the order of the characters reversed, the `lower` function, which returns the lowercase equivalent of a string, the `upper` function, which returns the uppercase equivalent of a string, the `len` function, which returns the length of a string and the `sub` function, which returns the substring of a string that starts at and ends at the two character positions given as arguments. There are more, and their documentation can be found in the Lua 5.2 Reference Manual.

6 Appendix:Software testing

The term **software testing** refers to a number of methods and processes that are used to discover bugs and programming mistakes in computer software. Software testing can be done statically, in which case in is called static testing and is done without executing the computer software, or dynamically, in which case it is called dynamic testing and is done while the computer program that is being tested is running.

6.1 Type checking

> In programming languages, a **type system** is a collection of rules that assign a property called a *type*[a] to the various constructs—such as variable[b]s, expression[c]s, function[d]s or modules[e]—a computer program[f] is composed of. The main purpose of a type system is to reduce bug[g]s in computer programs by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. This checking can happen statically (at compile time[h]), dynamically (at run time[i]), or it can happen as a combination of static and dynamic checking. Type systems have other purposes as well, such as enabling certain compiler optimizations, allowing for multiple dispatch[j], providing a form of documentation, etc.

a `https://en.wikipedia.org/wiki/type%20%28computer%20science%29`
b `https://en.wikipedia.org/wiki/variable%20%28computer%20science%29`
c `https://en.wikipedia.org/wiki/expression%20%28computer%20science%29`
d `https://en.wikipedia.org/wiki/function%20%28computer%20science%29`
e `https://en.wikipedia.org/wiki/modular%20programming`
f `https://en.wikipedia.org/wiki/computer%20program`
g `https://en.wikipedia.org/wiki/bug%20%28computer%20programming%29`
h `https://en.wikipedia.org/wiki/compile%20time`
i `https://en.wikipedia.org/wiki/run%20time%20%28program%20lifecycle%20phase%29`
j `https://en.wikipedia.org/wiki/multiple%20dispatch`

Type-checking can be done, as the extract from Wikipedia brilliantly said, at run time or at compile time. If it is done at compile time, the compiler, when compiling source code, will verify the type safety of the program and guarantee that the program satisfies certain type safety properties—generally, static type-checkers will simply verify that variables always have values of the same type and that arguments passed to functions will have the right type.

The static approach allows bugs to be discovered early in the development cycle. The dynamic approach, in contrast, consists in verifying that the program follows the type constraints when it is running. While this means that dynamic type-checkers should be able to verify more constraints, most dynamically typed languages do not have many type

constraints. Lua is a dynamically typed language: in Lua, values have types, but variables do not. This means that the value of a variable can be a number at some point of the program's execution and be a string at another point.

Lua's type system is very simple in comparison with most other languages. It performs type checking when operators are used (attempting to add two values of which at least one is not a number and cannot be coerced to one, for example, will raise a type error) and when functions of the standard libraries are called (functions of the standard library reject arguments that do not have the right type and raise an appropriate error).

Since Lua does not have functionality for specifying a type for function parameters, the `type` function can be useful to verify that arguments passed to functions are of the appropriate type. This is most useful for functions that will be passed arguments provided by users while a program is running (for example, in an interactive environment for calling predefined Lua functions), since adding code for type checking to functions makes them more verbose and adds maintenance overhead.

6.2 White-box testing

The term *white-box testing* refers to the practice of using knowledge of the internal workings of software to create test cases to verify its functionality. It is relevant at three levels of software testing, but the one most interesting for Lua programs is the unit level, since Lua programs are usually part of a bigger application where the integration and system testing would take place.

There are multiple frameworks available for unit testing in Lua. Testing at the unit level is most appropriate for libraries, since it generally consists in writing test cases that pass specific arguments to functions and provide a warning when a function returns an unexpected value. This requires writing test cases for new functionality, but has the benefit of making errors introduced in code easier to notice when they modify the behavior of functions in a way that makes the tests not pass anymore.

