Mendel University in Brno

Algorithmization and Programming in Lua
Study text

Tomáš Hála
Mendel University in Brno

Project: Innovative Open Source Courses for Computer Science Curriculum

24. 6. 2022

Co-funded by the Erasmus+ Programme of the European Union
West Pomeranian University of Technology Szczecin
UNIVERSITY OF ŽILINA
Mendel University in Brno

Reviewer: Ing. Pavel Stříž, Ph.D.
Project: Innovative Open Source Courses for Computer Science Curriculum
© Mendel University in Brno, Zemědělská 1, 613 00 Brno, Czech Republic
ISBN 978-80-7509-893-1 (online ; pdf)
DOI https://doi.org/10.11118/978-80-7509-893-1

This material teaching was written as one of the outputs of the project "Innovative Open Source Courses for Computer Science Curriculum", funded by the Erasmus+ grant no. 2019-1-PL01-KA203-065564. The project is coordinated by West Pomeranian University of Technology in Szczecin (Poland) and is implemented in partnership with Mendel University in Brno (Czech Republic) and University of Žilina (Slovak Republic). The project implementation timeline is September 2019 to December 2022.

Project information

Project was implemented under the Erasmus+.
Project name: **"Innovative Open Source courses for Computer Science curriculum"**
Project nr: **2019-1-PL01-KA203-065564**
Key Action: **KA2 – Cooperation for innovation and the exchange of good practices**
Action Type: **KA203 – Strategic Partnerships for higher education**

Consortium

ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY W SZCZECINIE
MENDELOVA UNIVERZITA V BRNĚ
ŽILINSKÁ UNIVERZITA V ŽILINE

Erasmus+ Disclaimer

Copyright Notice

Co-funded by the
Erasmus+ Programme
of the European Union

Acknowledgments

I would like to thank Pavel Stříž for his valuable comments and suggestions for the future improvement of this text.

Contents

1 Introduction

The Lua programming language is not entirely new, but it has only become more widely known and widespread in the last 15 years, mainly because of its ease of use in computer game development.

What is Lua[1]?

It is a programming language created in 1993 by the group of authors – Roberto Ierusalimschy, Luiz Henrique de Figueiredo and Waldemar Celes, members of the Computer Graphics Technology Group Computer Graphics Group) at the Pontifical University of Rio de Janeiro, Brazil.

Their goal was to create a simple language with minimal system requirements, portable between different platforms and, most importantly, easily embedded into other applications.

These advantageous features, in addition to the aforementioned use in game development, have caused the Lua programming language *'to become widely used in all kinds of industrial applications, such as robotics, literate programming, distributed business, image processing, extensible text editors, Ethernet switches, bioinformatics, finite-element packages, web development, and more.'* (Ierusalimchy, de Figueiredo and Celes, [2007]).

How to learn a programming language? While there are manuals on the market dealing with this language, but they are not usually freely available. Internet resources, whether the lua.org website or other servers, offer resources more of a documentary nature, which in their scope or content will be appreciated by more experienced users, but cannot fully meet the needs of beginner users.

This textbook has therefore been written in an effort to provide this particular group of novice users an initial basic overview of Lua's capabilities, along with a more detailed explanation of the generally used terms and concepts.

The programming language Lua is available in many versions in various repositories. To try the examples in this textbook, you need version 5.3 or higher.

The text covers the Lua programming language, describing its basic features – data types, commands, structures, and appropriate procedures of their use. However, it should not be considered as a textbook only of programming language, since mere knowledge of the elements of a programming language does not effective use of the computer. Therefore, the emphasis in this textbook is on the correct application of the principles of algorithmization and on the knowledge and use of basic algorithms.

[1] The word *lua* is taken from Portuguese and means *moon*

The purpose of this book is to provide guidance on how to create useful, yet simple programs, especially for their own use, i.e. programs with easy control.

In the explanation in this text, it is assumed that the reader already has the basic experience with operating computer software, especially with operating system and the editor.

In the list of references, the reader will find a list of sources that can be recommended for further study of programming language Lua. From these sources, some examples have also been taken.

Colour notation:

```
Codes in Lua.
```

```
Incorrect codes in Lua which you should avoid.
```

```
Input or output data for the related programs
```

```
Bash commands or scripts
```

```
Source codes for ConTeXt
```

2 Algorithms and programming languages

Before one begins to create a program, it is not only necessary to be clear about the issues involved, but it is also advisable to be familiar with the terminology.

Therefore, this chapter will be devoted to an overview and explanation of terms that the user encounters when creating programs.

Algorithm is a procedure we can use to solve a given problem. Any good algorithm must satisfy certain properties, i.e. must be:

- unambiguous (deterministic),
- finite (resultative), i.e. always leading to certain results,
- general , i.e. applicable to the solution of a given problem using any admissible data,
- repeatable, i.e. always leading to the same results with the same input data.

In practice, it is also recommended that the algorithm be **comprehensible** and **clear**, allowing corrections and modifications to be made easily.

A number of different means are available for expressing the algorithm. Algorithm can thus be expressed:

- verbally – in natural language,
- graphically – flowchart or structure chart,
- mathematically – a relationship between quantities, a system of equations, matrices,
- programming language.

All methods of expressing an algorithm allow a person to perform a specified action (calculate, search, sort, etc.). Only one of them – the programming language – is comprehensible to a computer under certain circumstances.

In this text, we will prefer to describe the algorithm in words. Some algorithms will be expressed mathematically, since mathematical notation is generally accepted and familiar way of symbolic expression.

2.1 Algorithmization and programming

The activity leading to construction of an algorithm that satisfies the above mentioned properties is called **algorithmization**. Its input of algorithmization is a **problem** and the **algorithm** is the output. In algorithmization, we can find multiple procedures to solve one problem, ie. multiple algorithms.

If we have a good algorithm, we can do another activity, which is called **programming**. The input here is the algorithm and the output is a computer program. A **program is an algorithm expressed in a programming language**. In programming, we can write the chosen algorithm in various ways to get multiple programs that produce the same results.

This is a very important point: if you solve different problems in a collective, each of you may have more or less different solutions, and all of them may be correct. This is perfectly fine, because both algorithmization and programming are creative activities: they depends on intellect, skill, knowledge and sometimes on intuition.

2.2 Programming language and machine code

A programming language, however, is only used to write a program. In fact, the program must be translated into a language that it understands the computer, namely the processor, i.e. into **machine code**machine code. The conversion from programming language to machine code is called **compilation**.

If we work on a computer with a particular programming language, it means that there is a program on that computer that allows to compile the program we have created into machine code. Such a program is called **compiler**. The fact that the compiler is a program is very important for us. A program written in a programming language must be written very precisely and according to a set of rules of that programming language.

In addition, **interpreted programs** are also very popular. This means that they are executed line by line or command by command. Here the program as a not translated as a whole but partial sections of code are executed sequentially by a program called an **interpreter**.

For the Lua programming language, there are both interpreter programs `lua` and compiler `luac`. Both will be more explained in the next chapter.

3 Tools for creating programs in Lua

Program development tools may vary depending on the operating system. Besides the basic and simple command line work, there are various integrated development environments. There are also online tools available at the internet.

3.1 Linux command line

Run the program `lua` on the command line to interpret the code. If not present, install it using:

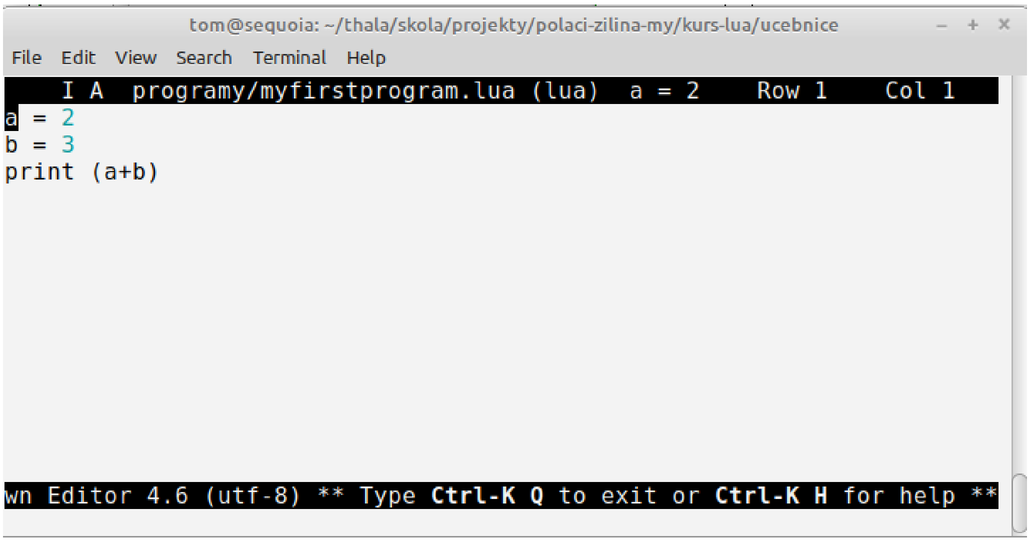
```
sudo apt install lua      # Ubuntu, Mint atd.
sudo yum install lua      # CentOS, Fedora atd.
```

```
Package lua is a virtual package provided by:
  lua5.3:i386 5.3.3-1ubuntu0.18.04.1
  lua5.3 5.3.3-1ubuntu0.18.04.1
  lua5.2:i386 5.2.4-1.1build1
  lua5.1:i386 5.1.5-8.1build2
  lua50 5.0.3-8
  lua5.2 5.2.4-1.1build1
  lua5.1 5.1.5-8.1build2
You should explicitly select one to install.
```

If the installer offers more than one version, we recommend you to use the latest one. After running `lua`, we will see:

```
your_os_prompt: lua
Lua 5.3.3  Copyright (C) 1994-2016 Lua.org, PUC-Rio
>
```

Now we can write commands after the prompt >:


tom@sequoia: ~/thala/skola/projekty/polaci-zilina-my/kurs-lua/ucebnice
File Edit View Search Terminal Help

```
    I A  programy/myfirstprogram.lua (lua)  a = 2    Row 1    Col 1
a = 2
b = 3
print (a+b)






wn Editor 4.6 (utf-8) ** Type Ctrl-K Q to exit or Ctrl-K H for help **
```

Figure 3.1 Our first program in Lua

```
your_prompt: lua
Lua 5.3.3  Copyright (C) 1994-2016 Lua.org, PUC-Rio
> a = 2
> b = 3
> print (a+b)
5
> =a+b
5
```

In addition to `print`, the most commonly used tool for displaying result, you can also use *shortcut* – the character `=`, which will also display the result of the expression. Note that the equal sign, instead of the `print`, cannot be used in batch processing.

We can exit the program correctly with the key `Ctrl-D` or interrupt the run with `Ctrl-C`.

3.1.1 Scripts

The above solution is not convenient for longer programs, because when repeatedly running it is necessary to write everything again.

That is why we prefer to use a text editor (see the Figure 3.1) to create a longer program. Then program save on a file can be run as a batch:

```
lua myfirstprogram.lua
5
```

3.1.2 Precompilation

An executable program is a file that stores machine code instructions. It is not common to write a program directly in machine code, so we use so-called compilers to create executable programs. Here we will be talking about the `luac` program. It should be remembered that with `luac` we do not create a standalone executable program, as is the case with many other programming languages, but only precompiled code that speed up the loading of code into memory.

```
luac myfirstprogram.lua
```

By looking in the directory (`ls -l`), we see that a new file has appeared:

```
-rw-rw-r-- 1 tom tom 219 aug 17 19:53 luac.out
-rw-rw-r-- 1 tom tom  24 aug 17 19:50 myfirstprogram.lua
```

The `luac.out` name is the default one and it is not obvious which one source code it refers to. It is more useful to specify the name of the output file explicitly:

```
luac myfirstprogram.lua -o myfirstprogram.out
```

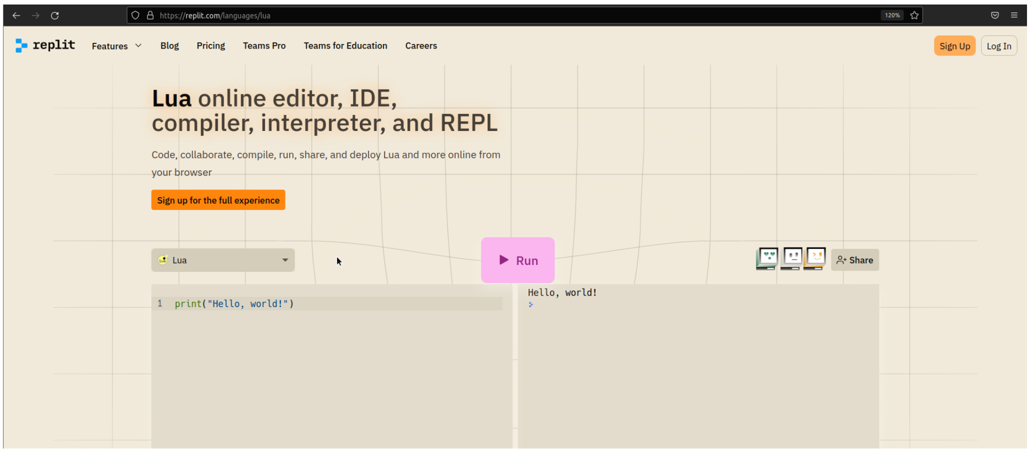
```
-rw-rw-r-- 1 tom tom 219 aug 17 19:53 luac.out
-rw-rw-r-- 1 tom tom  24 aug 17 19:50 myfirstprogram.lua
-rw-rw-r-- 1 tom tom 219 aug 17 19:54 myfirstprogram.out
```

For more information about `luac`, see eg. Lua.org (2020).

3.1.3 Online tools

In addition to the software installed on our own computer, we can use popular tools available online. The use of them is quite intuitive, so here is just an overview of the most well-known ones.

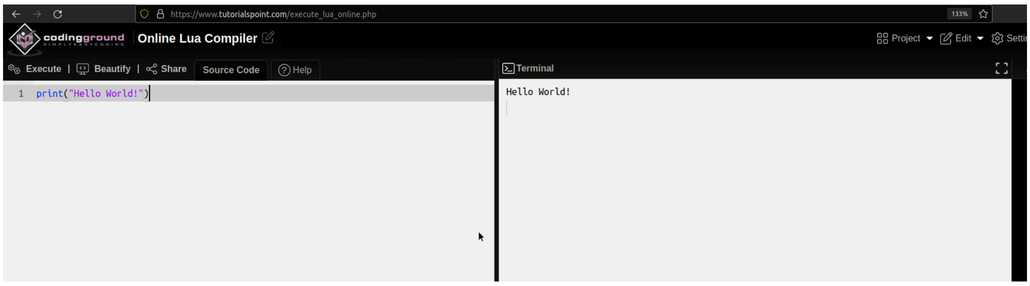
https://geekflare.com/online-compiler/lua
https://www.jdoodle.com/execute-lua-online/
https://onecompiler.com/lua/3y5j9aajb
https://replit.com/languages/lua
https://www.tutorialspoint.com/execute_lua_online.php
https://www.lua.org/demo.html


https://replit.com/languages/lua
replit
Features
Blog
Pricing
Teams Pro
Teams for Education
Careers
Sign Up
Log In
Lua online editor, IDE, compiler, interpreter, and REPL
Code, collaborate, compile, run, share, and deploy Lua and more online from your browser
Sign up for the full experience
Lua
Run
Share

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

```
Hello, world!
>
```

Figure 3.2 Replit: Online development environment



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

```
Hello World!
```

Figure 3.3 Development environment of Lua online

Questions

(1) What is a compiler?
(2) What is the difference between a compiler and interpret?
(3) Why is batch processing preferable to interactive work?

