Understanding

A simple introduction to the LUA programming language

Contents :

Chapter 1 : Variables

Chapter 2 : Variables

Chapter 3 : Keywords and comments

Chapter 4 : Blocks

Chapter 5 : Strings

Chapter 6 : Tables

Chapter 7 : Functions

1. First of all

LUA is, like all programming languages, a structured language. Therefore you have to make a certain agreement with yourself about the "name-design" of your code. If you use variables (explained in this first part), you can use lowercase, uppercase or a combined design of uppercase and lowercase characters. It's up to you what you prefer, but please be consistent !

2. Variables

When a computer is running a program, some data has to be remembered during the execution of the program. These data will be stored in the computers memory. The computer is doing this for you in it's own way, but you also have to remember what data you use in a language that's easy to understand. To be able to work with the data, we can give them "names".

2.a. Naming data

You can choose your own name for a variable, but you have to consider some general rules :

- a variable name has to start with a character or an underscore :

  ok : **player**, **_player**
  not ok : ***player**

- using uppercase and lowercase is very important :

  player is not the same as **Player**

- only characters, numbers and underscore are allowed
- reserved words can't be used, don't use names like :

  function, **end**, **for**, …

2.b. Local and global variables

A variable is valid in a certain work area. LUA offers only two kinds of variable scopes :

- **global** variables : can be used in every part in the program and are always remembered
- **local** variables : used in the block where they have been created and active as long as the block is active. One exception : *functions* (see later in this chapter). After leaving the block where a local variable exists, the value in it is lost

Programming in LUA is done inside blocks. A block starts with a *statement* (*do*, *if*, *for*, ...),

followed by a block of code and finally the word "*end*" (again with one exception, see also later). A block can also be a part of another block.

A variable is normally global, unless it has the word **local** in front of it.

2.c. Assigning a value to a variable

To assign (or "give" it) a value to a variable you simply use the "=" character :

```
player1="Charles"
mymoney=100
```

The "=" is called the *operator of allocation* (assignment).

2.d. Variable types

A variable stores a value. This value can be :

- NIL (nothing, empty, not existing)
- numbers
- characters (letters, words, …)
- conditions (true/false, yes/no, 1/0)
- functions (as a special block)
- tables (a collection of variables)

There are two other types existing, but for this moment these are the mainly used types.

2.d.a. NIL

Nothing much to say about NIL, but sometimes it's very useful. If a variable is used that is no existing, a value NIL is generated. This prevents the program from crashing.

You can also delete a variable by assigning NIL to it :

```
player1=NIL
```

2.d.b. Numbers

Numbers are mostly needed for calculation or counting. In LUA this is kept very simple :

```
myage=53
charlesiq=105
fathersmoney=2,000
generalscore=20,000
actualheight=-2
```

Some small rules here :

- use a point as a decimal sign and vice versa
- negative numbers have a minus in front

2.d.c. Characters

Sometimes you have to remember text or single characters, e.g. to write something on the screen. For this purpose you can use a character variable, also known as "**string**".

A string is a concatenation of characters (letters, numbers, special characters). To assign a string to a variable, you have to use quotes :

```
player1="Tom"
mymother='Elly'
mylife=[[gender : male
        place of birth : at home]]
```

You can use the double quote as well as the single quote, but you must use twice the same type of quotes.

We also can include text between [[and]]. With these brackets it's possible to continue the text on a new line.

Strings and numbers can be concatenated very easily. For this purpose two points are used. In LUA there is no need to convert the number into a string :

```
print("player"..2)
```

2.d.d. Conditions

A condition is a type of variable that only knows 2 states. This type is called "boolean" and can be designated in different ways. With LUA we use the words "true" and "false" :

```
player1_Start=true
```

2.d.e. Functions

Functions are variables that are doing something when they are "called". We will describe the use of functions later on.

There are two ways to "call" a function :

```
Calculate=function()
function Calculate()
```

A function is always ended with the reserved word ***end***. Between the beginning and the end of a function there is a block which describes what has to be done.

In the beginning of this chapter was mentioned that there is an exception about local variables. Suppose there is a block that is a global function. After executing this block the (local) variable is still existing inside the function but only inside this function.