There are multiple unit testing frameworks for Lua. One of them, busted, supports the standard Lua virtual machine as well as LuaJIT, and can also be used with MoonScript and Terra, the former a language that compiles to Lua and the latter a low-level language that is interoperable with Lua. Another unit testing framework for Lua, Luaunit, is written entirely in Lua and has no dependencies. Shake is a simpler test framework, initially part of the Kepler Project, that uses the `assert` and `print` functions but is no longer actively developed.

6.3 Further reading

The lua-users wiki, an excellent resource to find information about Lua, provides the following material that is related to software testing. Some of these pages consist in links to other pages or to projects that can be useful for various tasks.

- Lua Type Checking[1]
- Unit Testing[2]
- Debugging Lua Code[3]
- Program Analysis[4]
- Debugging and Testing[5]

1 http://lua-users.org/wiki/LuaTypeChecking
2 http://lua-users.org/wiki/UnitTesting
3 http://lua-users.org/wiki/DebuggingLuaCode
4 http://lua-users.org/wiki/ProgramAnalysis
5 http://lua-users.org/wiki/DebuggingAndTesting

7 Glossary

This is a glossary that contains terms related to programming in the context of Lua. Its use is recommended to find the meaning of words that are not understood.

abstract class

An abstract class[1] is a class of which instances cannot be created directly. Abstract classes are abstract types[2].

abstract data type

An abstract data type[3] is a model to represent a class of data structures[4] that have similar behavior. Abstract data types are defined by the operations that can be performed on them and by mathematical constraints of these operations rather than by the implementation and the way the data is stored in the memory of the computer.

abstract type

An abstract type[5] is a type of data of which instances cannot be created directly.

actual parameter

See argument[6].

additive inverse

The additive inverse[7] of a number is the number that, when added to that number, yields zero. For example, the additive inverse of 42 is -42.

arithmetic negation

Arithmetic negation is the operation that produces the additive inverse[8] of a number.

arithmetic operation

An arithmetic operation[9] is an operation whose operands are numbers.

arity

1 `https://en.wikipedia.org/wiki/concrete%20class%23Abstract%20and%20concrete`
2 Chapter 8 on page 53
3 `https://en.wikipedia.org/wiki/abstract%20data%20type`
4 Chapter 8 on page 53
5 `https://en.wikipedia.org/wiki/abstract%20type`
6 Chapter 8 on page 53
7 `https://en.wikipedia.org/wiki/additive%20inverse`
8 Chapter 8 on page 53
9 `https://en.wikipedia.org/wiki/arithmetic%20operation`

The arity[10] of an operation or of a function is the number of operands or arguments the operation or function accepts.

argument

An argument[11] is a value passed to a function[12].

array

An array[13] is a data structure[14] consisting of a collection of values, each identified by at least one array index or key.

associative array

An associative array[15] is an abstract data type[16] composed of a collection of pairs of keys and values, such that each possible key appears at most once in the collection.

augmented assignment

Augmented assignment[17] is a type of assignment that gives a variable a value that is relative to its prior value.

binary operation

A binary operation[18] is an operation of which the arity is two.

boolean

See logical data[19].

boolean negation

See logical negation[20].

chained assignment

Chained assignment[21] is a type of assignment that gives a single value to many variables. Example: `a = b = c = d = 0`.

chunk

A chunk is a sequence of statements.

compound assignment

See augmented assignment[22].

10 `https://en.wikipedia.org/wiki/arity`
11 `https://en.wikipedia.org/wiki/parameter%20%28computer%20programming%29`
12 Chapter 8 on page 53
13 `https://en.wikipedia.org/wiki/array%20data%20structure`
14 Chapter 8 on page 53
15 `https://en.wikipedia.org/wiki/associative%20array`
16 Chapter 8 on page 53
17 `https://en.wikipedia.org/wiki/Augmented%20assignment`
18 `https://en.wikipedia.org/wiki/binary%20operation`
19 Chapter 8 on page 53
20 Chapter 8 on page 53
21 `https://en.wikipedia.org/wiki/Chained%20assignment`
22 Chapter 8 on page 53

concatenation

String concatenation[23] is the operation of joining two strings of characters. For example, the concatenation of ”snow” and ”ball” is ”snowball”.