4 Elements of the programming language Lua

Let's first compare the languages that people speak. Each language has its own alphabet. The alphabets of some languages are similar (compare English with Slovak), others (e.g. Czech, Greek and Russian) are are different. The alphabet is made up of symbols, which we call letters.

Programming languages – even though they are artificially created and designed for machine processing – are in this respect living languages similar. Every programming language has its own 'alphabet', which is called **set of symbols**. The 'alphabet' of some programming languages are similar, others are are very different from each other.

The symbol set (alphabet) of the Lua programming language consists of consists of letters, numbers, and special symbols.

4.1 Reserved words

The programming language Lua contains 22 reserved words, see the following tables.

Table 4.1 Reserve words in Lua

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

Table 4.2 Reserve words in Lua divided into groups

`constants`	`false true nil`
`variables`	`local`
`operators`	`and not or`
`conditions`	`if  then else elseif end`
`loops`	`for  in repeat until while`
	`do  end`
`functions`	`function  return`
`jumps`	`break goto`

4.2 Identifiers

The language symbols described so far are not enough to write a program. For one thing, there are too few, and most of them have their own special meaning. Therefore, anyone who writes a program creates his own building blocks, his own elements to which he gives an unambiguous name, or **identifier**. The identifier must meet certain rules:

- consists of letters, numbers and underline but the numeral identifier must not begin with;
- There is distinction between lower case and upper case (identifiers are case sensitive); the
- identifier must not be identical to the expressed word;
- identifier must be unique within the program or its part, that is, the same identifier cannot be used to refer to two or more simultaneous occurring simultaneously;
- the length of the sequence of characters of which the identifier consists, i.e. **identifier length** is not limited in Lua.

Table 4.3 Examples of correctly formed identifiers

NUMBER	_	A1a	_a
Number	a	AVERAGE	_ZS
number	x	SUM_OF_ELEMENTS	A1_2
nUMBER	z	MATRIX	TMP1

4.3 Numbers

We divide numbers into whole numbers and numbers with a decimal part. We can use Lua to write integerintegerinteger decimal or hexadecimal notation, hexadecimal notation begins with the prefix `0x`.

Numbers with decimal part is written only in the decimal system, but we can choose between decimal and semilogarithmic notation.

Decimal notation differs from the European continental convention in that after decimal point instead of a comma. **Semilogarithmic notation** of mantisy and exponentiation. Let's recall with an example: the number 14.75 can be written as $1.475{\cdot}10^1$. In the language Lua (as well as in other programming languages), the part '·10' is replaced by the letter `E` (or `e`). The number is then written as `1.475\,E\,1\,`.

Table 4.4 Examples of incorrectly created identifiers

Identifier	error description
1MATRIX	digits at the beginning
Numeró, Číslo	characters of the national alphabets are not among the letters
R>S	special symbols cannot be used
A1–2	minus sign (dash) '–' cannot be used, it is a character other than '_'
Sum of elements	separators (spaces) must not be part of the identifier
in	identical to the reserved word
FAST!	exclamation mark is not allowed

Before each number – the integer, with the decimal part, but also before the exponent – can have a + or − sign.

Table 4.5 Examples of correctly written numbers

Decimal system			Hexadecimal system
Integers	With decimal part		Integers
−18	24.	1.97e3	$A1A1
−18	104.75	−24e−2	−000 −0xF321
+18	−0.88765	0.11245E−06	−0x321
−0	−0.00000	0.00e00	−0x0

Table 4.6 Examples of incorrectly written numbers

Number	Error Description
23,98	decimal comma instead of a dot
123.456 e 7	number notation must not contain spaces
0x23.98	hexadecimal numbers must not have a decimal part
0x12W3	number notation contains an illegal character (W is not a digit)
2 * 3	occurrence of the special symbol *

4.4 Strings

A string is a sequence of arbitrary characters. Any sequence of characters that is to be understood as a string, is delimited from the surrounding text by an ’ (apostrophe), eg. `Hello!`, or by " (quotations marks), eg. `"Hello!"`.

The existence of two characters with which we can mark strings allows us to use these characters also inside the string, which otherwise would not be so simple:

```
"Hello, Mr O'Brien!"
'The character " is the quotation mark.'
```

A string that contains no characters is called **empty string**:

```
\type{''} or \type{""}
```

4.5 Separators

Separators separate the symbols from each other. In language Lua, there are four kinds of separators used: space, tab (a character with ordinal number 9, end of line, and a comment. There can be an unlimited number of delimiters side by side, allowing programs to be written neatly.

4.6 Comments

When writing a program, it is very useful to take notes on the program, that we create. A well annotated program is easy to read even after a long time. We will use them extensively in the examples given in this book notes. Notes are written in two ways, as we distinguish between notes single-line and multi-line.

```
january = 31   -- January has 31 days

january = 31   -- [[ January
has
31 days
--]]
```

Questions

(1) What do we call a symbol set?
(2) What are reserved words?
(3) What are identifiers used for?
(4) What rules are used to create identifiers?
(5) Decide which identifiers are written correctly and which are incorrect:

- First program
- ’Second program’
- ’−0.876 e6’
- "Great Britain"
- ’CALCULATION RESULT IS’
- ’CALCULATION_RESULT_IS’
- ’repetition
- "3 o’clock"

(6) What does the semilogarithmic notation of a number look like and what are its parts called?
(7) What does $ mean in integer notation?
(8) Which character do we use in Lua to separate the integer and decimal parts of a rational number?
(9) Decide which numbers are written correctly and which are incorrect:

- 123
- −256
- $8{\cdot}10^2$
- .333333333
- 1.999
- −876 e6
- C
- 1.8755e08
- −34C.17
- 0x12AB
- 0x12GA

5 Variables, data types, expressions

Each program (not only in programming language Lua) transforms input data into output data using precisely described steps. As a rule, programmes are not written for specific input data, but for input data that has certain specified properties.

5.1 Variables and Data Types

For example, let's consider two programmes: one programme can add two arbitrary numbers that we specify, the other can add only two specific numbers, for example 12 and 5. Such a program makes virtually no sense.

A program that has to add two numbers has to 'memorize' these numbers.

The part of operational memory where these numbers are stored is called **variable**.

Each variable must have two properties specified:

- the set of allowed values and
- the internal representation in the computer (memory size, encoding of values)
- the set of operations allowed for the given type.

We denote the description of these three properties collectively by the term **data type**.

In terms of the relationship between variable and data type, programming languages can be divided into two groups. In the first, 'classical', the data type is specified when the variable is declared, and thus it is given within the program that the variable will always take values only of this data type and no other. The Lua programming language belongs to the second group, where the data type is specified in the moment where a value is assigned. We say that variables are typed dynamically, i.e., when the program is running.

Therefore, there is no command to specify the data type in advance in Lua.

5.2 Brief overview of Data Types in Lua

There are eight basic types:

- nil – indicates an unassigned (non-existent) value
- boolean – data type of logical values true and false
- number – one common data type for integer as well as for floating point number, typically double (eight bytes) according to IEEE 754
- string – sequence of characters
- function – own or existing subroutines
- table – for all structured data type, implemented as an associative array
- userdata and thread (data from the hosting program in C and for implementation of coroutines, respectively (not discussed here)

Except the last two, all other data table will be explained in detail in the special sections.

The Lua programming language provides also a multiple assignment. On the left side, there is the list of variables, on the right side of the assignment the corresponding number of values. In the following text, we will see that on the right-hand side of the assignment can also used a function (see chapter 10) that creates the corresponding number of values at runtime.

```lua
a, b, c = 0, 0, 0
price, name, available = 100, "nice new shirt", false
```

Multiple assignment is very useful when setting default values in subroutines (called variable initialization) and is often used in iterative functions, which we will use for example for traversing through structured variables, called tables.

To conclude the talk about assignment, the following case needs to be pointed out:

```lua
local a,b=1,a+1
print(a,b)
```

If you have tried this example, you can enjoy the error message containing a reference to a non-existent variable a But we do use with a here. In any assignment – simple or multiple – all expressions on the right-hand side are evaluated first, followed by the actual storage of the values. So at the time of evaluating the a+1 expression, the a variable is not really available yet.

5.3 Expressions

An expression is a prescription for getting a value. An expression consists of operands, operators and parenthesis. These are properties we already know from mathematics. But let's look at the differences between a mathematical expression and an expression in Lua.

The first difference is that we write expressions in Lua at one level (called linearization of notation). We do not use fractions, exponents, or other special notations (for example, $\sqrt{}$ for the square root).

The second difference is found in the use of parentheses. There are three kinds of brackets used in mathematics – (), [] and {}. Expressions in programming languages including Lua only suffice with just one kind – parentheses (round brackets). However, there is no limit to the number of pairs of parentheses, so they can be used to express any complex expression.

The third difference lies in the type of value after evaluating the expression. In mathematics, the the value obtained is usually numerical. In a programming language, after evaluating the the value of not just a numeric value, but an arbitrary, not just a simple data type.

Operands can be expressed by variables, function calls (see chapter 10) or expressions.

Let's show a few expressions without further explanation:

```
     1                  true                "hello"

   a>=b         math.abs(x)>math.sqrt(y)   a+b

     a                 a*b+c                (a+c)*b

   (math.abs(x)+math.sqrt(z)>math.sqrt(y)-c)=

        (-math.abs(x)+math.sqrt(x)<>y*y+s)
```

The expressions, as we can see, are of varying complexity. The rules that allow expressions to write must cover all possibilities.

5.4 Operations and operators

In the previous text we noticed that the individual parts of the expressions are connected by special symbols, which here represent operators. The meaning of the term operator in programming languages corresponds to the common understanding of operators in mathematics.

Operators have two important properties:

- operator arity – expresses the number of operands that must be given to the corresponding operator. Typical operators are **unary**, which require one operand, and **binary** operators requiring two operands. For example, the operator / (division) requires a nominator and a denominator. The `not` operator requires only one operand, whose value it negates.
- precedence – expresses the order of expression evaluation. The priority of the operators is given in the Table 5.8.

5.4.1 Operation with logical values

The logical data type works with two logical values: true and false. The data type for these two values is called `boolean`.

There are four operations available for a logical data type, as listed Table 5.1. Table 5.2 shows what results we get for different values of p and q.

Table 5.1 Operation with logical data type

Operation name	Operator	Number of values required to calculate
logical negation	`not`	1
logical product	`and`	2
logical sum	`or`	2
logical exclusive sum (nonequivalence)	`~=`	2

Table 5.2 Evaluation of logical operations

Operands		Operations results			
p	q	p `and` q	p `or` q	p ^q	`not` p
false	false	false	false	false	true
false	true	false	true	true	
true	false	false	true	true	false
true	true	true	true	false	

5.4.2 Relational operations

Associated with a logical data type are the **relational operations** that that express a relationship between two values. Relational operations are binary (they have two operands), the symbols expressing the type of relationship are called **relational operators**relational operator. An overview of relational operations is shown in Table 5.3, the individual operations correspond to the same operations as in mathematics.

Table 5.3 Relational operations and relational operators

Relation	Mathematical Notation	Symbol in Lua
less than	$<$	`<`
less than or equal to	$\leq$	`<=`
equal	$=$	`==`
not equal	$\neq$	`~=`
greater than or equal to	$\geq$	`>=`
greater than	$>$	`>`

Suppose there are two numbers, a and b. The statement $a > b$ is either true or false. Expressed in the language Lua, the expression $a > b$ either takes the value `true` or `false`.

Thus, the result of any relational operation is a value of the data type `boolean`. In the previous paragraph we used the numbers a and b as an example. However, relational operations can be performed with other data types, eg with data types boolean or string.

5.4.3 Operations with numeric data type

Basic arithmetic operations are given by Table 5.4. The operations with numbers correspond to the usual conventions. The `-` character is used in two senses – as a unary minus to represent negative values (e.g., `-5`), and as a subtraction operator (e.g., `5-3`).

It is also worth remembering that we distinguish between ordinary and integer division, and that the remainder operation after division can be applied to numbers with a decimal part.

We use the abs function from the math library to determine the absolute value. The concepts of function and library will be explained later, for now it is sufficient to know that the library name and the function name are joined together by a full stop.

Table 5.4 Operations with numeric data types

Operation	Identificator or Symbol	Number of operands
(unary) minus	type-	1
addition	+	2
subtraction	-	2
multiplication	*	2
exponentiation	^	2
float division	/	2
integer division	//	2
remainder after division	%	2
absolute value	`math.abs`	1 parameter

Table 5.5 Comparison of math and Lua expressions

Math expression	Expression in Lua
$\frac{a+b}{a.b}$	`(a+b)/(a*b)`
$\sin\gamma$	`math.sin(gamma)`
$I < J < K$	`(I<J) and (J<K)`
$m \in (33{,}5; 100\rangle$	`(m>33.5) and (m<=100)`
$x \bmod y$	`x % y`
x^2	`x^2`
x^{i+2}	`x^(i+2)`

5.4.4 Rounding functions

The programming language Lua provides two basic functions for rounding. They are called `math.floor` and `math.ceil` and result in rounding down or up, respectively.

```
print(math.floor(5.5), math.floor(-5.5))
print(math.ceil(5.5), math.ceil(-5.5))
```

If we want to round to decimal places, we have to use this way:

```
print(math.floor(5.55*10)/10, math.floor(-5.55*10)/10)
print(math.ceil(5.55*100)/100, math.ceil(-5.55*100)/100)
```

5.4.5 Bitwise operations with integer numbers

Bitwise operations belong to lower-level operations, because they manipulate the individual bits of the operands directly. To understand these operations, we need to imagine the internal representation of integers in computer memory. The number is converted into a binary system (either a positive number into a direct code, or a negative number using an additional code) and this sequence (vector) of bits is the operand for bitwise operations.

The operations we present here are performed for each pair of corresponding bits separately, and their results correspond to the results of logical operations.