Example :

```
do                          -- start of the block
  local cntr=1              -- assignment of variable cntr
  function Incr()           -- creation of a function
    cntr=cntr+1             -- adding 1 to value of cntr
    return cntr             -- returning the vakue of cntr
  end                       -- end of function
end                         -- end of block
print(Incr())               -- show output of function → 2
print(cntr)                 -- show output of variable cntr → nil
```

If we reuse the variable after the function then it seems that the variable became global :

```
do                          -- start of the block
  local cntr=1              -- assignment of variable cntr
  function Incr()           -- creation of a function
    cntr=cntr+1             -- adding 1 to value of cntr
    return cntr             -- returning the vakue of cntr
  end                       -- end of function
  cntr=5                    -- value is changed after the function
end                         -- end of block
print(Incr())               -- show output of function → 6
```

2.d.f. Tables

Tables are variables that store other variables inside them. It makes working with columns easier. They are used very often to store a "family" of data. To assign a table as a variable, use { for the beginning and } for the end :

```
mytable={}
```

Between { and } we can store variables. By using {} there are no values inside, this means the table is empty.

Possible assignments :

```
mytable1={1,2,"Charles",true}
mytable2={X=10,Y=20,Z=50}
```

To access the values of the table, there are a few possibilities :

```
mytable1[1]    → 1
mytable1[2]    → 2
mytable1[3]    → "Charles"
mytable1[4]    → true
```

mytable2.X → 10
mytable2.Y → 20
mytable2.Z → 50
mytable2[X] → 10
mytable2[Y] → 20
mytable2[Z] → 50
mytable2["X"] → 10
mytable2["Y"] → 20
mytable2["Z"] → 50

As shown here, there are two different kinds of tables : one with numbers and the other with names.

Tables can include other tables as variables, so a table is a kind of "more-dimensional variable store".

More about tables will be explained in later examples.

3. Conclusion and advice

It's best to make an arrangement with yourself about the "name-design" of the variable names.

In the next chapters we will use this rules :

- variable names in one word are in lowercase
- functions will have an uppercase first character
- parameters use two words divided by an underscore
- variable names with more than one word are divided with an underscore and every new word starts with an uppercase character

Naturally you can use your own rules...

1. Arythmetic operator types

1.a. Types

LUA uses the most common arythmetic operators :

+	for additing a value	c=a+b
-	for subtracting a value	c=a-b
*	for multiplication of a value	c=a*b
/	for dividing a value	c=a/b
^	for exponentiation of a value	c=a^b
-	(again) for negation of values	c=-a

Examples :

	value a	**value b**	**result c**	
c=a+b	1	2	3	1+2=3
c=a-b	4	3	1	4-3=1
c=a*b	2	4	8	2*4=8
c=a/b	6	2	3	6/2=3
c=a^b	2	3	8	2^3=8 (2*2*2)
c=-a	2		-2	-+ gives -
	-3		3	-- gives +

1.b. Variables and arythmetic operators

In the first chapter variables are explained. The purpose of arythmetic operators is mainly to make calculations. A value of a variable can change depending on an arythmetic operator :

```
myvar=1+2
myvar=yourvar*2
```

Naturally you have to use numbers, LUA allows also numbers with quotes, but numbers only :

ok : **myvar=1+2**
ok : **myvar="1"+"2"**
not ok : **myvar="1"+"2y"**

2. Relational operator types

2.a. Types

LUA also has a second operator class, called relational operators. They are used to make comparisons and are used very often.

The result of a comparison is **either true or false** :

==	is left side equal to right side ?	a==b
<	is left side smaller then right side ?	a<b
>	is left side bigger then right side ?	a>b
<=	is left side smaller then or equal to right side ?	a<=b
>=	is left side bigger then or equal to right side ?	a>=b
~=	is left side not equal to right side ?	a~=b

Examples :

	value a	**value b**	**result c**	
==	1	2	false	1==2
<	2	3	true	2<3
>	8	4	true	8>4
<=	6	2	false	6<=2
>=	5	3	true	2>=3
~=	2	5	true	2~=5

Some remarks :

- in the most programming languages a "difference check" is mostly done by **<>**, in LUA it's done by **~=**.
- watch out when using **=** for comparison, you have to type it **twice**

For the comparison it's also possible to use text :

```
"Tom"=="Tom"        true
"Tom">"Tommy"       false
```

2.b. Variables and relational operators

A value of a variable can also be compared with another value or another variable, and can even be combined with an arythmetic operator :

```
myvar<1
myvar>=yourvar*2
myvar=="Tom"
```

3. Logical operator types

3.a. Types

A third kind of operators is called logical operators. They extend the possibilities by giving you the opportunity to combine other operator parts.