concrete class

A concrete class is a class of which instances can be created directly. Concrete classes are concrete types[24].

concrete type

A concrete type is a type of which instances can be created directly.

condition

A condition is a predicate[25] that is used in a conditional statement[26] or as an operand to the conditional operator[27]. Conditions, in Lua, are considered as true when their expression evaluates to a value other than `nil` or `false`, and are considered as false otherwise.

conditional operator

A conditional operator[28] is an operator that returns a value if a condition[29] is true and another if it isn't.

conditional statement

A conditional statement[30] is a statement that executes a piece of code if a condition[31] is true.

data structure

A data structure[32] is a particular way of storing and organizing data in the memory of a computer. It is the implementation of an abstract data type[33].

data type

A data type[34] is a model for representing the storage of data in the memory of a computer.

dictionary

See associative array[35].

exclusive disjunction

---

23 `https://en.wikipedia.org/wiki/String%20concatenation`
24 Chapter 8 on page 53
25 Chapter 8 on page 53
26 Chapter 8 on page 53
27 Chapter 8 on page 53
28 `https://en.wikipedia.org/wiki/conditional%20operator`
29 Chapter 8 on page 53
30 `https://en.wikipedia.org/wiki/conditional%20%28computer%20programming%29`
31 Chapter 8 on page 53
32 `https://en.wikipedia.org/wiki/data%20structure`
33 Chapter 8 on page 53
34 `https://en.wikipedia.org/wiki/data%20type`
35 Chapter 8 on page 53

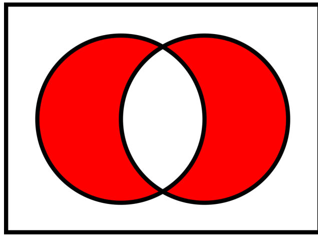

Figure 4 Venn diagram[a] of $a \veebar b$

a https://en.wikipedia.org/wiki/Venn%20diagram

The exclusive disjunction[36] operation is a binary operation[37] that produces the value `true` when one of its operands is true but the other is not. The exclusive disjunction of a and b is expressed mathematically as $a \veebar b$. There is no operator corresponding to exclusive disjunction in Lua, but $a \veebar b$ can be represented as `(a or b) and not (a and b)`.

formal parameter

See parameter[38].

function

A function is a sequence of statements (instructions) that perform a specific task. Functions can be used in a program wherever that particular task needs to be performed. Functions are usually defined in the program that will use them, but are sometimes defined in libraries that can be used by other programs.

hash map

See hash table[39].

hash table

A hash table[40] is an implementation as a data structure[41] of the associative array[42]. A hash table uses a hash function[43] to compute an index into an array of buckets or slots, from which the value corresponding to the index can be found.

inline if

See conditional operator[44].

integer

36 https://en.wikipedia.org/wiki/exclusive%20or
37 Chapter 8 on page 53
38 Chapter 8 on page 53
39 Chapter 8 on page 53
40 https://en.wikipedia.org/wiki/hash%20table
41 Chapter 8 on page 53
42 Chapter 8 on page 53
43 https://en.wikipedia.org/wiki/hash%20function
44 Chapter 8 on page 53

An integer[45] is a number that can be written without a fractional or decimal component. Integers are implemented in Lua in the same way as other numbers.

length operation

The length operation is the operation that produces the number of values in an array.

literal

A literal is a notation for representing a fixed value in source code. All values can be represented as literals in Lua except threads and userdata.

logical complement

The logical complement[46] of a boolean value is the boolean value that is not that value. This means the logical complement of `true` is `false` and vice versa.

logical conjunction

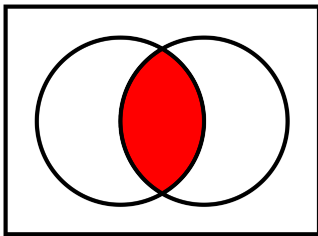

Figure 5 Venn diagram of $a \wedge b$

The logical conjunction[47] operation is a binary operation[48] that produces the value `true` when both of its operands are true and `false` in all other cases. It is implemented as the `and` operator in Lua and it returns its first operand if it is `false` or `nil` and the second operand otherwise. The logical conjunction of a and b is expressed mathematically as $a \wedge b$.