As an example, we will show the operation of the bit product of the values 25 and 7. First, we will convert the numbers into the binary system:

```
25 = 16 + 8 + 0 + 0 + 1  =>  11001
15 =  0 + 8 + 4 + 2 + 1  =>  01011
```

We will now evaluate a pair of bits using logical operations. We assume that `1=true` and `0=false`:

```
11001
01011
-----
01001
```

In the first and fourth positions from the right we got ones, because both operands of the given position also have a one. In the other three cases, the logical sum acquires the value false, we wrote it as zero. The following is the conversion of the result back to the decimal system:

```
01001 =>  0 + 8 + 0 + 0 + 1 = 9
```

Further bitwise operations are given with a short explanation in table 5.6. The bitwise operations shift right and shift left shift a sequence of bits by the specified number of positions, padding the missing positions with zeros:

```
25 (00011001) >> 2 =>   6 (00000110)
15 (00001011) << 4 => 240 (10110000)
```

Table 5.6 Bitwise operations and operators

operation	operation	number of operands	
bitwise NOT (negation)	~	1	bit inversion
bitwise AND ()	&	2	both must be 1
bitwise OR ()	\eTD \bTD	2	at least one must be 1
bitwise exclusive OR ()	~	2	just one must be 1
shift right	>>		
shift left	<<		

5.4.6 Operation with string data type

By the term string, we imagine a sequence of characters that together form a certain meaningful unit (e.g. word, sentence). By character we mean any eight-bit value according to the character table, regardless of encoding. Encoding can also be multi-byte (e.g. UTF-8), a multibyte character is stored on multiple bytes. Control character (characters with ordinal value between 0 and 31, eg.`\0`, `\1`, ...) are also permitted.

The basic two operations with the string data type are length determination and string concatenation (see table 5.7.

Table 5.7 String operations

symbol	operation	number of operands
#	string length determination in bytes	1
..	concatenation of strings	2

Strings in the Lua programming language are understood as immutable, which means that we cannot intervene in them (change characters, remove characters, etc.) The `string` library is intended for these operations, see Chapter 8.

```
r = "Hello"
print(r,#r)
r = r .. " world!"
print(r,#r)
```

5.4.7 Detection of data type

There is the predefined function `type` which returns a string containing the name of a data type of the parameter:

```
print(type(123))
print(type(123.456))
print(type(0x1234))
print(type("Hello!")
print(type(true))
print(type(nil))
print(type(math.abs))
```

5.4.8 Conversion between number and string

To change the data type from number to string and back, the Lua programming language offers two functions – tostring and tonumber:

```
a = 5
print(a,type(a))
b = tostring(a))
print(b,type(b))
c = b.."7"
print(c,type(c))
d = tonumber(c))
print(d,type(d))
```

Non-number strings cannot be converted to a number:

```
p = "Hello!"
q = tonumber(p)
print(q)     -- nil
```

5.5 Expression evaluation and precendence of operators

As in mathematics, an expression that is in parentheses, has an even higher priority, which means we evaluate it first. The evaluation of standard functions also has a higher priority. About some standard functions we have already mentioned in the data types. An overview of selected Lua math functions is given elsewhere.

5.5.1 Expression evaluation a precedence of operators

If all operators have the same priority, they are evaluated from left to right, as they were used in the just processed expression.

See the Table 5.5 for some examples of mathematical expression notation in Lua.

Table 5.8 Precendence of operators

operators/operations	symbols
exponentiation	^
unary operators	not # -
multiplicative operators	* / // %
additive operators	+ -
string concatenation	..
bitwise shifts	<< >>
bitwise and	&
bitwise not	~
bitwise or	|
relational operators	< > <= >= = ==
logical product	and
logical sum	or

Note: Operators are ordered by its priority in descending order.

Questions

(1) Explain the three main differences between the mathematical notation of the expressions written in the language Lua.
(2) Explain the terms arity and operator precedence.

Exercises

(3) Write the following expressions in Lua:

- $\frac{5}{7}$
- $0{,}25 \cdot y$
- $\frac{1}{3}a + \frac{1}{9}a^2$
- $\sqrt{2}(a-b)^2$
- $\sin x^2 + \sin^2 x$
- $x^2 < y^2 < z^2$
- $[(d-e)^3 + f] \cdot (a-2)$
- $a <> b <> c$

(4) Evaluate the following expressions:
- 20 // 7
- 20 / 7
- 20 % 7
- math.floor(5.5)
- math.floor(−5.5)
- math.ceil(5.5)
- math.ceil(−5.5)

For the following expressions, decide whether they are written correctly or not.
- b∗math.sin(a)<a∗sin(b)
- 2
- a≠ b−1
- (a<b)+1=0
- (a<b)=(c>0)
- sqrt(r−1)/100

(5) For the following pairs of values, determine the result of the expression. Write the results with a pencil and then correct them according to the computer.

boolean boolean	false and false	
	false and true	
	true and true	
	false or false	
	false or true	
	true or true	
boolean nil	false and nil	
	true and nil	
	false or nil	
	true or nil	
nil boolean	nil and false	
	nil and true	
	nil or false	
	nil or true	
nil number	nil and 11	
	nil and 0	
	nil or 11	
	nil or 0	
number number	4 and 2	
	4 or 2	
	x and 4	
	x or 4	
string number	"aaa" and 2	
	"aaa" or 2	
string nil	"aaa" and nil	
	"aaa" or nil	
	nil and "aaa"	
	nil or "aaa"	

6 Procedures for data input and output

In the previous text, we used procedures `io.read` and `print` with no detail explanation. In addition, there is also a procedure `io.write` for the same purpose as `print`. Procedures `io.read` and `io.write` deserve much more attention, as they can be used to communicate between the user and the program. As a rule, every program handles some input data into output. The input data must be entered, the output of the program is needed to be displayed.

Input and output data are usually stored in in files. A file is a data type that will be described in detail in chapter .

However, here we will mention only two special files, which are the standard input file and the standard output file. Both files are text files, which means that there is a pair of characters between the lines with the coordinate values 13 and 10[2].

In programming language Lua, these two files are accessed automatically.

So using the `io.read` and `io.write` procedures is actually using the standard input and output files. The following section will discuss the behavior of these two functions. How do they behave with other kinds of files will be described along with the file data type.

Table 6.1 lists the data types of the programming language Lua that can can be handled by `io.read` and `io.write`.

[2] This pair of characters occurs only on personal computers running DOS, Windows, etc. In operating Unix-like systems, the end of the line is marked only by the character 10.

Table 6.1 Overview of read and write options of data types by the `io.read` and `io.write` procedures

data type	`io.read`	`io.write/print`
Simple data types		
number	yes	yes
char	yes	yes
boolean	no	yes
Structured data types		
string	yes	yes
table	NO	NO

6.1 Read procedure `io.read`

Reading into an arbitrary numeric variable – From the input file, characters corresponding to the lexical definition of a number, i.e. sign, digit, for floating point numbers, then a decimal point, possibly a letter `e` for exponent are read.

The reading of a number may be preceded by an arbitrarily long sequence of delimiters (spaces, line breaks, tabs), which the procedure will ignore.

```
x = io.read()  -- Enter a number
print(x, type(x))
print(x*2)     -- Not possible
```

```
x = tonumber(io.read())  -- Enter a number
print(x, type(x), x*2)   -- OK
```

Reading into a variable of type `string` – the string is read from the input file a whole line and it is assigned to a variable as a whole.

```
x = io.read()  -- Enter a string
print(x, type(x))
```

Reading into a variable of type `string` with defined amount of characters – one is read from the input file character is read from the input file and assigned to a variable.

```
x = io.read(4)   -- Enter the string "Hello!"
print(x, type(x))
```

Note: If we reach the end of a line and read the input by characters, Linux users will have to read the end-of-line character to get the beginning of the ne line. Windows users will have to read both (13+10) characters.

6.2 Procedure `io.write`

Displaying the value of the data type `string` – in the output file the string will appear as it is stored in memory.

Displaying value of data type `boolean` – the output the strings `false` or `true` will appear on the output depending on the value of the expression.

Displaying numbers is identical to the usual notation. Floating point numbers will be printed out in decimal form. For too big numbers, the exponential form, i.e., mantissa, the `e` sign, and the exponent, will be used automatically.

6.3 The differnce between `io.write` and `print`

Let's start with an example:

```
a = "Hello"     b = "world"     c = "!"
print(a,b,c)
io.write(a,b,c)
print(a)
print(b)
print(c)
io.write(a)
io.write(b)
io.write(c)
io.write(a.." "..b..c)
```

So we see that the difference is the addition of the end-of-line character.

6.4 Output Information Formatting

For printing formatted strings, we can use the function `string.format`. It interprets the string literals in the provided string and replaces them with the appropriate value which is defined by the first parameter. The first example will arrange name, surname, city and favourite colour:

```
n = "John" s = "Smith" c = "Dallas" fc = "blue"
print(string.format("Mr %s %s from %s prefers %s colour.",n,s,c,fc))
print(string.format("%s, %s (%s): %s",s,n,c,fc))
print(string.format("%s' citizens (eg. %s %s) prefer %s.",c,n,s,fc))
```

There are variables for day, month and year. Print them in Czech, US, British, French, and international numeric formats.[3]

```
d = 30    m = 5   y = 2003
date = string.format("%02d.%02d.%04d",d,m,y) -- GB
print(date)
date = string.format("%02d/%02d/%04d",m,d,y) -- US
print(date)
date = string.format("%02d. %02d. %04d",d,m,y) -- CZ
print(date)
date = string.format("%2d/%2d/%04d",d,m,y) -- FR
print(date)
date = string.format("%02d-%02d-%04d",y,m,d) -- ISO
print(date)
```

Table 6.2 Overview of output formats

symbol	for which output format	short example	output
d	decimal number	`"%02d",44`	44
x	hexadecimal	`"%02x",44`	2c
o	octal	`"%02x",44`	54
f	floating point number	`"%5.2f",8.2`	5.20
s	strings	`"%s","Lua"`	Lua
q	adding quotations marks	`"%q","Lua"`	"Lua"
%	percent sign	`"3d%%",100`	100%

[3] There are more ways how to write date in UK, US, or France; this is only one possible way.

For more complex cases, we can use a clearer way with the extra string variables which contains the output format specification:

```
d = 5; m = 11; y = 2021
dateformat = "%02d/%02d/%04d"
date = string.format(dateformat,d,m,y)
print(date)
```

Assume variables cr and ci with real and imaginary part of a complex number. Print them in typical way in math:

```
cr = 5.111  ci = 2.555
formatcomplex = " %010.5f + %010.5fi "
print ( string.format(formatcomplex, cr, ci) )   -- not very fine
formatcomplex = " %10.5f + %10.5fi "
print ( string.format(formatcomplex, cr, ci) )   -- better
formatcomplex = " %.5f + %.5fi "
print ( string.format(formatcomplex, cr, ci) )   -- the best
```

The following example shows the use of q:

```
string.format('%q', 'This string contains "quotes" and \n new line')
```

will produce the following strings:

```
"This string contains \"quotes\" and \
new line"
```

Questions

(1) Which data types cannot be processed by the `io.read` procedure?
(2) Which data types cannot be processed by the `io.write` procedure?
(3) What is called formatting of output information?
(4) What methods do we use to modify output information?

Exercises

(5) Which date format do you use in your country? Prepare the formatting string.
(6) Assume there values, numerator, denominator, and their quotient. Prepare the formatting string which will give you this output:

```
If we divide 3 by 7 (3/7), we will get 0.428571, ie. 42.86%.
```

7 Control structures of Lua

7.1 Conditional command

7.1.1 Conditions and command `if`

The `if` command is used to branch the program into two branches (called binary branching), whereby the program uses only one of the branches and skips the other.

```
if condition
  then commands
end

if condition
  then commands1
  else commands2
end
```

The symbol ‘condition’ will be used here and in other syntactic graphs as an expression whose evaluation yields a value belonging to the data type `boolean`, i.e. either a value `true` or `false`.

The second example contains the ‘else’ part, which is optional. This means that the `if` statement can occur in two forms. The first of these is called **incomplete command `if`**, the second one **complete command `if`**.

How is the `if` command executed?

(1) The program first evaluates the boolean expression.
(2) If the resulting value is `true`, then the statement or statements following the reserved word `then`.
(3) If the value `false` is obtained, the part after the word `then` is is skipped and the commands following the reserved word are executed `else`, of course only if we have ‘else’ have been used.

```
if shape=="rectangle" then read(CisloA, CisloB) else read(Cislo) end
if Cislo%2==0 then print('This number is odd.') end
if Cislo<10 then Soucet = Soucet+CisloA end
if Mesic==1 then Days = 31 else
  if Mesic==2 then Days = 28 else
    if Mesic==3 then Days = 31
end end end
```

We will demonstrate the use of `if` in a program that calculates the functional value of a function $y = \frac{\sqrt{x}+3}{x^2-1}$ for the number x that is in the input file. If the function value is not in the real number domain, the output will be message ‘Undefined’.

Analysis: The function does not have a solution in the real number domain for all $x < 0$ because x is subtracted. At the same time, $x^2 - 1$ must be different from zero, otherwise it cannot be divided.

```
io.read(x)
if x>=0 and x~=1
   then print((math.sqrt(x)+3)/(x*x)-1)
   else print('Undefined.')
end
```

As another example let’s create a program that calculates the roots of a general quadratic equation in normalized form. The coefficients of each term are on the standard input.

The input is three numbers. The output is one or two numbers – the roots of the quadratic equation. Algorithm:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
a, b, c = io.read("*n", "*n", "*n")
discriminant = (b*b)-4*a*c --[[ Calculating the discriminant with
                         must be performed before using it under the square root--]]
if discriminant>=0 -- Calculate the real roots
 then sqrtd = math.sqrt(discriminant)
      root1 = (-b + sqrtd) / (2*a)
      root2 = (-B - sqrtd) / (2*a)
      print('Real roots: ',root1)
      print(' ',root2)
 else -- Computing complex roots
      root1 = -b / (2*a)
      root2 = math.sqrt(-discriminant) / (2*a)
      print ('Complex roots: ', root1,'+',root2,'i ')
      print(root1,'-',root2,'i ')
 end
```

Another example is a program that reads three numbers and prints them sorted by size.

```
 local a,b,c = io.read("*n", "*n", "*n")
 if a>b
   then if b>c then print(a,b,c)
               else if a>c then print(a,c,b)
                           else print(c,a,b)
                    end
        end
   else if a>c then print(b,a,c)
               else if b>c then print(b,c,a)
                           else print(c,b,a)
                    end
        end
end
```

7.1.2 Command if+elseif

The `if` command can also be used to branch a program into multiple branches.