The result of a combination is **also either true or false**.

LUA knows three type of logical operators :

and	a==b and c==d
or	a<b or c>d
not	not a>b

Examples :

	part a	part b	result	
and	1==2	2==3	false	1==2 and 2==3
or	2<3	4>3	true	2<3 or 4>3
not	2==3		true	not 2==3

Logical operators can be combined :

```
if 2<3 and 4>3 or 10==9+1 then dosomething end
if 2<3 and 4>3 or not 10==9+1 then dosomethingelse end
```

(the reserved words ***if***, ***then*** and ***end*** will be explained later)

3.b. Variables and logical operators

Again, a comparison of a value of a variable can also be combined with another comparison of a value or another variable, and again can even be combined with an arythmetic operator :

```
myvar<1 and yourvar==2
myvar>=yourvar*2 and ourvar>5
myvar=="Tom" and not yourvar=="Jerry"
```

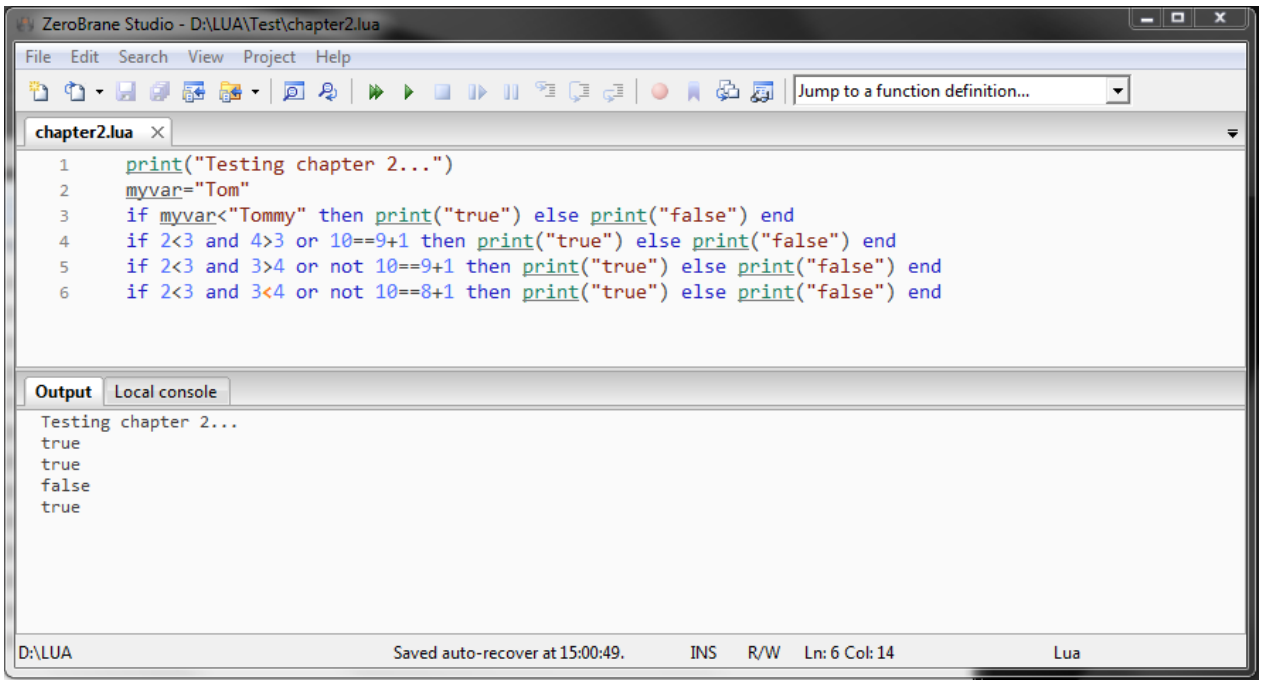
4. Conclusion and advice

Operators are a powerful ingredient of your program, because they can force blocks of code to be executed depending of their result. They let the program make decisions.

Some advice : be carefull using *and*, *or* and *not* ! If you combine them wrong, you can get strange results. Bu sure what to use first… and test your code !