logical data

The logical data type[49], which is generally called the boolean type, is the type of the values `false` and `true`.

logical disjunction

45 `https://en.wikipedia.org/wiki/integer`
46 `https://en.wikipedia.org/wiki/logical%20complement`
47 `https://en.wikipedia.org/wiki/logical%20conjunction`
48 Chapter 8 on page 53
49 `https://en.wikipedia.org/wiki/logical%20data%20type`

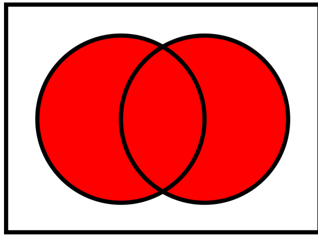

Figure 6 Venn diagram of $a \vee b$

The logical disjunction[50] operation is a binary operation[51] that produces the value `false` when both of its operands are false and `true` in all other cases. It is implemented as the `or` operator in Lua and it returns the first operand if it is neither `false` nor `nil` and the second otherwise. The logical disjunction of a and b is expressed mathematically as $a \vee b$.

logical negation

Logical negation[52], implemented in Lua by the `not` operator, is the operation that produces the logical complement[53] of a boolean value.

map

See associative array[54].

method

A method[55] is a function that is a member of an object and generally operates on that object.

modulo

See modulo operation[56].

modulo operation

The modulo operation[57], implemented in Lua by the `%` operator, is the operation that produces the remainder of the division of a number by another.

modulus

See modulo operation[58].

multiple assignment

50 https://en.wikipedia.org/wiki/logical%20disjunction
51 Chapter 8 on page 53
52 https://en.wikipedia.org/wiki/Logical%20negation
53 Chapter 8 on page 53
54 Chapter 8 on page 53
55 https://en.wikipedia.org/wiki/method%20%28computer%20programming%29
56 Chapter 8 on page 53
57 https://en.wikipedia.org/wiki/modulo%20operation
58 Chapter 8 on page 53

See parallel assignment[59].

nil

The type nil is the type of the value `nil`, whose main property is to be different from any other value; it usually represents the absence of a useful value.

not operator

See logical negation[60].

number

The number type represents real (double-precision floating-point[61]) numbers. It is possible to build Lua interpreters that use other internal representations for numbers, such as single-precision float or long integers.

operator

An operator[62] is a token that generates a value from one or many operands.

parallel assignment

Parallel assignment[63] is a type of assignment that simultaneously assigns values to different variables.

parameter

A parameter[64] is a variable in a function definition to which the argument that corresponds to it in a call to that function is assigned.

predicate

A predicate is an expression that evaluates to a piece of logical data[65].

procedure

See function[66].

relational operator

A relational operator[67] is an operator that is used to compare values.

routine

See function[68].

sign change

59 Chapter 8 on page 53
60 Chapter 8 on page 53
61 `https://en.wikipedia.org/wiki/double-precision%20floating-point`
62 `https://en.wikipedia.org/wiki/operator%20%28computer%20programming%29`
63 `https://en.wikipedia.org/wiki/Parallel%20assignment`
64 `https://en.wikipedia.org/wiki/parameter%20%28computer%20programming%29`
65 Chapter 8 on page 53
66 Chapter 8 on page 53
67 `https://en.wikipedia.org/wiki/relational%20operator`
68 Chapter 8 on page 53

See arithmetic negation[69].

simultaneous assignment

See parallel assignment[70].

string

The type string represents arrays of characters. Lua is 8-bit clean: strings can contain any 8-bit character, including embedded zeros.

string literal

A string literal[71] is the representation of a string value within the source code of a computer program. With respect to syntax, a string literal is an expression that evaluates to a string.

subprogram

See function[72].

subroutine

See function[73].

symbol

See token[74].

symbol table

A symbol table[75] is an implementation as a data structure[76] of the associative array[77]. They are commonly implemented as hash tables[78].

token

A token is an atomic piece of data, such as a word in a human language or such as a keyword in a programming language, for which a meaning may be inferred during parsing.

variable

A variable[79] is a label associated to a location in the memory. The data in that location can be changed and the variable will point to the new data.

variadic function

A variadic function[80] is a function of indefinite arity.