```
if condition then commands1
  elseif condition2 then commands2
  elseif condition3 then commands3
  elseif condition4 then commands4
  ...
  else commandsx
end
```

The branches are labeled with conditions that are evaluated sequentially as it's their turn. If any of the conditions takes the value `true`, the conditions of the other branches are no longer evaluated and the branches are skipped.

Again, we can write an optional branch starting with a reserved word `else`. The command given in this branch is executed just then, if none of the conditions for the individual branches have been met.

Another example with multiple branching advises the entrepreneur how to proceed in his warehouse management. Assume that the variable `how many` represents the current quantity of the goods in stock:

```
if        how many <   10 then print('Quickly restock and use express shipping!')
  elseif how many <   33 then print('Restock!')
  elseif how many <= 299 then print('Do nothing.')
  else    print('Discount and get rid of stock.')
end
```

Let's think about what happens if multiple branches are described by the same constant. The branch that is listed first is executed, and skip the other branches. In the above example this would occur, for example, if we changed the order of the conditions:

```lua
if       how many <   10 then print('Restock quickly and use express shipping!')
  elseif how many <= 299 then print('Do nothing')
  elseif how many <   33 then print('Restock!')
  else   print('Discount and get rid of stock')
end
```

Now the entrepreneur gets the advice to do nothing, for any quantity of interval 11 to 299, and the program never gets to the third condition.

Exercises

(1) Create a program in Lua language to find out if the given number is even or not.
(2) Create a program in Lua to determine whether a given number is positive, negative or zero.
(3) Create a program in Lua to calculate the perimeter of a triangle given by the lengths of its three sides.
(4) Create a program in Lua to calculate the area of a right triangle given by the lengths of its sides.
(5) Create a program in Lua to calculate the perimeter of a right triangle given the lengths of its branches.
(6) Create a program in Lua to determine whether a triangle given by its side lengths is equilateral, isosceles, or right-angled.

7.2 Loops

Loop commands allow you to order multiple repetitions of a command or a sequence of commands. The part to be repeated is called the loop body. Furthermore, each loop statement must contain a specification of the number of repetitions, either explicitly, i.e., by specifying specific numbers, or by a condition (implicitly), i.e. by specifying the condition at which the loop is to terminate.

- command `while` – the number of repetitions is specified by a condition evaluated *before* the part of the program to be repeated,
- `repeat` – the number of repetitions is determined by the condition evaluated by the *after* part of the program to be repeated,
  `for` – the number of repetitions is determined explicitly and is determined by the number of values.
- `for...in` – the number of repetitions is specified explicitly and is determined by the number of values.

The way the number of repetitions is specified has a great influence on the choice of the appropriate loop statement in different situations. When deciding which loop to use, we must first be clear whether we are able to specify the number of repetitions in advance, or whether the loop will be controlled by reaching a certain state.

For comparison the loops, see the following code:

```
while condition do
  commands
end

repeat
  commands
until condition

for variable=start,end do
  commands
end

for variable=start,end,step do
  commands
end

for key,value in list do
  commands
end
```

7.2.1 Loop `while`

The procedure for processing the `while` loop:

(1) Evaluates a boolean expression.
(2) If the resulting value is true, the following commands are executed after the reserved word `do` until the reserved word is encountered `end`.
(3) Then the boolean expression is evaluated again (go to step 1).

This loop continues until a situation occurs in which the resulting Boolean expression is false. The statement after the reserved word `do` is then no longer executed, and continues with the command following the loop `while`, i.e. after the word `end`.

The `while` loop has a boolean expression at the beginning, i.e. before the first the first iteration of the loop. If the expression is false on the first evaluation, then the loop body is not executed at all. It follows that the number of repetitions of this of the loop is from the interval $\langle 0; \infty \rangle$.

We will demonstrate the use of `while` on two programs. First, calculate and print of the square and third powers and of the powers:

```
x=0
while x<100 do
  x=x+1
  x2=x*x
  print (x, x2, x2*x, math.sqrt(x), x^(1/3) )
end
```

The variable `x` is of great importance here, since its value will ensure that in certain point in time. Therefore, its value is 0 at the beginning and inside the loop its value is always incremented by 1.

Here we can also notice two other algorithmic constructions – the calculation of the third power and the third root. Since the third power is computed as the product of the square root (which anyway we need) and the next x, it pays to calculate the square root in advance and then use it twice. For the third power, the situation is more interesting – for this calculation we do not have a predefined function for this calculation. That iss why we used other way – a mathematical expression. Similarly, we could do the same for the second power (`x^(1/2)` or `x^0.5`), but if we can, we prefer existing subroutines.

The second program calculates the greatest common divisor by successive subtraction:

```
i,j = io.read("*n", "*n")
while i<>j do
  if i>j then i=i-j else j=j-i end
  print(i)
end
```

Here the loop is terminated if `i` and `j` are equal.

7.2.2 Loop `repeat/until`

The procedure for processing the `repeat` loop:

(1) Executes a statement or sequence of statements between reserved words `repeat` and `until`.
(2) A boolean expression is evaluated.
(3) If the resulting value is false, the execution is repeated. the statement(s) in the loop (go to step 1).

The loop proceeds in this manner until a situation occurs in which the resulting value of the boolean expression is true.

The `repeat` loop has the boolean expression at the end, i.e. after the body of the loop. This means that the possible number of repetitions of this of the loop is from the interval $\langle 1; \infty \rangle$ (compare with the `while` loop).

We will show the use of this statement again with three examples, first we will calculate the product of two natural numbers by successive addition.

```
if i~=j then tmp=j j=i i=tmp end
    -- Now the value of variable i is a smaller value of variable j
product=0
tmp=i
repeat
  product:=product+j
  tmp=tmp-1
until tmp==0
print (i," * ", j, " = ", product)
```

The second example shows a program that calculates a factorial by successive multiplication.

```
factorial=1
factorial=0
end=10e10
repeat
  digit=digit+1
  factorial=factorial*cislo
  print(number, factorial)
  --[[ This print command
       is used only to illustrate the operation of the program.
   --]]
until factorial>end
print(number, "! = ", factorial)
```

And the last one computes the numeric sum of the given number.

```
x = io.read()
x = math.abs(x)
local nsum = 0
repeat
  local nx = x//10 -- math.floor(x/10)
  nsum = nsum + (x-nx*10)
  x = nx
until x==0
```

For integer division we have an alternative method of calculation in the note. This was necessary in older versions of Lua. From version 5.3 onwards we will use the integer division operator in preference.

7.2.3 Loop `for` – numeric

The `for` command is used in cases where the number of of loop repetitions. We always specify the starting (S) and ending (E) value of the loop variable. Number of repetitions is given by $N = |E - S| + 1$. The following value is incremented by 1 from the previous value.

In the Lua programming language, we can also specify a pair of values, where the initial value is less than the final value ($S < E$). For the loop to run, we must then add negative step information between two consecutive values.

We can use this extended notation for any other step.

Let's demonstrate the `for` loop statement on a program that has to add 10 numbers of the input series. Thus, the number of repetitions of the loop is known.

```
sum=0
for count=1,10 to
  number = io.read("*n")
  sum = sum + number
end
print(sum)
```

The variable that controls the loop's operation is extinguished after the `for` loop ends. It is therefore a so-called local variable.

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

The value nil, provided by the last `print` statement, is the proof, that the variable no longer exists and therefore can have no value.

7.2.4 Loop `for` – generic

See the chapter on tables for more examples of using the `for` loop, where it will serve as a basic tool for traversing an indexed array (table). There you will also see its other variant `for...in`, which traverses associative array, i.e. a table whose values are not indexed.

7.3 Examples to practice loops

Let's have an input file with the following data: first, an integer N that gives the number of employees, followed by N numbers representing the amounts to be to be paid to each employee. We have banknotes with a value of CZK 2,000, CZK 200 and CZK 100. Find out how many of each type the cashier needs to pay all the employees.

```
-- The counters must be reset before use.
TotalB2000, TotalB200, TotalB100 = 0, 0, 0
io.read(number of employees)
for i=1,number of employees to
  io.read(salary)
  B2000, zust = salary//2000,salary%2000 -- Number of banknotes with a value of 2 000 CZK
  B200 = zust//200
  B100 = (zust-B200)//100
  totalB2000 = totalB2000+B2000
  totalB200 = totalB200+B200
  totalB100 = totalB100+1
end
  print(totalB2000, totalB200, totalB100)
```

A teacher has decided to award a grade in a certain subject according to the number of points on the midterm tests.

Point range		Grade
from	to	
100	80	1
79	65	2
64	50	3
49	30	4
29	0	5

The input is a series of integers representing the points scored. The series is terminated by by −1. Find the average grade corresponding to the given score values.

```
sum = 0
count = 0
io.read(points)
while points<>-1 do
  if     points>=80 then grade = 1
    elseif points>=65 then grade = 2
    elseif points>=50 then grade = 3
    elseif points>=30 then grade = 4
    else   grade = 5
  end
  count = count + 1
  count = count + grade
  io.read(points)
end
print(sum/number)
```

7.3.1 Reading from a text file

We know that `repeat` and `while` are used when we don't know in advance the number of repetitions. In the least common divisor example, the loop until `i` and `j` were equal. Reading the input data in this problem was to read two numbers from the input file. The reading was done with the procedure `read` with two parameters. So we knew in advance the number of values we needed to read.

However, there are a number of tasks (and perhaps most of them), where we don't know in advance the number of numbers we're going to read. These cases are dealt with in the next two sections.

7.3.2 Reading the input series to a stop value

Let's have an input series of numbers in the input file. Their number is not predetermined, so we can't tell which is the last number. To make it clear when to stop reading, a number of algorithms use a so-called **stop value**, which is the number listed next after the number series. Its value is known in advance and this value does not appear anywhere in the in the whole number series. This value is usually very different from other values.

Let us give an example: In the input file, there is a series of numbers that represent the number of students in a class. Find the total number of students in the whole school (in all classes).

The input series looks like this:

```
24 21 17 33 32 25 .............. 17 12 9999
```

To find out when to stop reading, we give the last value 9999, which obviously does not represent the number of students. This number is a stop and we will check to see if it has already been read.

```
school = 0                   -- Each counter must be reset.
io.read(class)               -- Read the number of pupils in the class
while class<>9999 to         -- If there is no stop value...
  school = school + class -- Add class to school.
  io.read(class)             -- Read the next class.
end
print(school)                -- Display the result.
```

7.3.3 Read input data to the end of the input file

Let's have in the input file an input series consisting of numbers, the number of which again we don't know. There is not even a stop to indicate the end of the input data. We will show what facilities the Lua language provides for dealing with this situation in the following example.

Build a program that calculates the average of all the numbers in the input file. For example, we can imagine that these are the weights of some products, expressed in kg, possibly including the decimal part.

First, let's build a verbal algorithm:

- Prepare a variable to which we will add the continuously read and a variable in which we will keep the number of read values.
- As long as *there is a number in the input file*, read the number, add it and increment the read counter by one.
- When all the numbers have been read, calculate the average as the ratio of the sum and the number.

Now we build the program, except for what is written with *cursive*, i.e. at least the part that we can write with familiar commands:

```
count = 0
while \emph{\uv{is some number in the input file}} do
  io.read(number)
  count = count + 1
  count = count + number
end
print(sum, count)
```

The Lua language does not have any special function that can detect this. Therefore, different solutions are used. The first is to read the first value already before the loop starts.

```
count = 0
number = io.read("*n")
while number do
  count = count + 1
  count = count + number
  number = io.read("*n")
end
print(sum, count)
```

```
count = 0
count = 0
number = io.read("*n")
repeat
  count = count + 1
  count = count + number
  number = io.read("*n")
until not number
print(sum, count)
```

Conditions expressed by the identifier of a numeric variable (see `while cislo do`) has an unusual behavior for us: the program does not evaluate the value of the number, but whether there is any number in the variable `cislo` (here numeric) value, or whether the variable contains the value `nil`, which indicates the absence of a (numeric) value. Thus, if the input was a number, we get here the value `true` (some number is present) and this allows the loop to be entered.

After processing the number, we read the next value from the input at the end of the loop and everything repeats.

With this solution, some programmers consider the necessity of two read statements, one before the loop and one at the end of the loop body. Therefore, there is another method that transfers the decision to continue in the middle of the loop body. This saves one read statement, but it with more complex constructs, it may not be obvious at first glance where the loop actually ends. On the other hand, this solution is used quite often in practice.

```
count = 0
set = 0
while true do
  number = io.read("*n")
  if not number then break end
  count = count + 1
  count = count + number
end
print(sum, count)
```

If we had characters (not numbers) as input, we could afford to one more trick:

```lua
while io.read(0) do
  character = io.read(1)
  print(character)
end
print("---")
```

What's going on here? The `io.read` function with a numeric parameter supplies from input stream a string of length corresponding to the specified number of characters. With zero, the function returns an empty string if there is still some data in the input are present, allowing us to read the next character. Otherwise, the function will return the value `nil` as expected, which in condition on the `while` loop, it evaluates to `false`, and this causes terminate the loop.

Questions

(7) List and briefly describe the structured commands in Lua.
(8) What is the `else` part of branching commands used for?
(9) Which loop always runs at least once? Why?
(10) What is a control variable?
(11) Why can't we use the `for` loop to process a series of numbers with a stop

Exercises

(12) Rewrite the `for a = 10,2,$-$1 do ...` with the `repeat`.
(13) Create a program in Lua to calculate the factorial of a given natural number.
(14) Create a Lua program to calculate the maximum of a series of numbers ending in zero.
(15) Create a Lua program to list the number of numbers smaller than the first number in the series. The series is terminated by a value equal to the first number.
(16) Create a program in Lua to list the first n Fibonacci numbers.
(17) Create a program in Lua to calculate the value of a combinational number (n k) for the given values of n, k.

(18) Create a program that reads a number from an input file and determine whether it is in the interval $\langle -\frac{3}{2}\pi; 0\rangle$ or $(\frac{\pi}{2}; \pi\rangle$.

(19) Create a program that reads the coordinates of a point and finds out, whether that point lies on the line $y = 5x - 3$.