5. The real stuff

To show the use of the operators, here's a small example of code :



Line #		result
2	give myvar the value "Tom"	
3	test if "Tommy" is greater then "Tom"	true
4	test if 2 is smaller then 3 and if 4 is bigger then 3, or if 10 equals 9+1	true
5	test if 2 is smaller then 3 and if 3 is bigger then 4 or 10 differs from 9+1	false
6	same as 5 to test the condition "not"	true
	test if 2 is smaller then 3 and if 3 is smaller then 4 or 10 differs from 8+1	

1. Keywords

As allready mentioned in previous chapters, LUA uses variables to work with. You can almost freely choose the name for a variable. Yes, *almost*, because there are a number of "reserved" names that have another meaning in LUA. They are called **keywords**.

The following keywords are reserved and cannot be used as variable names :

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

At this moment there's no more to say about these keywords, in later chapters we'll use them in some code.

2. Comments

Every programming language uses comments. They are used to give a better overview and to describe what's going on. This comment "information" is mainly for yourself, but if you give your code to others, they'll appreciate it if they are well informed.

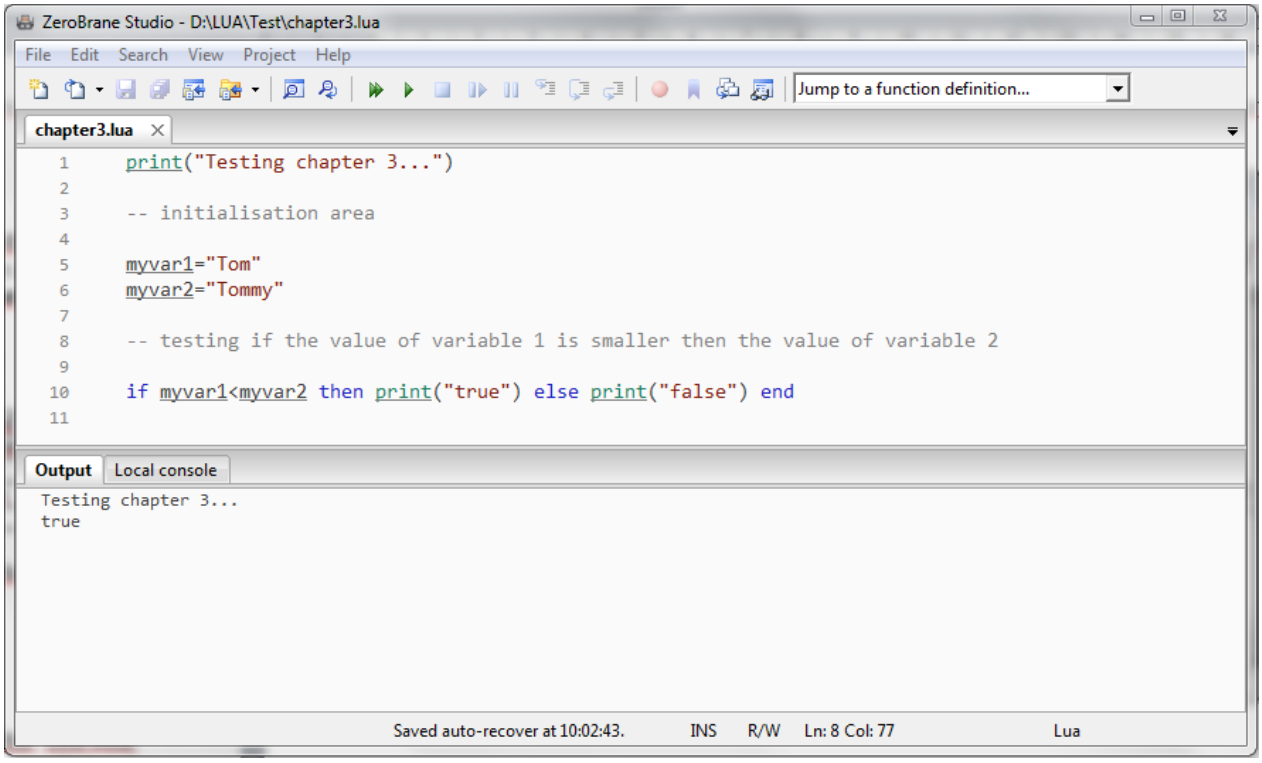
Commenting in LUA is very simple : if you use a double minus sign (--), LUA knows that all the text on that line is comment and is not necessary for the execution of the code.

For more than one line of comment you can use --[[for the beginning and]] for finishing the comment.

Example :

In this example 2 variables get a textvalue in lines 5 and 6, and are compared in line 10.