69 Chapter 8 on page 53
70 Chapter 8 on page 53
71 `https://en.wikipedia.org/wiki/string%20literal`
72 Chapter 8 on page 53
73 Chapter 8 on page 53
74 Chapter 8 on page 53
75 `https://en.wikipedia.org/wiki/symbol%20table`
76 Chapter 8 on page 53
77 Chapter 8 on page 53
78 Chapter 8 on page 53
79 `https://en.wikipedia.org/wiki/variable%20%28computer%20science%29`
80 `https://en.wikipedia.org/wiki/variadic%20function`

8 Index

This is an alphabetical index of the book.

8.1 A

- Allocation[1]
- Arithmetic operations[2]
- Arguments[3]
- Arrays[4]
- Assertion[5]
- Assignment[6]
- Associative arrays[7]
- Augmented assignment[8]
- Automatic memory management[9]

8.2 B

- Binary operators[10]
- Bit shifts[11]
- Bitwise AND[12]
- Bitwise NOT[13]
- Bitwise operations[14]
- Bitwise OR[15]
- Bitwise XOR[16]

1 Chapter 5.1.2 on page 33
2 Chapter 2.1.1 on page 8
3 Chapter 4 on page 27

4 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables

5 Chapter 5.1.1 on page 33
6 Chapter 3.1 on page 17

7 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables

8 Chapter 3.1.3 on page 19
9 Chapter 5.1.2 on page 33
10 Chapter 2.1 on page 7
11 Chapter 2.4 on page 12
12 Chapter 2.4 on page 12
13 Chapter 2.4 on page 12
14 Chapter 2.4 on page 12
15 Chapter 2.4 on page 12
16 Chapter 2.4 on page 12

- Blocks[17]
- Booleans[18]
- `break` statement[19]
- busted[20]
- Bytecode[21]

8.3 C

- Captures[22]
- Chained assignment[23]
- Character classes[24]
- Character ranges[25]
- Character sets[26]
- Chunks[27]
- Code annotations[28]
- Coercion[29]
- Collaborative multithreading[30]
- Comments[31]
- Compound assignment[32]
- Concatenation[33]
- Conditional statement[34]
- Condition-controlled loops[35]
- Coroutines[36]
- Count-controlled loops[37]

17 Chapter 3.4 on page 23
18 Chapter 2.1.3 on page 9
19 Chapter 3.3.2 on page 22
20 Chapter 6.2 on page 42
21 Chapter 3.4.1 on page 24
22 Chapter 5.3 on page 36
23 Chapter 3.1.3 on page 19
24 Chapter 5.3 on page 36
25 Chapter 5.3 on page 36
26 Chapter 5.3 on page 36
27 Chapter 3.4.1 on page 24
28 Chapter 1 on page 3
29 Chapter 2.3 on page 12
30 Chapter 5.2 on page 35
31 Chapter 1 on page 3
32 Chapter 3.1.3 on page 19
33 Chapter 2.1.4 on page 11
34 Chapter 3.2 on page 20
35 Chapter 3.3.1 on page 21
36 Chapter 5.2 on page 35
37 Chapter 3.3.2 on page 22

8.4 D

- Debugging[38]
- Decimal precision errors[39]
- Deprecation[40]
- Dictionaries[41]
- `do` statement[42]
- Dynamic testing[43]

8.5 E

- `else` block[44]
- `elseif` block[45]
- Empty captures[46]
- Empty statements[47]
- Errors[48]
- Expressions[49]
- Extended Backus–Naur form[50]

8.6 F

- Fields[51]
- Float division[52]
- Floats[53]
- Floor division[54]
- Foreach loop[55]
- Frontier patterns[56]