```
The numbers ..., ... and ... satisfy the given inequality.  (or:)
The numbers ..., ... and ... do not satisfy the given inequality.
```

(20) The input is a number N followed by N numbers. Create a program, to find out how many numbers lie in the interval $\langle -0.1; 5)$.
(21) Construct programs to calculate a number series with precision ϵ=0.001:

- $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}\dots$
- $\frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \frac{1}{16} - \frac{1}{32}\dots$

(22) Create a table of function values (function table) for the values with step 0.1 in the interval $\langle -5; +5\rangle$ for functions:

- $y = \frac{\sin\sqrt{x}+3}{x^2-1}$.
- $y = x^2 + 3x - 1$
- $y = 6z^2 + \frac{3}{z} - \frac{1}{8}$

Note that for some numbers the function value may not be defined.

(23) Rewrite the example with the number of students in the school using the repeat loop. Decide which of the two loops is more appropriate in this case.

(24) Complete the calculation of the average number of points for the problem 'Notes' and the average grade.

(25) The input file contains a series of numbers ending with −5,000. Find out how many numbers belong to the interval $\langle 1; 100\rangle$, $\langle 101; 200\rangle$, $\langle 201; 300\rangle$, $\langle 301; 400\rangle$ and $\langle 401; 500\rangle$. Find out how many numbers did not belong to any in any of these intervals, too.

(26) In the input sequence, there are triples of numbers that represent the lengths of of the sides of the triangle. The sequence of triples is terminated by the triple 0 0 0. Find out how many triangles are right-angled, how many are equilateral and how many isosceles.

(27) The input is decimal numbers representing the exchange rate of the US dollar to the Czech crown. Find the subtraction between the best and worst exchange rate.

(28) The input is a pair of numbers representing the performance of the competitors in the two rounds of the slalom. The numbers are expressed in seconds or with decimal fractions. The number zero instead of time represents a disqualification of a competitor. Find out how many competitors were disqualified in the first round, how many in the second round (those disqualified in the first round do not enter the second round) and also has a zero). Next, find out the fastest competitor in each round and the competitor who was fastest overall in both rounds. The sequence of pairs of numbers is terminated by the pair −1 −1.

8 Strings and `string` library

We have already familiarized ourselves with the first two subroutines for working with strings, determining the length and concatenating strings. These two activities are implemented directly in the Lua programming language. Here we will look at the string library, which contains other useful functions.

As you can see from the following example, you can divide functions for working with strings into four thematic categories.

```
-- CHARACTER MANIPULATION
string.byte (s [, i [, j]])
string.char (···)

-- STRING MANIPULATION
string.lower (s)
string.upper (s)
string.reverse (s)
string.rep (s, n [, sep])

-- SEARCH & REPLACE
string.find (s, pattern [, init [, plain]])
string.gmatch (s, pattern)
string.gsub (s, pattern, repl [, n])
string.match (s, pattern [, init])
string.sub (s, i [, j])

-- OTHER FUNCTIONS
string.len (s)
string.format (formatstring, ···)
```

8.1 Conversion between characters and ordinal values

In each character set, the individual characters are arranged into the given order and the positions in the character set are called ordinal values. The functions `string.byte` and `string.char` are used to determine the ordinal value of the given character(s) and to determine which character corresponds to the specified ordinal value, respectively.

```
s = "@ABCDEF"
print (string.byte(s))          -- 64 (ord. value of @ is 64)
print (string.byte(s,4))        -- 67 (ord. value of C is 67)
print (string.byte(s,4,4))      -- 67 (the same)
print (string.byte(s,4,6))      -- 67 68 69 (values of three characters)
```

By default, the ordinal value of the first character will be returned. We can specify another character using its position in a string. For multicharacter process, the third parameter determines the ending position in a string.

8.2 String manipulation

8.2.1 Upper and lower case letters

The first snippet does not need any explanation:

```
s = "Programming language Lua"
print(string.upper(s))
s = "HELLO!"
print(string.lower(s))
```

Unfortunately, this work only for English alphabet, national characters will not be changed:

```
s = "CS:čšřďt FR:àâæïî DE:äëöü"
print(s,string.upper())
```

The following snippet is also incorrect:

```
s = "Programming language Lua"
string.upper(s)
print(s)
```

But why? The second line calls the `string.upper` as a procedure. However, we know that all strings in Lua are immutable, so we cannot expect that the string will be changed.

All these functions create a new string and use them as a 'return' value which must be assigned to another variable:

```
s = "Programming language Lua"
s = string.upper(s)
print(s)
```

8.2.2 Reverse and replication

The `string.reverse` function reverses the order of characters and the `string.rep` replicates the string (the first parameter) n-times (the second parameter), or separated with the string from the third parameter.

```
s = "Programming language Lua"
print(string.reverse(s))

print("*********************")
print(string.rep("*",21))
print(string.rep("*",11," "))
```

8.3 Search and replace functions

In examples for searching and replacing, let's assume the following assignments:

```
s        = "Programming language Lua"
word = "language"
pattern2 = "a"
```

8.3.1 Simple search

Does the string s contain the word 'language'? The function `string.find` searches for the given pattern.

```
print(string.find(s,"a"))            --  6  6
print(string.find(s,"language"))     -- 13 20
print(string.find(s,"language",12)) -- 13 20
print(string.find(s,"language",15)) -- nil
```

It might be surprising that two values have been returned. The values indicate the start and end positions of the first occurrence of the pattern. The third parameter stands for the determination of the starting point for the search. The `nil` value indicates no occurence of the given pattern.

8.3.2 Search with classes

Character classes cover groups of characters, eg class digits stand for characters 0 to 9, upper case letter for characters A to Z, etc. Except the first one (see Table 8.1), all classes start with the % symbol followed by a letter. If you compare class with lower and upper case symbols, the upper case class is the complement of the lower case class.

Table 8.1 Predefined and own character classes

symbol for the class	description
.	one any character
%a	letters
%c	control characters
%d	digits
%g	printable characters except spaces
%l	lower case letters
%p	punctuation characters
%s	space characters
%u	upper case letters
%w	alphanumeric characters
%x	hexadecimal digits
%A--%Z	capital letters denote complements to %a–%z
[...]	own class; set of characters
[^...]	own class; complement of the set of characters

Let's continue with the first class mentioned in the Table 8.1, with `.`. We would like to get position of substrings covered by the pattern.

```
a = "square root has been computed"
b = "our new swimming pool has been closed"
print ( string.find(a, '.oo.') )
print ( string.find(b, '.oo.') )
as, ae = string.find(a, '.oo.')
bs, be = string.find(b, '.oo.')
print (as, ae, bs, be)
```

We can now use these indices to get the matched text, but there's a better way: the string.match function. It returns the matched text, or nil if the pattern is not found: (actually, find also returns the matched text, but it first returns the indexes; match only returns the text)

8.3.3 Getting a substring using indices

After determining the indices where the searched substring starts and ends, we are able to use the string.sub function to get this substring:

```lua
print ( string.sub (a, as, ae) )
print ( string.sub (b, bs, be) )
```

8.3.4 Getting a substring using matching function

For getting the substring corresponding to the `.oo.` pattern (instead of indices), we can use the `string.match` function. It returns the matched text, or nil if the pattern is not found:

```lua
print ( string.match(a, '.oo.') )
print ( string.match(b, '.oo.') )
d = "2018/10/22"
print ( string.match(d, '%D%d%d%D') )
d = "2018-10-22"
print ( string.match(d, '-%d%d-') )    -- stands for something else
print ( string.match(d, '%-%d%d%-') )
```

Till now both with fixed string or with class, we were very limited because the we can match strings with a fixed length. Therefore, there are four repetition operators in programming language Lua to enable the work with any length, see the Table 8.2.

```lua
e = "ananas"
print ( string.match(e, '.+') )      -- all
print ( string.match(e, 'n.+a') )
print ( string.match(e, 'n%l+a') )
print ( string.match(e, 'n%l?a') )
print ( string.match(e, 'n%l-a') )
```

Own classes can be created by wrapping a group of characters in square brackets, too. Such class will match one of the characters.

Table 8.2 Iterators (repetition operators)

repetition operator	meaning
'*'	0–n
'+'	1–n (greedy)
'-'	1–n (non-greedy)
'?'	0–1

```
g = "barracuda"
print ( string.match(g, 'b[ar]+' ) )
print ( string.match(g, 'b[ar]*' ) )
print ( string.match(g, 'b[ar]-' ) )
print ( string.match(g, 'b[ar]?' ) )
```

```
h = "bus"
print ( string.match(h, 'b[ar]+' ) )
print ( string.match(h, 'b[ar]*' ) )
print ( string.match(h, 'b[ar]-' ) )
print ( string.match(h, 'b[ar]?' ) )
```

If the first character inside the brackets is ^, then it will match a character not in the group.

```
g = "barracuda"
print ( string.match(g, 'b[^ar]+' ) )
print ( string.match(g, 'b[^ar]*' ) )
print ( string.match(g, 'b[^ar]-' ) )
print ( string.match(g, 'b[^ar]?' ) )
h = "bus"
print ( string.match(h, 'b[^ar]+' ) )
print ( string.match(h, 'b[^ar]*' ) )
print ( string.match(h, 'b[^ar]-' ) )
print ( string.match(h, 'b[^ar]?' ) )
```

8.3.5 Captures

Captures are sequences of characters cover by a pattern item which has been enclosed in parenthesis. Captures extract parts of the given string and return them. For getting all captures, one has to use corresponding number of variables to by assinged by th returned value pro the `string.match` function.

```
date = "2022:08:10"
y, m, d = string.match(date, '(%d+):(%d+):(%d+)' )
print (d, m, y)
```

8.3.6 Substitution

The `string.gsub` function operates with three or four parameters. The first one is clear – it is a string that we will use for the substitution. The second and the third parameters specify the substring to be searched for and the string for the replacement, respectively. By default, all possible matches will be replaced unless you tell otherwise by the optional fourth parameter which specifies the number of substitutions.

```
e = "Programming language Lua"
print ( string.gsub (e, "Lua", "LUA") )
print ( string.gsub (e, "a", "A") )
print ( string.gsub (e, "a", "A", 3) )
```

A pair of values is returned, the modified string and the number of substitutions made.

Classes mentioned above can be also used here:

```
print ( string.gsub (e, "%u", "=") )
print ( string.gsub (e, "[aeiou]", "=") )
print ( string.gsub (e, "%s", "=") )
print ( string.gsub (e, "%a", "=") )
print ( string.gsub (e, ".", "=") )
print ( string.gsub (e, "%a-", "_") )
print ( string.gsub (e, "%a+", "_") )
```

8.3.7 Getting the list of matches

The last function we will talk about here – `string.gmatch` will help us for separating the input text into substring by matching the pattern. This function belongs to the group of so-called iterator functions that return instances of the pattern gradually in the `for` loop.

```
s = "Programming language Lua"
for word in string.gmatch( s , "%a+")        do print(word) end
                    -- per words
for char in string.gmatch( s , "%a")         do print(char) end
                    -- per characters
for x    in string.gmatch( s , "%a-[aeiou]") do print(x)    end
                    -- sequences ended by a vowel
d = "2021/12/31"
for x    in string.gmatch( d , "%d+") do print(x)    end
                    -- parts of a date
h = "48656c6c6f21"
for x    in string.gmatch( h , "%x%x") do print(x)    end
                    -- per hexadecimal digits
```

8.4 Other functions

The function `string.len` is an alternative to the operator #:

```
s = "Programming language Lua"
print(#s, string.len(s))
```

For `string.format`, see the Chapter 6.4.

8.5 Shortened syntax

In practice, programmers very often requires more effective ways how to write some structures. Programming language Lua offers one such tools for accessing string functions:

```
s = "Programming language Lua"
print(string.len(s))
print(s:len())
```

We can use this shortened syntax for all functions except `string.char`

```
s = "Programming language Lua"
print(s:char(101,102,103))
```

because in this case, there is no existing data to be converted, values 101, 102 and 103 are in no relation to the given string s.

8.6 UTF-8 encoding

The Lua programming language offers basic support for utf8-encoded strings. This basic support consists of six functions that deal with multibyte characters:

```
utf8.char (···)  -- converts byte sequence to a corresponding UTF-8 character
utf8.charpattern -- matches exactly one UTF-8 byte sequences
utf8.codes (s)   -- iterates over all characters in string s
                      for beginnings and ends of UTF-8 characters
utf8.codepoint (s [, i [, j]])
                 -- returns the codepoints (as integers) from all characters in s
                 -- that start between byte position i and j (both included).
utf8.len (s [, i [, j]])
                 -- returns the number of UTF-8 characters in string s,
                 -- specification of a subpart for searching with i and j is possible
utf8.offset (s, n [, i])
                 -- returns the position (in bytes) where the -encoding of the
                 -- n-th character of s (counting from position i) starts
```

Questions

(1) What does ‘immutability’ stand for?

Exercises

(2) Write patterns which check whether the input line contains:
   (a) name and surname only.
   (b) file name (letter, dot, 3 letters, only letters or digits) only,
   (c) price (number plus 3-character-long abbreviation of a currency).

(3) What will you get into the variable result?

```
e = "Programming language Lua"
result = ((e:gsub ("Prog", "B")
             ):gsub("amm","i" )):gsub("ing","lliant")
```

(4) Get the length of the specified character string.

(5) Print a list of words from the given string. (Assume that words are separated by spaces.)

(6) Prepare the list of words from the given file (read them from the standard input).

(7) Prepare a frequency table of words from the given file (read them from the standard input).

9 Tables

Simple data types are intended to describe one data (one number, one string, one logical value). In practice, however, there are many situations where an object is described by several pieces of data.

Consider, for example, an automobile. For each car we want to specify the manufacturer (factory brand), the type, the engine capacity v cm^3, color, number of axles, number of doors, etc. Each characteristic separately can be described by some suitable simple data type: the manufacturer will be of type string, engine capacity of type integer number, etc.

Description of the whole car by these simple data types will not be acceptable, because for we have to declare a completely separate variable for each property, isolated from the other variables describing other properties.

Similarly, we can imagine a chessboard of size $8 times 8$ squares. If we wanted to create a chessboard with 64 squares, we would have to create 64 variables that would be described by a simple data type. There is, as in the previous case, no binding between them. We have at our disposal 64 separate, isolated variables.

That is why programming languages introduce so-called structured data types, **structured data types**. For example, vector (one-dimensional array), matrix (multidimensional array), record (struct), often implemented by an associative array, or various types of sets. Structured data types differ from each other both in their structure and the operations that can be performed.

In Lua, as a consequence of minimizing the overhead, which we have already mentioned in the Introduction, there is only one structured data type called `table`. And it is up to us how we use it, i.e. what it represents for us.

To be precise, the structured data types also include data type file. However, this is operating system and hardware dependent. Mainly, however, we do not access the data in the file directly, but through services provided by the operating system. This makes files very different, which is why we devote a separate chapter to them.

9.1 Table initialisation – constructor

A table as a complex and sometimes relatively complex data structure cannot be created just like that 'by itself'. Practically, a variable of type table is not stored directly in memory, but contains a reference to other memory location where the table's own data is located. This reference is called a pointer and is actually the address of that other memory location.

The space reservation (allocation) and its address are stored in a variable of type table, when we use the so-called table constructor. If we want to create an empty table, we use this notation:

```
t={}
```

9.2 Table as an 1D array (vector)

The following two code snippets do the same thing – they both create a table and insert into it the first few values into the table.