Notice the commentlines 3 and 8.


ZeroBrane Studio - D:\LUA\Test\chapter3.lua
File Edit Search View Project Help
Jump to a function definition...
chapter3.lua
1 print("Testing chapter 3...")
2
3 -- initialisation area
4
5 myvar1="Tom"
6 myvar2="Tommy"
7
8 -- testing if the value of variable 1 is smaller then the value of variable 2
9
10 if myvar1<myvar2 then print("true") else print("false") end
11
Output Local console
Testing chapter 3...
true
Saved auto-recover at 10:02:43. INS R/W Ln: 8 Col: 77 Lua

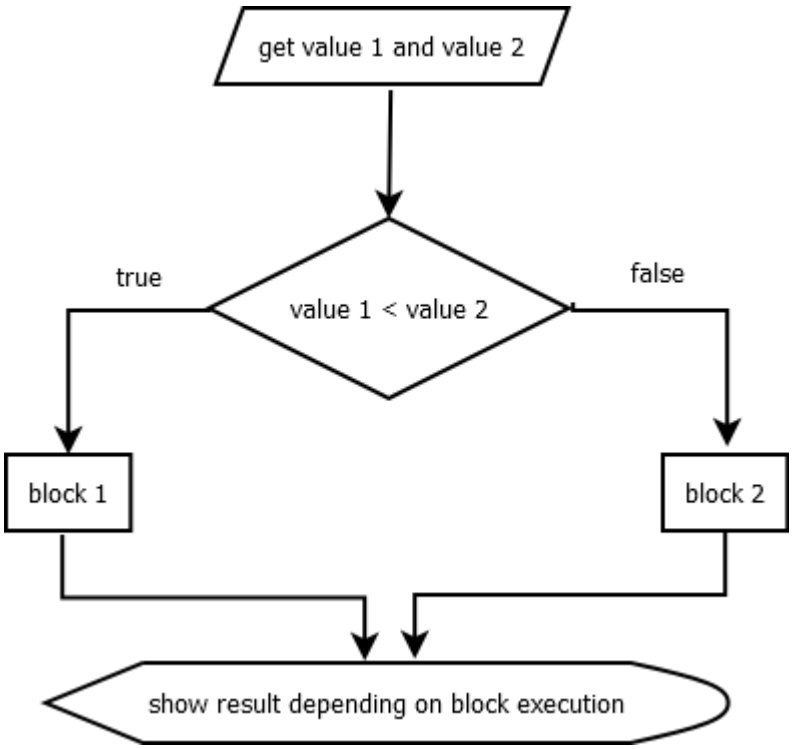
1. Blocks

A program is normally made in a structural way. That means that parts of the code are combined in small pieces of code that are doing something. It will allow you to have a better readability of the code, and mostly blocks are used to be reused several times. Morely less code in a program, but more than once executable at any time…

A block can contain another block and so on…

There are mainly three types of blocks : conditional structures, functions and loops. Blocks always closed with the reserved word ***end*** (with one exception).

In this image a possible program structure is shown :



1.a. Conditional structures

In this type of block code is executed depending on a condition. Sometimes "questions" have to be asked, and depending on the "answer" certain code is executed. The question is "What if…", or translated to LUA made by the keywords ***if***, ***else*** and ***elseif***.

There are 3 possible conditional structures (keywords are shown in **blue**) :

One option :

```
if condition then
    block
end
```

Two options :

```
if condition then
    block_1
else
    block_2
end
```

More options (as many as you want) **:**

```
if condition_1 then
    block_1
elseif condition_2 then
    block_2
elseif condition_3 then
    block_3
else
    block_4
end
```