38 Chapter 4.2 on page 29
39 Chapter 3.3.2 on page 22
40 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Foreach%20loop
41 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
42 Chapter 3.1.2 on page 18
43 Chapter 6 on page 41
44 Chapter 3.2 on page 20
45 Chapter 3.2 on page 20
46 Chapter 5.3 on page 36
47 Chapter 3.4 on page 23
48 Chapter 4.2 on page 29
49 Chapter 2 on page 7
50 Chapter 1 on page 3
51 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
52 Chapter 2.1.1 on page 8
53 Chapter 2.1.1 on page 8
54 Chapter 2.1.1 on page 8
55 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Foreach%20loop
56 Chapter 5.3 on page 36

- Functions[57]

8.7 G

- Garbage collection[58]
- Generic `for` loop[59]

8.8 H

- Hash maps[60]
- Hash tables[61]
- Hello, world![62]

8.9 I

- Incrementation[63]
- Identifiers[64]
- `if` statement[65]
- Index[66]
- Infinite loops[67]
- Integers[68]
- Integration testing[69]
- Iterators[70]

57 Chapter 4 on page 27
58 Chapter 5.1.2 on page 33
59 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Foreach%20loop
60 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
61 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
62 Chapter 1 on page 3
63 Chapter 3.3.2 on page 22
64 Chapter 3.1.1 on page 17
65 Chapter 3.2 on page 20
66 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
67 Chapter 3.3 on page 21
68 Chapter 2.1.1 on page 8
69 Chapter 6.2 on page 42
70 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Iterators

8.10 J

8.11 K

- Kepler Project[71]
- Key[72]

8.12 L

- Left shift[73]
- Literals[74]
- Local variables[75]
- Logical errors[76]
- Long brackets[77]
- Long comments[78]
- Loops[79]
- Loop variables[80]
- luac[81]
- LuaJIT[82]
- Luaunit[83]

8.13 M

- Main thread[84]
- Maps[85]
- Metalanguage[86]
- Metamethods[87]
- Metatables[88]

71 Chapter 6.2 on page 42
72 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
73 Chapter 2.4 on page 12
74 Chapter 2.2 on page 11
75 Chapter 3.1.2 on page 18
76 Chapter 4.2 on page 29
77 Chapter 1 on page 3
78 Chapter 1 on page 3
79 Chapter 3.3 on page 21
80 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Creating%20iterators
81 Chapter 3.4.1 on page 24
82 Chapter 6.2 on page 42
83 Chapter 6.2 on page 42
84 Chapter 3.1.2 on page 18
85 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
86 Chapter 1 on page 3
87 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Metatables
88 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Metatables

- Methods[89]
- Modulo operation[90]
- MoonScript[91]
- Multiple assignment[92]

8.14 N

- Names[93]
- Nil[94]
- Numbers[95]
- Numeric `for` loops[96]

8.15 O

- Operator precedence[97]
- Operators[98]

8.16 P

- Parallel assignment[99]
- Patterns[100]
- Pattern items[101]
- Parameters[102]
- Precedence[103]
- Protected calls[104]

89 `https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Methods`
90 Chapter 2.1.1 on page 8
91 Chapter 6.2 on page 42
92 Chapter 3.1.3 on page 20
93 Chapter 3.1.1 on page 17
94 Chapter 2.1.2 on page 9
95 Chapter 2.1.1 on page 7
96 Chapter 3.3.2 on page 22
97 Chapter 2.5 on page 14
98 Chapter 2.1 on page 7
99 Chapter 3.1.3 on page 20
100 Chapter 5.3 on page 36
101 Chapter 5.3 on page 36
102 Chapter 4 on page 27
103 Chapter 2.5 on page 14
104 Chapter 4.2.1 on page 30

8.17 Q

8.18 R

- Registers[105]
- Regular expressions[106]
- Relational operators[107]
- Replacement strings[108]
- Return statement[109]
- Right shift[110]

8.19 S

- Scope[111]
- *self* parameter[112]
- Semantic errors[113]
- Shake[114]
- Short comments[115]
- Simultaneous assignment[116]
- Software testing[117]
- Sorting tables[118]
- Stack[119]
- Stack overflow[120]
- Statements[121]
- State values[122]
- Static testing[123]
- Strings[124]