```
t={}

t[1]=1  t[2]=2   t[3]=7
t[4]=5  t[5]=13  t[6]=-1

--

t={1,2,7,5,13,-1}
t={1,2,7,5,13,-1,}
```

This constructor also contains values. They are separated by a comma. Test that an extra comma after the last value does not cause any error.

All values are of the same data type, so we are talking about a homogeneous field. As the example shows, the individual components of the array (table) are accessed by appending a more precise determination after the table identifier, that is, an index into that array. All kinds of arrays where values are inserted in this way 'one another' and can be accessed by numeric indices, we also call them indexed arrays.

```
t={1,2,7,"Lua",true,{},-9.9999,false,8888}
```

This example of the constructor shows a case where the values are not of the same data type, there are a mixture of numbers, strings, booleans and, moreover, another table constructor is present, too.

These types of arrays (tables) are called heterogeneous arrays and can also be indexed.

9.3 Table: insert, remove, number of elements, list of elements

The recommended method to insert additional values into the table, or to remove values from the table are the `insert` and `remove` functions. The first parameter is always the table. The value is always inserted at the end of the table, in the case of removals, also from the end of the table. The second parameter is the inserted value.

The following code also shows a direct insert to the next position in the table via the index, which was calculated by the expression *last used index (#t) + 1*.

```
t = { 5, 10, 20, 30, 60 }
print (#t)      -- 5
table.insert(t,88)
print (#t)      -- 6
t[#t+1]=99
print (#t)      -- 7
```

Remove any element from the table with the `remove` function:

```
table.remove(t)
```

Naturally, after the many manipulations with the table, we are interested in what we actually have stored in table t. Since it is an indexed array, we can use the following loop:

```
for i=1,#t do print (t[i]) end
```

And what if we want to insert the data in a different position than the last one? Just use the extended entry of the insertion subroutine, where we also specify the position. This is given as the second parameter and the value to be inserted becomes the third parameter.

Similarly, modify the entry for the subroutine for removing – add the position from which we want to remove.

The advantage of using these subroutines is that they automatically take care of reindexing the remaining elements of the table.

```
table.insert(t,3,777)
table.insert(t,1,111)
table.remove(t,2)
```

And now we can list the table again in the familiar way.

9.4 Solved exercises

Determine the average of values in a table.

```
local a = {5, 3, 2, -1, 9}
local sum = 0
for i=1,#a do
  sum = sum + a[i]
end
print (sum/#a)
```

For avoiding division by zero at (unwanted) empty table, we can print out the result only conditionally:

```
if #a>0 then print (sum/#a) end
```

Find the minimum value in a table. In this example, we will use the more general way which is not restricted only an interval <1 ; n> as was used in the previous example.

```
local t = {1, 3, 7, 6, 4, 0}
local index, maxvalue = 1, t[1]

local key, maxvalue = 1, t[1]      -- temporary maximum
for k, v in ipairs(t) do
    if t[k] > maxvalue
       then key, maxvalue = k, v
    end
end

print(key, maxvalue)
```

This algorithm will keep the original order of values. If it is not necessary, one can use also this trick:

```
local t = {1, 3, 7, 6, 4, 0}
table.sort(t)
print(t[1])
```

9.5 Sorting

There are a number of methods for sorting values and also quite extensive theory. We will skip the theoretical analysis at this point because the sorting algorithm in the pre-prepared sort subroutine is very efficient. The following code snippet shows how to simply write the sort of the indexed data. However, data homogeneity is a prerequisite, so it is not possible to sort numbers and strings at the same time.

```
t = { 1, 3, 5, 2, -3, 8, 0, -11 }
table.sort(t)
```

The sort subroutine makes it very easy to change the sorting algorithm. If we want to sort in descending order (not ascending as it is normally), we add a second parameter. This parameter will be of function type. And it does not matter whether we will write the body of the function directly into the parameter, or whether we will declare the function first and then just put its identifier here. So both of the following examples lead to the same result.

```
t = { 1, 3, 5, 2, -3, 8, 0, -11 }
table.sort(t, function (a,b) return b<a end )
```

Písmeny `a` a `b` symbolicky označuje dvě hodnoty, které vstupují do porovnání, jež součástí každého řadicího algoritmu.

```
function myf (a,b)
  return b<a
end

table.sort(t,myf)

for i=1,#t do print (t[i]) end
```

We will always prefer the latter way if the same method of comparison of values is used in multiple places in our program or if the comparison algorithm is so complex that it would make the program code very cluttered.

9.6 Table as a hash

The associative field can be understood as a structure that carries within itself information about the meaning of values. In short, this approach is often called *key-value* system. In practice, this means that each value is named.

```
t={}

t["jan"]=31
t["feb"]=28
t["mar"]=31

t.jan=31
t.feb=28
t.mar=31
```

The example shows that the Lua programming language allows two approaches to values – first, we can use keys as strings as 'indexes' in a table, and the key becomes an identifier that we append to the table name with a period. Both notations have the same meaning.

Now we would like to take another look at what is in this table. Commands

```
print (t)
for i=1,#1 do print(t[i]) end
```

will not satisfy us – the first one returns a response of type `table: 0x560b808666f0`, which only indicates that the table will be present somewhere in memory, and the second one does not provide any response. This is because writing `#t` here will return zero (the for loop cannot run even once), which is the number of items indexed by *numbers*. The items of the associative array accessed via keys are included in this count are not included.

We have to use a different method:

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

In this way, we encounter a variant of the for cycle commands, which acquires non-numeric values, but values from the list created by the pairs function. Here, by pair we mean just a key-value pair, and by list we mean a sequence of these pairs. Thus, each pass through the loop processes one pair.

If you run this loop multiple times in succession for the same values, it may be surprising for you that the order of the keys is different each time. Therefore, the order of of the values in the associative array will never be relied upon.

If we want to get a predetermined order of keys – a typical example is alphabetically ordered list – we need to prepare the order of the keys in advance.

We will show it in a zoo task, which for each kind of animal records the number of pieces.

```lua
local zoo={}

zoo["gepard"]=3
zoo["lynx"]=6
zoo["parrot"]=28
zoo["horse"]=14
zoo["eagle"]=6

local keys = {}
 for k,_ in pairs(zoo) do
   table.insert(keys,k)
 end
 table.sort(keys)

 for i=1,#keys do
   print (keys[i],zoo[keys[i]])
 end
```

The first loop goes through the table with the animals and keys, stores in the keys table. The pairs function returns a pair of values, but the associated value we don't need for further work. This unneeded value is stored in a variable called _, which is a perfectly normal variable (it could easily be v), but in practice it is common to use _ to indicate that this value is not processed further.

The second loop then loops through the sorted key table and selects from table of animals.

Note the double indexing (`zoo[keys[i]]`).

9.7 Searching in an array

A very common activity when working with tables is to find a value. Assume that the table now behaves like an array, i.e. it has a set of values stored in it. As an idea, we can use the list of pupils in this classroom, to which the teacher gradually adds individual names as the pupils arrive:

A very common activity when working with tables is to look up a value. Suppose now that the table behaves like an array, i.e. it has a set of values stored in it. As an idea, we can use the list of pupils in this classroom, to which the teacher gradually adds individual names as pupils arrive.

```
class = { "Bob", "Paul", "Amy", "John", "Susan", "Roger", "Audrey" }
```

Now we are interested if Amy came to the class as the second (third, ...) person.

```
print (class[2] == "Amy")  -- false => Amy was not the second
print (class[3] == "Amy")  -- true  => Amy was the third
```

If we do not know the number of Amy's arrival, we how to iterate through all values:

```
local i = 0
local who = "Amy"
repeat
  i = i + 1
  print(i,class[i],class[i]==who)
until class[i] == who
print(who,i)                -- Amy's order
print(class[i] == who)      -- binary answer: Amy is/is not here.
```

```
1   Bob     false
2   Paul    false
3   Amy     true
```

The first function `print` will show how the loop will work. First two runs did not find Amy. Therefore, the condition was evaluated as false. The third rum was succesfull, the loop ends up and the second print function gives us the result.

However, the snippet has red border which indicates that there is something wrong with this code. Are we able to answer if Eve came too? Let's assign "Eve" to who instead of "Amy" and execute the code once more.

Problably you will have interrupt the run pressing `Ctrl-C`.

To resolve this problem, we have to add one more condition:

```
repeat
  ...
until i==#class or class[i] == who
print(class[i] == who)      -- binary answer: Eve is/is not here.
```

The additional condition stops the loop after checking the whole array, regardless of whether she came or not.

9.8 Conversion array to hash

After changing the organization of the data structure of the table from an array to a hash, there must also be a change in the procedures leading to finding out whether the student is present or not, or what is the order of his/her arrival in class.

First, let's see how to convert an array to a hash:

```
hashclass = {}
for i=1,#class do
  hashclass[class[i]] = 1
end
```

Nyní se stačí zeptat takto:

```
print (hashclass["Amy"] == 1)    -- true,  value 1 found at Amy
print (hashclass["Eve"] == 1)    -- false, no value for missing student
```

However, we can simplify it using the operators `and` and `or`.

```
who = "Amy"
print (hashclass[who] and "PRESENT" or "missing")
who = "Eve"
print (hashclass[who] and "PRESENT" or "missing")
```

If the student is in the classroom, hashclass[who] must exist. Therefore, the its value exists too (evaluated as `true`).

For missing students, no information will be found, so the first part will be evaluated as `false`. This will cause that the part `and "PRESENT"` will be skipped and the value `missing` becomes as a result of this expression.

We can assign the value of `i` instead of value `1` to the given name. So that means that this solution will store the order of students' arrival in the class:

```
hashclass = {}
for i=1,#class do
  hashclass[class[i]] = i
end
```

Now we are able to answer both questions:

```
who = "Amy"
print (hashclass[who] and ("PRESENT: "..hashclass[who]) or "missing")
who = "Eve"
print (hashclass[who] and ("PRESENT: "..hashclass[who]) or "missing")
```

9.9 Array and hash together

```
t = { 10, 20, 30, 40, 50 }
t.name = "Challenge"
t.type = "yacht"
t.year = 2000
```

The table data type allows storing both indexed and associated values at the same time. We can create such an array quite easily: However, we have to write the values in two goes:

```
-- indexed part
for i   = 1,#t     do print (i, t[i]) end
-- key-value part
for k,v in pairs(t) do print (k, v) end
```

9.10 Table as an 2D array (matrix)

9.10.1 Initialization, constructor

We have already met the one-dimensional field in the preceding text. Now let's see how to create a two-dimensional array. Let's start with an example that shows a certain, often occurring error of carelessness.

```
t = {}
t[1][1] = 1
t[1][2] = 2    -- etc.
```

After introducing the table with the constructor, we started trying to assign to certain positions in the matrix. It is possible to assign a number to `t[1]` but not a table (the second dimension is also a table! which must be initialized before) as indicated by the error message `attempt to index field '?' (a nil value)`.

```
print(type(t), type(t[1]))
```

So the right ways are:

```
t = {}
t[1] = t[1] or {}          -- if we need initialization on the fly
for i=1,5 do t[i] = {} end  -- if we know number of rows in advance

t = { {}, {}, {}, ... }     -- empty 2D-array
t = { {  11, 12, 13}, { 21, 22, 23 }, { 31, 32, 33 } }
                            -- if we know even values in advance
```

9.10.2 Displaying values of a two-dimensional array

For displaying values of a two-dimensional array we need two nested loops for:

```
for i = 1, #t    do
  for j = 1, #t[i] do
    io.write( t[i][j], "\t" )
  end
  io.write("\n")
end
```

Here we have used simplified formatting using tab (\t) and line breaks (\n), it is of course better to use the `string.format` function and adapt the listing to our needs.

9.10.3 Algorithms for matrices

Transposition of an array means swapping rows and columns.

```
tm = {}
for i=1,#t[1] do tm[i] = {} end

for i = 1, #t    do
  for j = 1, #t[i] do
    tm[j][i] = t[i][j]
  end
end
```

Then print out the matrix using the algorithm above.

Print the values from on the main diagonal of this matrix. Here we will need only one loop because the main diagonal is a vector, not a matrix:

```
for i=1,#t do
  print (t[i][i])
end
```

Display the value lying on the next line of the same column. Assume coordinates r and c for the row and the column, respectively.

```
print (t[x]  [y])  -- this value
print (t[x+1][y])  -- value from the next line
```

The second line of this code will produce an error if x is equal to the last index. (x+1 will not exist.)

```
if t[x+1] then print (t[x+1][y]) end   -- conditional print #1
print (t[x+1] and t[x+1][y] or "n/a")  -- conditional print #2
```

9.10.4 A more general way to display values of a table

We gradually work with more and more complex table structures. With this, the complexity of writing out the table also increases, which complicates life when debugging programs. It would require creating a tool that could list any table regardless of its internal structure. This tool is presented in Chapter 11.3, so we recommend that you read that chapter now. The tool is called myinspect and it is a function whose parameter is the table identifier. Its contents will be written to standard output. Generally, the process of creating this is called table serialization. On the Internet, you can find other, similar functions designed for the same purpose.

9.11 Table as a set

Typical operations in a set are the union, intersection and difference of two sets and finding the cardinality of a set. Next, we will need to create an empty set, then add and remove an element, and check if the element is in the set.

We will also implement the set using a table. We first have to create a set:

```
set = {}                 -- empty set
set = {[3]=true, [4]=true, [8]=true}      -- set with first three elements
```

The second method is somewhat more demanding to write, so it is only worthwhile for a smaller number of elements with which we initialize the set.

We can verify the content of the set by listing it.

```
for k,v in pairs(set) do print (k, v) end
for k   in pairs(set) do print (k)    end   -- v is not necessary here
```

The following two command will not live to our expectations. The first of them will not produce anything because the value `#set` is not defined, we do not work with this table as with typical array. And the second one? It will dislay only a part of the set so we have to be sure that all values belong to the interval `<1; 10>`. Moreover, it will display plenty of elements which we are not interested in.

```
for i=1,#set do print (i, set[i]) end
for i=1,10 do print (i, set[i]) end
```

If we want to add another element to this set, it is enough to write: Removing an element from the set here means removing the corresponding element from the table, which we ensure by setting the value `nil` to the corresponding index in the table.

```
set[3]=true
set[5]=true
set[4]=nil
```

For the next three operations, assume two existing sets:

```
set1  = {[3]=true, [4]=true, [8]=true}
set2  = {[1]=true, [4]=true, [12]=true, [15]=true}
```

Union:

```lua
new = {}
for k in pairs(set1) do new[k] = true end
for k in pairs(set2) do new[k] = true end
```

Intersection:

```lua
new = {}
for k in pairs(set1) do new[k] = set2[k] end
```

Difference:

```lua
new = {}
for k in pairs(set1) do
  if set1[k] and not set2[k] then new[k] = true end
end
```

Cardinality:

```lua
c = 0
for k in pairs(set1) do c = c + 1 end
print(c)
```

10 Functions (subroutines)

Every well-written program should be clear. One possibility, to increase its clarity is to separate out some of the separate parts into separate sections of the program, called **subroutines**, whose purpose is to perform a predefined action.

In the Lua programming language, we have only one dedicated word for for declaring subroutines – function. Thus, in terms of syntax, we are talking only about functions.

However, in terms of semantics, we can distinguish two common, from many other programming languages that are known to use subroutines – use as procedures that 'only' perform a certain action, and use as a function, i.e., after performing an action, usually a computation the function passes the result to the parent structure (we say that the function returns value).

These are usually named sections (we will discuss one exception later), which means that each function has its own name – an identifier, which may be followed by parameters in round brackets.

10.1 Declaration and parameters

Each subroutine must first be declared, i.e. the chosen name is assigned a certain sequence of commands. The use of a subroutine is called **subroutine call**, or in the case of a procedure, **command procedure**.