105 Chapter 3.1.2 on page 18
106 Chapter 5.3 on page 36
107 Chapter 3.2 on page 20
108 Chapter 5.3 on page 36
109 Chapter 4.1 on page 29
110 Chapter 2.4 on page 12
111 Chapter 3.1.2 on page 18
112 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Methods
113 Chapter 4.2 on page 29
114 Chapter 6.2 on page 42
115 Chapter 1 on page 3
116 Chapter 3.1.3 on page 20
117 Chapter 6 on page 41
118 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Sorting
119 Chapter 4 on page 27
120 Chapter 4.3 on page 31
121 Chapter 3 on page 17
122 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Creating%20iterators
123 Chapter 6 on page 41
124 Chapter 2.1.4 on page 9

- String concatenation[125]
- String manipulation[126]
- String matching[127]
- String patterns[128]
- Symbol tables[129]
- Syntactic errors[130]
- Syntax[131]
- System testing[132]

8.20 T

- Type constraints[133]
- Table constructors[134]
- Tables[135]
- Terra[136]
- Test cases[137]
- Transformation function[138]
- Tuples[139]
- Type checking[140]
- Types[141]
- Type safety[142]
- Type system[143]

8.21 U

- Unary operators[144]
- Unit testing[145]

---

125 Chapter 2.1.4 on page 11
126 Chapter 5.3 on page 36
127 Chapter 5.3 on page 36
128 Chapter 5.3 on page 36
129 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
130 Chapter 4.2 on page 29
131 Chapter 1 on page 3
132 Chapter 6.2 on page 42
133 Chapter 6.1 on page 41
134 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
135 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables
136 Chapter 6.2 on page 42
137 Chapter 6.2 on page 42
138 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Creating%20iterators
139 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Unpacking%20tables
140 Chapter 6.1 on page 41
141 Chapter 2.1 on page 7
142 Chapter 6.1 on page 41
143 Chapter 6.1 on page 41
144 Chapter 2.1 on page 7
145 Chapter 6.2 on page 42

- Unpacking tables[146]

8.22 V

- Vararg functions[147]
- Variadic functions[148]

8.23 W

- Weak references[149]
- Weak tables[150]
- White-box testing[151]

8.24 X

8.25 Y

8.26 Z

8.27 Lua API

There is a separate index for the functions and variables that are part of the Lua API. This index points to parts of the book where functions or variables in the API are mentioned.

8.27.1 Basic functions

- assert[152]
- collectgarbage[153]
- dofile[154]
- error[155]
- getmetatable[156]

146 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Unpacking%20tables
147 Chapter 4.4 on page 32
148 Chapter 4.4 on page 32
149 Chapter 5.1.2 on page 34
150 Chapter 5.1.2 on page 34
151 Chapter 6.2 on page 42
152 Chapter 5.1.1 on page 33
153 Chapter 5.1.2 on page 34
154 Chapter 3.4.1 on page 24
155 Chapter 4.2.1 on page 30
156 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Metatables

- ipairs[157]
- load[158]
- loadfile[159]
- next[160]
- pairs[161]
- pcall[162]
- print[163]
- rawequal[164]
- rawget[165]
- rawlen[166]
- rawset[167]
- select[168]
- setmetatable[169]
- tonumber[170]
- tostring[171]
- type[172]
- xpcall[173]

8.27.2 Coroutine manipulation

- coroutine.create[174]
- coroutine.resume[175]
- coroutine.running[176]
- coroutine.status[177]
- coroutine.wrap[178]
- coroutine.yield[179]

157 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Foreach%20loop
158 Chapter 3.4.1 on page 24
159 Chapter 3.4.1 on page 24
160 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Foreach%20loop
161 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Foreach%20loop
162 Chapter 4.2.1 on page 30
163 Chapter 1 on page 3
164 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Metatables
165 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Metatables
166 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Metatables
167 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Metatables
168 Chapter 4.4 on page 32
169 https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Metatables
170 Chapter 2.3 on page 12
171 Chapter 2.3 on page 12
172 Chapter 2.1 on page 7
173 Chapter 4.2.1 on page 30
174 Chapter 5.2 on page 35
175 Chapter 5.2 on page 35
176 Chapter 5.2 on page 35
177 Chapter 5.2 on page 35
178 Chapter 5.2 on page 35
179 Chapter 5.2 on page 35

8.27.3 String manipulation

- string.dump[180]
- string.find[181]
- string.gmatch[182]
- string.gsub[183]
- string.len[184]
- string.lower[185]
- string.reverse[186]
- string.sub[187]
- string.upper[188]

8.27.4 Table manipulation

- table.foreach[189]
- table.foreachi[190]
- table.sort[191]
- table.unpack[192]

180 Chapter 3.4.1 on page 24
181 Chapter 5.3 on page 36
182 Chapter 5.3 on page 36
183 Chapter 5.3 on page 36
184 Chapter 5.3 on page 36
185 Chapter 5.3 on page 36
186 Chapter 5.3 on page 36
187 Chapter 5.3 on page 36
188 Chapter 5.3 on page 36
189 `https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Foreach%20loop`
190 `https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Foreach%20loop`
191 `https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Sorting`
192 `https://en.wikibooks.org/wiki/Lua%20Programming%2FTables%23Unpacking%20tables`

9 Contributors

Edits	User
1	Adrignola[1]
14	Darklama[2]
4	Dirk Hünniger[3]
3	J36miles[4]
4	JackPotte[5]
1	Johnwayne1986[6]
2	Lmika[7]
5	Marbux[8]
287	Mark Otaris[9]
6	Markhobley[10]
1	Maths314[11]
2	Pi zero[12]
1	QuiteUnusual[13]
3	Recent Runes[14]
1	SoulSlayer55[15]

1 https://en.wikibooks.org/wiki/User:Adrignola
2 https://en.wikibooks.org/wiki/User:Darklama
3 https://en.wikibooks.org/wiki/User:Dirk_H%25C3%25BCnniger
4 https://en.wikibooks.org/wiki/User:J36miles
5 https://en.wikibooks.org/wiki/User:JackPotte
6 https://en.wikibooks.org/w/index.php%3ftitle=User:Johnwayne1986&action=edit&redlink=1
7 https://en.wikibooks.org/w/index.php%3ftitle=User:Lmika&action=edit&redlink=1
8 https://en.wikibooks.org/w/index.php%3ftitle=User:Marbux&action=edit&redlink=1
9 https://en.wikibooks.org/wiki/User:Mark_Otaris
10 https://en.wikibooks.org/wiki/User:Markhobley
11 https://en.wikibooks.org/wiki/User:Maths314
12 https://en.wikibooks.org/wiki/User:Pi_zero
13 https://en.wikibooks.org/wiki/User:QuiteUnusual
14 https://en.wikibooks.org/wiki/User:Recent_Runes
15 https://en.wikibooks.org/w/index.php%3ftitle=User:SoulSlayer55&action=edit&redlink=1

List of Figures

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[16]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

16 Chapter 10 on page 71

1	en:User:Cburnett[17]	CC-BY-SA-3.0
2	en:User:Cburnett[18]	CC-BY-SA-3.0
3	User:Boivie[19], User:Boivie[20]	
4	CommonsDelinker, Herbythyme, Jarekt, JarektBot, SchlurcherBot, Stephan Kulla, Waldir, Watchduck	
5	CommonsDelinker, Filbot, JarektBot, SchlurcherBot, Tony Wills, Waldir, Watchduck	
6	CommonsDelinker, JarektBot, JurgenNL, SchlurcherBot, Waldir, Watchduck	

17 https://en.wikipedia.org/wiki/User:Cburnett
18 https://en.wikipedia.org/wiki/User:Cburnett
19 http://commons.wikimedia.org/wiki/User:Boivie
20 https://commons.wikimedia.org/wiki/User:Boivie

10 Licenses

10.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy

both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

10.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

10.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.