```
function print_two_numbers (number1, number2)
   print ("Two numbers: ",number1, number2)
end

print_two_numbers (10,99)
```

The function name, i.e. the function identifier, is therefore `Pi`. The sequence of statements within the function will consist of one statement that assigns the function identifier to the resulting value.

```
function sum_of_two_number (number1, number2)
   local sum = number1 + number2
   return sum
end

a = sum_of_two_number (10, 99)
print (a)             -- or
print (sum_of_two_number (10, 99))
```

10.2 Recursion and recursive functions

The command part of the subroutine can contain calls to other subroutines, either procedures or functions. Special case, where a subroutine calls itself is called a **recursive call** or **recursion**, or **recursion!direct**.

For **indirect recursion** we need at least two subroutines, e.g. R and S. The subroutine R calls the subroutine S and then in its command part calls again subroutine R.

We can use recursion effectively to solve problems whose algorithm is itself recursively defined, such as the factorial or the greatest common divisor (GCD).

$$n! = \begin{cases} \dfrac{1, \text{ if } n = 0}{n \cdot (n-1)! \text{ for } n > 0} \end{cases}$$

GCD(a,b) =

a, je-li $a = b$

GCD($a - b$,b), je-li $a > b$

GCD(a,$b - a$), je-li $a < b$

Note the important fact that in every definition of a recursive algorithm a specific value (1) is prescribed for a well-defined situation (n=0). This property causes the algorithm to be finite.

```
function GCD(X, Y)
if x==y then return =x
        else if x>y then return GCD(x-y,y)
            else return GCD(x,y-x)
  end

cislo1, cislo2 = io.read("*n", "*n")
print(GCD(cislo1, cislo2)
```

GCD is a function, so there must be an assignment statement to assign a value to the GCD identifier. The greatest common divisor (see above) is defined for three situations (a=<>b) differently, and so the function must assign values in each variant; this is why the assignment statement is used three times.

Calculating the greatest common divisor by the recursive algorithm is not very convenient. Each time the function `GCD` is called, the takes up memory unnecessarily. In general, problems that can be solved by both recursion and looping, are less time- and space-consuming if we use a solution without recursion.

For recursive solving, problems that use the **system stack**, which is a memory that contains the local variables of the called (a so far terminated) subroutines. An example of such a task might be a program that takes input strings as output the strings in the reverse order from that in the input file.

10.3 Function as a data type

The subroutine data type allows the activity of another subroutine to be influenced in a certain way by various other subroutines with corresponding parameters during the program's runtime.

The example that shows this tools is given at `table.sort` function.

10.4 Iterators and closures

At one website[4] I found nice explanation of closures: *'Closures are hard to describe. But [...] "You know it when you see it."'*

In simple words, a closure is a function inside a function where the inner function can see local variables of the outer function. Closures can be used for a variety of powerful features. For us, the most important point is use within so-called iterators.

An iterator is any construction, in programming language Lua typically a function, that iterates over the elements (in Lua in a table). Each time we call that function, it returns one of the elements from the given table.

In the previous chapter dealing with tables we met functions `pairs`, `ipairs` and `lines` (this one will be used also in the Chapter 14). They iterate over key-value pairs and lines, respectively, and these values are gradually processed by `for` loop.

10.4.1 User iterators

The principle will be shown on user iteration function which will produce values from even indices of a table.

[4] http://www.troubleshooters.com/codecorn/lua/luaclosures.htm

```lua
function only_at_even_indices(t)
  local i, n = 0, #t
  return function ()
           i = i + 2
           if (i <= n) then return t[i] end
         end
end
```

Then we can process a table:

```lua
t={21,12,3,4,15,29,12}
for value in only_at_even_indices (t) do print(value) end
```

This function returs only values. If we need only the corresponding indices, we change the return command:

```lua
function only_even_pairs(t)
  local i, n = 0, #t
  return function ()
           i = i + 2
           if (i <= n) then return i, t[i] end
         end
end
```

Then we can process a table:

```lua
t={21,12,3,4,15,29,12}
for index, value in even_pairs(t) do print(index, value) end
```

Questions

(1) What is a subroutine?
(2) What are subroutine parameters used for?
(3) How do we distinguish subroutines in terms of semantics?
(4) What is structured programming?
(5) What are subroutine parameters and what are they used for? %cvicitem What is a function call?
(6) What must a function contain within its statement part in order to the result of a computation to be used at the location of the function call?
(7) What are recursive subroutines?
(8) How do we ensure that a recursive subroutine is not infinite?

Exercises

(9) Write a procedure that prints a series of asterisks. Number of stars is given as a parameter to the procedure.
(10) The function `signum` is defined so that for positive values of the parameter, it gives a value of 1, for negative −1, and for zero it gives 0. Build this subroutine and use it.
(11) Create a function that computes the following in a Cartesian coordinate system the distance between two points. The coordinates of these two points are the parameters of the function.
(12) Use the function constructed in the previous problem in a program that obtains the coordinates of the vertices of a triangle from the input file and calculates its perimeter.

For these exercises, all input values will be used as parameters of functions.

Some of the problems are very similar to some of the problems in the previous chapters. Feel free to use these solved problems here.

Questions

(13) Create a Lua function to calculate the root of the linear equation $ax + b = 0$ (i.e. the value of x).
(14) Create a Lua function to determine the minimum of two given numbers.
(15) Create a Lua function to find the absolute values of the difference of two given numbers.
(16) Create a Lua function to determine whether or not a given number is even.
(17) Create a Lua function to determine whether a given number is positive, negative, or zero.
(18) Create a Lua function to calculate the perimeter of a triangle given by the lengths of its three sides.
(19) Create a function in Lua to calculate the perimeter of a right-angled triangle given by given by the lengths of its sides.
(20) Create a Lua function to determine whether a triangle given by its lengths is of its sides is equilateral, isosceles, or rectangular.
(21) Create a Lua function to retrieve three values and list them in ascending order (listing from smallest to largest).
(22) Create a Lua function to calculate the roots of a quadratic equation based on given coefficients a,b,c, where $ax^2 + bx + c = 0$.

11 Modules (libraries)

A library is usually an isolated piece of code that contains of code that already contains separate subroutines. The purpose of libraries is to speed up the creation of programs, since many subroutines can be used in a variety of programs. The creator of these libraries creates his subroutines once and for all and then just uses them. Libraries can be divided into two types according to their origin:

standard libraries – They are created by the creator of the language implementation, the subroutines contained in them are suitably supplement the basic resources. Declared variables Variables and subroutines are listed in the documentation.

user libraries – The creator can be anyone. User libraries usually cover specific needs of the programmer.

11.1 How to create own library

Unlike most programming languages, where libraries are a three-part file (eg. Pascal) – interface, implementation and initialization parts and sometimes (e.g. in C++) even with a split into two files, is to create a library in Lua very simple – it is a single file containing both the declarations functions, as well as any initialization (execution) part.

Since the interface is not a separate part here, it is necessary to have the in mind that all subroutines and data structures are accessible from other libraries or from the main program, i.e. global. If we do not want to make some library elements accessible, we must mark them as local, using the reserved word `local`.

The following code shows the simple library for some triangle computations. The whole has to be saved as a separate file (let's call it `mytriangle.lua`):

```lua
local mt = {}                               -- mt = mytriangle
function mt.circumference (a, b, c)
  return a+b+c
end
function mt.area (a, b, c)                  -- Heron's formula
  local s = mt.circumference(a,b,c) / 2
  return math.sqrt(s*(s-a)*(s-b)*(s-c))
end

return mt                                   -- important!
```

This is the second way how to prepare a library:

```
local mt = {}                              -- mt = mytriangle
mt.circumference = function (a, b, c)
  return a+b+c
end
mt.area = function (a, b, c)               -- Heron's formula
  local s = mt.circumference(a,b,c) / 2
  return math.sqrt(s*(s-a)*(s-b)*(s-c))
end

return mt                                  -- important!
```

11.2 Joining the library

The library will be joined by the function `require` followed by the library filename (without the extension `.lua`:

```
local m = require "mytriangle"
io.read(a,b,c)
print(m.circumference(a,b,c), m.area(a,b,c))
```

Questions

(1) What is a Lua library?
(2) What can a library contain?
(3) What is the difference between a standard library and a user library?

Exercises

(4) Add to the library `mytriangle` (a) calculation of the area of the triangle using the side and the corresponding height, (b) calculation of the area of a triangle using two sides and an internal angle.
(5) Build a library of math functions that are not
(6) Build a library of mathematical functions that are not not implemented in Lua, e.g. tangent, arc sine, arc cosine, decadic logarithm, third root, *n*-th power, etc.

11.3 Inspect

```lua
local output, indent, width = io.write, 0, 3

local fout      = function (x) output (tostring(x)..",\n") end
local fempty    = function ( )                        end
local fboolean  = function (x) fout (x)               end
local ffunction = function (x) fout ("function")      end
local fnil      = function ( ) fout ("nil")           end
local fnumber   = function (x) fout (x)               end
local fstring   = function (x) fout ('"'..x..'"')     end

local typelist = {
   ["string"]   = fstring,     ["nil"]      = fnil,
   ["number"]   = fnumber,     ["function"] = ffunction,
   ["boolean"]  = fboolean,
}

myinspect = function  ( d )
  local typed = type(d)
  local fprint = typelist[typed] or fempty
  if typed == "table"
    then
      if indent == 0 then output("table = ") end    output("{\n")
      indent = indent + width
      for k,v in pairs(d) do
        output ((' '):rep(indent) .. '["' .. k .. '"] = ')
        myinspect(v)
      end
      indent = indent - width
      output ((' '):rep(indent) .. " },\n")
    else
      fprint(d)
  end
end

return myinspect
```

12 Abstract Data Types

By the term **abstract data type** (ADT) we mean a set of data types (values) and operations associated with them, which are precisely specified independently of a specific implementation. Using ADT, we usually create a model of a more complex data type close to reality, but for the implementation of which we do not have direct tools in the given programming language. However, we use the existing tools in the given programming language to implement the ADT. The advantages of ADT are that:

- ADT is determined by what we want/need in it.
- ADT can be implemented in different ways without affecting its behaviour.
- ADT is implemented using a suitable data structure.

To describe ADT, we can use a graphic method – a signature diagram, an axiomatic method, or we can also express it in a programming way in the form of an ADT user interface.

12.1 Axiomatic description for ADT Queue

Assume that we will need the following five operations (functions, methods) for ADT Queue:

```
init(_) :       --> queue
count(_): queue --> number
empty(_): queue --> boolean
put(_,_): queue, data --> queue
get(_):   queue --> data
```

The parenthesis describe number of parameters, to the left of the arrow we have data types of these parameters and to the right we see the data type of the return value for a particular method.

12.2 Implementation of the ADT Queue

The following code is the content of the new module file, eg. module_queue.lua. For more comfortable work, the print function has been added:

```
local Q = {}

Q.init  = function (t)
              local q = {}
              for _, l in ipairs(t) do table.insert(q,l) end
              return q
          end
Q.put   = function (t,e) table.insert(t,e) end
Q.get   = function (t)
              local e = table.remove(t,1)
              return e
          end
Q.count = function (t) return #t end
Q.empty = function (t) return #t==0 end
Q.print = function (t)
               for i=1,#t do io.write(t[i]," ") end
               print ()
          end

return Q
```

As you can see, we implement the queue using the data type `table` and we used the tools that programming language Lua provides for working with tables to the maximum extent.

12.3 Interface for a user

Interface for a user can be derived from the axiomatic description as well as from the implementation – we used function headings:

```
queue   = Q.init  (queue)
          Q.put   (queue, data)
data    = Q.get   (queue)
number  = Q.count (queue)
boolean = Q.empty (queue)
```

Then we can use it in our main program:

```
Q = require "module_queue"

myqueue = Q.init{}              -- empty queue or
myqueue = Q.init{"John"}        -- John was standing at the counter before it opened.

Q.put(myqueue,"Susan")
Q.put(myqueue,"Paul")
Q.put(myqueue,"Mark")
Q.print(myqueue)                -- Who is in the queue?
print(Q.get(myqueue,"Mark"))    -- First customer served.
print(Q.count(myqueue))         -- Number of pending customers' requests
print(Q.empty(myqueue))         -- Can I close the counter now?
```

12.4 Module for set operations

The following code follows the code from the previous chapter dealing with tables. It contains the adapted code for union, intersection, difference and cardinality. All of them are written as functions and the whole must be stored as independent module file (eg. module_set.lua).

```lua
local Set = {}

function Set.new (init)
  local set = {}
  for _, element in ipairs(init) do set[element] = true end
  return set
end

function Set.union (set1, set2)
  local newset = Set.new{}
  for element in pairs(a) do newset[element] = true end
  for element in pairs(b) do newset[element] = true end
  return newset
end

function Set.intersection (set1, set2)
  local newset = Set.new{}
  for element in pairs(set1) do newset[element] = set2[element] end
  return newset
end

function Set.difference (set1, set2)
  local newset = Set.new{}
  for element in pairs(set1) do
    if set1[element] and not set2[element]
      then new[element] = true
    end
  end
  return newset
end

function Set.cardinality (set)
  local c = 0
  for k in pairs(set) do c = c + 1 end
  return c
end

return Set
```

And use of the module in our main program follows:

```lua
S = require "module_set"

a = S.new{1,2,4}
b = S.new{10,2,40}
c = S.union(a,b)

for k in pairs(c) do
  print (k)
end
```

12.5 Module for set operations with a metatable

Metatables allow us to change the behaviour of a table. The use of metatables will be shown here, with sets but it is a general way how to simplify writing operations that have been implemented using tables.

From the operators point of view this expansion, ie. adding additional properties is called **operator overloading**.

In Lua, there are three steps to be done now:

(1) We have to create new and empty metatable table.
(2) Then we have activate the metatable mechanism.
(3) And finally, we have to assign functions that have been implemented for particular set operations to the corresponding operators.

```
...
Set.mt = {}                   -- step 1
...
function Set.new (init)
  local set = {}
  setmetatable(set, Set.mt)   -- step 2
  for _, element in ipairs(init) do set[element] = true end
  return set
end
...                           -- step 3
Set.mt.__add = Set.union
Set.mt.__mul = Set.intersection
Set.mt.__sub = Set.difference
...
return Set
```

In the main program, we can now use operator +, * and - for union, intersection and difference of sets, respectively:

```
s1 = Set.new{5,9,13,17}
s2 = Set.new{1,13,21}

su = s1 + s2
sd = s1 * s2
si = s1 - s2
```

Isn’t it beautiful?

13 Files

The term **file** refers to the portion of disk space that containing certain data. Regardless of the type of device, such as hard disks, floppy disks, virtual disks or CD-ROMs, the files are accessed in the same way in a programming language environment. The actual actions are handled by the operating system used, and we do not have to deal with the properties of the data carrier used when writing programs.

Each file is marked with a name within the disk space and the access path. These two pieces of information ensure that the file is uniquely identified.

In terms of file processing, we can classify files according to various criteria.

(1) According to the use of control characters:
   - text files,
   - non-text files with the specified data type,
   - non-text files with no type specified.

   We consider text files as character files in that the file is internally organized internally into lines. The end of the lines is marked with the agreed control characters.

(2) Depending on the type of file handling:
   - read-only files,
   - write-only files,
   - both read and write files.

(3) Depending on how the data in the file is processed:
   - files processed sequentially,
   - files with direct access.

This main division is general and does not depend on the programming language used. We have to keep in mind that the physical, real form of the file is always the same and that the above division refers to the exclusively our work with the file, our our logical approach to the file. Now let's briefly explain what is meant by each criterion.

13.1 Difference between text and binary files

If we decide to consider file as **text**, it means that the information inside should be understood as follows:

- Characters whose ordinal value is less than 32 are treated as control char-

acters, i.e. a predefined action is performed These characters may not be displayed in the output.
- Characters whose ordinal value is 32 or greater are the carriers of of textual information.

The **files** are the opposite of text files, i.e., they are processed a non-text file, characters with ordinal value are not considered control characters, but are treated as any other data.

Binary files are further divided into two groups according to the data stored. If all the data is of the same type, e.g. only numbers `real` or just characters, or just records, etc., then we speak of the **files with the specified data type**.

Binary files without a specified data type may contain data of different data types, i.e. a mixture of numbers, characters, records, sets, strings, etc. In this case, it must be known which data types and in which order they should appear in the file.

13.2 Access to a file

By the declaration of a data file variable we must imagine something other than the variables of all other data types. The declaration does not create a file in operational memory, but a data structure that contains all the information about the file that is needed to to communicate with the operating system. The actual file, of course, still resides on external, usually disk, memory.

The created data structure, or a pointer to it, is passed as a return value of the function `io.open` and we store it in the variable `f`:

```lua
local f = io.open(filename, "r")

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

13.3 Open

Information on how to open the file is also part of the file disclosure. The letter `"r"`, which we have used here, means open for reading. An overview of the other ways of opening the file, i.e. the ways we will process the data, is given in table 13.1.

Open a file operation is the action that, when successfully executed the operating system allows us to continue working with the file and its contents.

A read-only file allows only data retrieval. A file open for writing allows only the storage of data. A file that is open for both activities allows data to be retrieved and storage.

13.4 Close

Close a file is the action we communicate to the operating system, that the file will no longer be used.

We perform file opening and closing with each file regardless, whether we process it textually or non-textually, regardless of whether the file or read-only or write-only or both.

13.5 Data reading methods

We always treat text files as read-only files or for writing, not both. In contrast, non-text files can be handled in all three ways. Data processing in a file is done either by **processing** (sequential), or **direct access**.

Sequential processing means that the data is processed from file are read sequentially in the order in which they are written. When reading, it is not possible to to go back or skip elsewhere. Sequential file processing is used with text files (for these it is only option), and sometimes for other file types.

Direct access to the data in a file means that each folder in the file has its own serial number. We can use this number to jump to a particular folder. We use this method when working with binary files.

Table 13.1 Modes for opening files

Mode	Description
"r"	Read-only mode and is the default mode where an existing file is opened.
"w"	Write enabled mode that overwrites the existing file or creates a new file.
"a"	Append mode that opens an existing file or creates a new file for appending.
"r+"	Read and write mode for an existing file.
"w+"	All existing data is removed if file exists or new file is created with read write permissions.
"a+"	Append mode with read mode enabled that opens an existing file or creates a new file.

The io.read function reads strings from the current input file. Its arguments control what is read:

Table 13.2 Modes for reading values from a file

Mode	Shortcut	Description
"*all"	"*a"	reads the whole file
"*line"	"*l"	reads the next line
"*number"	"*n"	reads a number (including leading whitespaces)
num		reads a string, its length is determined by the *num* values

13.6 Processing of text files – solved examples

Create a frequency table of the words found in the given text file.

```lua
local f = io.open("a.a","r")
local words = f:read("*a")

t = {}
for w in words:gmatch("%a+") do
  t[w]= (t[w] or 0) + 1
end

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

Reading mode `*a` joined the whole file into one very long line (`words`). Then, using `words:gmatch`, we are extracting words (=sequences of letter; `%a+`). For each word, its own counter is incremented by 1.

When we get a word for the first time, its counter does not exist yet (`t[w]` is nil, ie. false). Therefore, the expression `t[w] or 0` has been used and for non-existing counter it works with value `0`.

For counting lines in a text file, there no built-in tool in Lua. We can use function `io.lines` (so-called line iterator) for gradual getting all lines. The function `io.lines` will work without explicit file opening:

```lua
local n = 0
for _ in io.lines'yourfile.txt' do
  n = n + 1
end
print(n)
```

The air conditioning in the apple warehouse can be in four states: heating (0), tempering (10), cooling (110), off (111). Each change is recorded in a text file. The information was recorded in text form on a file. Find out how many times the tempering condition occurred during the monitored period:

```lua
local codes = { ["0"]   = 0,  ["10"]  = 0,
                ["110"] = 0,  ["111"] = 0,  }

local code = ""
local f = io.open("apples.txt","r")
local b = f:read(1)

while b do
  code = code .. b
  if codes[code] then
    codes[code] = codes[code] + 1
    code = ""
  end
  b = f:read(1)
end

print(codes["10"])
```

For testing the program, use this data (omit spaces when writing values on a file):

```
11101011 01000110 11011101 01100110
```

13.7 Binary files

As a binary file for the first attempt, we will use any file in PDF format, because in most cases these are non-text.

Let's read the first five bytes and display their ordinal values:

```lua
local f = io.open("a.pdf", "rb")
local block = f:read(5)
for i=1,#block do
  local onebyte = block:sub(i,i)
  local value = onebyte:byte()
  print (i, string.format("%x  %d  %s",value,value,onebyte))
end
                                         -- or
local f = io.open("a.pdf", "rb")
for i=1,25 do
  local block = f:read(1)
  print (i, string.format("%3d  %02x  %s",block:byte(), block:byte(), block))
end
```

Function `f:read` returns a string. Therefore, we can use all string tools for further processing.

Linux users can check their results using Unix tool:

```
::: od -tu1 -tx1 -c x.pdf
```

```
0000000  37  80  68  70  45  49  46  55  10  37 204 213 193 212 197 216
         25  50  44  46  2d  31  2e  37  0a  25  cc  d5  c1  d4  c5  d8
          %   P   D   F   -   1   .   7  \n   % 314 325 301 324 305 330
```

14 Communication with OS

14.1 Library os

Most programming languages have a library of subroutines that to communicate with the operating system.

Here we briefly introduce the os library and its tools for working with environment variables, date, time and for running external programs.

Program communication with the operating system includes input/output operations, which were discussed earlier (the io library), and processing of command line parameters of the currently running program, which are available directly.

14.2 Reading parameters from the command line

Command line parameters (sometimes called positional parameters) are the information we enter when we run a command to specify to a generally written program, usually the files or directories with which the program is supposed to work with. For example:

```
cp this-file.txt new-file.txt
ls *.txt
```

In Lua scripts we can also get these parameters and use. In general, their use is convenient because they allow us to use one program repeatedly for different values without having to intervene the source code of the script. This is how most of the programs that make up basic Unix/Linux operating system.

Let's imagine a simple program:

```lua
local a, b = arg[1], arg[2]
print (a+b)
```

Let the file be named mycalc. Now let's try to run it and the values we want to add up are given as parameters:

```
lua mycalc 6 8
lua mycalc 100 100
lua mycalc 5 -4
```

14.3 Environmental variables

Environment variables refer to the running command interpreter (shell). They consist of a variable name and associated content. In OS Linux, a list of environment variables can be listed using the `printenv` (abbreviated) command:

```
HOME=/home/tom
LANG=en_US.UTF-8
LOGNAME=tom
PWD=/home/tom/lua/textbook
SHELL=/bin/bash
USER=tom
```

We can see that the variable names and contents are separated by an equals sign.

Typical operations are getting the content of a particular variable and getting a list of variables.

```
print (os.getenv("USER"))
```

```
local envvars = {}

for envline in io.popen("set"):lines() do
  envname = envline:match("^[^=]+")
  envvars[envname] = os.getenv(envname)
end
```

```
paths = (os.getenv("PATH")):gmatch("([%w/%-_]-):")
print(#paths)
```

```
empty buffer
```

14.4 Date and time

There are three functions for date and time management in Lua: `date`, `time`, `clock`, which converts the date in number form to human-readable form, converts human-readable to number form, and returns the number of seconds of CPU time for the program, respectively.

```
print(os.date("today is %A, %B %d"))
Today is Monday, September 15

print(os.time("Now is %X"))
Now is 13:51:53
```

The following table shows possible values for the function date:

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

14.5 Executing programms

Running external programs from a program written in Lua with the `execute` universal function, whose parameter is a string containing the command and all necessary parameters, which we would otherwise write directly to the command line.

```
os.execute("mkdir new_directory")
```

15 Use of Lua in Applications

15.1 Lua in ConTEXt

ConTEXt is an extension of the basic TEX[5] (similarly to LATEX).

It provides a wide range of tools needed to create documents of different types and different complexity.

Its author is Hans Hagen and it was originally called pragmatex. The name ConTEXt has been used since about 1996. Development of ConTEXtruns continuously, based on user requirements.

The examples in this chapter only gently illustrate the wide range of uses of language Lua. Those interested in the ConTEXt typographic system can find out more details on the ConTEXt website (contextgarden, 2022) or in the manuals posted there.

15.1.1 Small multiplication table

The first example creates a small multiplication table.

```
\starttext
\section{This is the small multiplication table}
\startluacode
context.bTABLE()
for i=1,20 do
  context.bTR()
  for j=1,20 do
    context.bTD()
    context(i*j)
    context.eTD()
  end
  context.eTR()
end
context.eTABLE()
\stopluacode
The end.
\stoptext
```

The output format of the ConTEXt system is PDF. The files are created by a compilation process from the source text above. The resulting file can be viewed with any PDF viewer.

The `\starttext` command was used to start processing the document. The com-

[5] The program TEX has been created in 1978 and its author is Donald E. Knuth.

mand `\section` creates a title above the table. The table itself is generated by a Lua code snippet that is bounded by a pair of commands `\startluacode` and `\stopluacode`.

If you are missing the `print` command, do not despair. It is not needed. The generated information will not appear on the standard output as it was in all previous examples, but it is written to the output stream from which the required PDF will be generated. This will resolve the `context` subroutine which extends the Lua language in ConTEXt.

The functions bTABLE, eTABLE, bTR, eTR, bTD, eTD represent the beginnings and ends of tables, rows, and individual cells, respectively. If this way of writing reminds anyone of HTML, it is no coincidence.

15.1.2 Use of data structures

In ConTeXt, all information is stored in large data structures – in tables. For example, language-dependent ones are drawn from these tables information.

The following example first displays the data in the various forms for various languages, to the PDF output stream, and then runs the inspect subroutine, which will display one sub-associative field related to specified month on standard output.

The following example first displays the output stream for PDF data in shapes for different languages and then runs the inspect subroutine which na standard output will display one sub-associative array related to specified month.

```
\starttext
\setupbodyfont[libertinus]

\startlines
\currentdate
\mainlanguage[cs] \currentdate
\mainlanguage[fi] \currentdate
\mainlanguage[sk] \currentdate
\mainlanguage[pl] \currentdate
\mainlanguage[ro] \currentdate
\stoplines

\startluacode
inspect(languages.data.labels.texts.june)
\stopluacode
\stoptext
```

```
table={      -- shortened
 ["labels"]={
  ["af"]="junie",
  ["cs"]="června",
  ["de"]="Juni",
  ["en"]="June",
  ["fi"]="kesäkuu",
  ["fr"]="juin",
  ["hr"]="lipnja",
  ["la"]="Iunius",
  ["lt"]="birželio",
  ["nn"]="juni",
  ["pl"]="czerwca",
  ["pt"]="junho",
  ["ro"]="iunie",
  ["sk"]="júna",
 },
}
```

And this is the output of this example:

December 28, 2022
28. prosince 2022
2022 joulukuu 28
28. decembra 2022
28. grudnia 2022
28 decembrie 2022

References

[Ierusalimschy, Roberto] Lua 5.4 Reference Manual [on-line]. In *Lua.org.* [s. l.] : PUC Rio, 2020–2022 [cit. 2022-11-30]. Dostupné na: https://www.lua.org/manual/5.4/.

Ierusalimschy, Roberto *Programming in Lua.* [s. l.] : Lua.org, 2016. 388 s. ISBN 8590379868.

Ierusalimschy, Roberto; de Figueiredo, Luiz Henrique; Celes, Waldemar *The Evolution of Lua* [on-line]. [2007]. [cit. 2022-11-30]. Dostupné na: www.tecgraf.puc-rio.br/~lhf/ftp/doc/hopl.pdf.

Lua Documentation [on-line]. In *Lua.org.* 2021 [cit. 2022-02-15]. Dostupné na: https://www.lua.org/docs.html.

Lua – Math library [on-line]. In *TutorialsPoint.* [s. d.] [cit. 2022-08-11]. Dostupné na: https://www.tutorialspoint.com/lua/lua_math_library.htm.

luac – Lua compiler [on-line]. In *Lua 5.3 Reference Manual.* 25 Aug 2020 [cit. 2022-02-17]. Dostupné na: https://www.lua.org/manual/5.3/luac.html.

Algorithmization and Programming in Lua
Study text

Author: Tomáš Hála
Mendel University in Brno

Publisher: Mendel University in Brno, Zemědělská 1, 613 00 Brno, Czech Republic
Graphic editing and typesetting: Tomáš Hála
Year of publishing: 2022
First edition
Number of pages: 110

ISBN 978-80-7509-893-1 (online ; pdf)
DOI https://doi.org/10.11118/978-80-7509-893-1