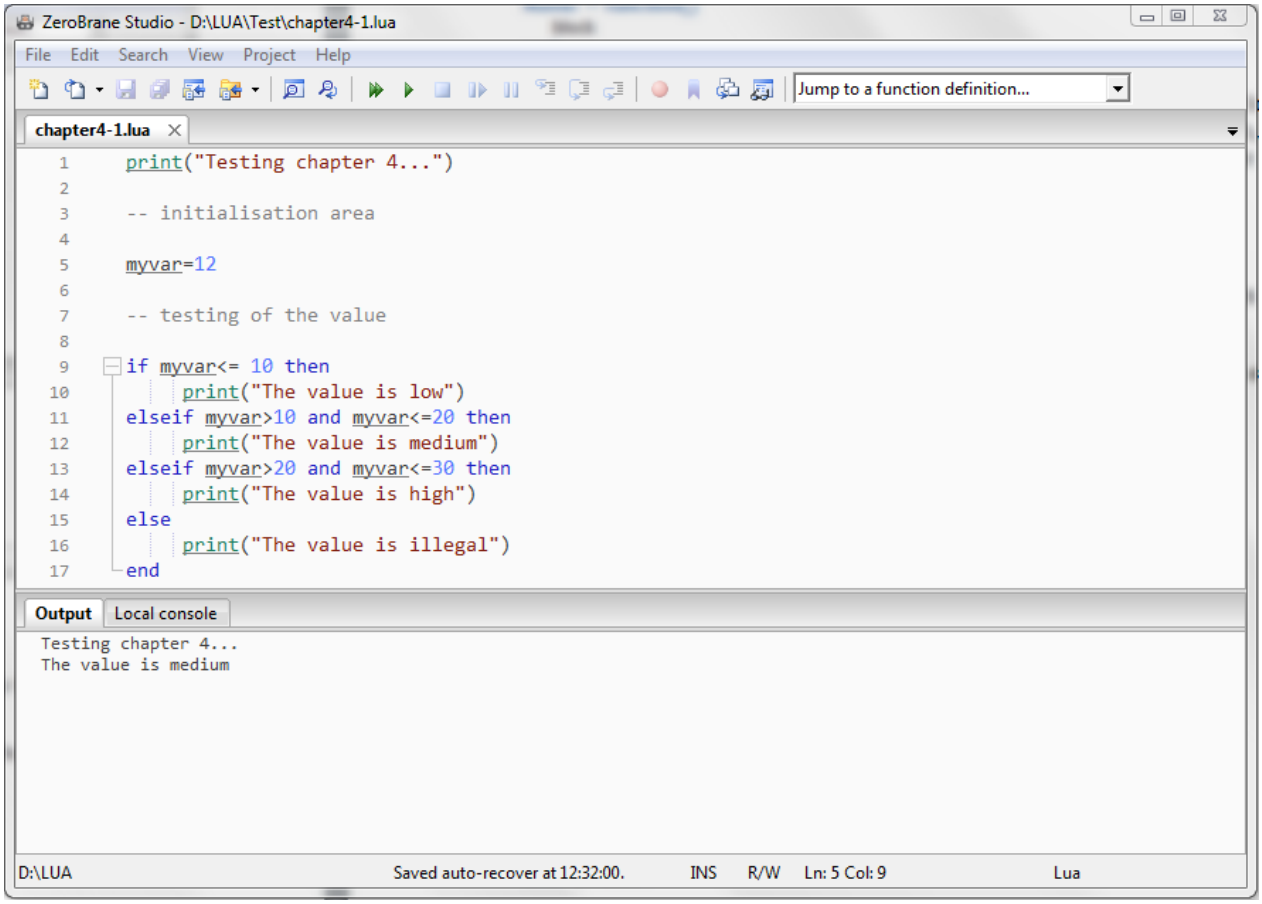
Notice that ***else*** has to be interpreted as "*otherwise*" or "*none of the above*".

Example :

Suppose we want to print a text depending on a certain value :

value	**text to be printed**
between 1 and 10	The value is low
between 11 and 20	The value is medium
between 21 and 30	The value is high
else	The value is illegal

We have 4 possibilities here. Let's show the code :



Line #

5	give myvar the value 12
9	test if myvar is smaller or equal then 10
10	if so, print "The value is low"
11	test if myvar is bigger then 10 and smaller or equal then 20
12	if so, print "The value is medium"
13	test if myvar is bigger then 20 and smaller or equal then 30
14	if so, print "The value is high"
15+16	in all other cases, print "The value is illegal"

Remarks :

- this is not complete ok. Why not ? If we use a value of zero or a negative value, the value will be interpreted as "low". It all depends on what's the goal of your program.
- conditions can be defined in several ways, e.g. *if myvar <= 10* can also be written as *if myvar < 11*. If you do so, at least the second condition in the above code must also be changed.

1.b. Functions

A function is a block of code that has one great feature : it can be reused several times, and can optional return (a) value(s) depending on the value(s) it gets when "called".

A function starts with the keyword ***function*** and ends with the keyword ***end***. It can be defined in two ways :

Method one :

```
function Name()
    block
end
```

Medthod two :

```
Name = function()
    block
end
```

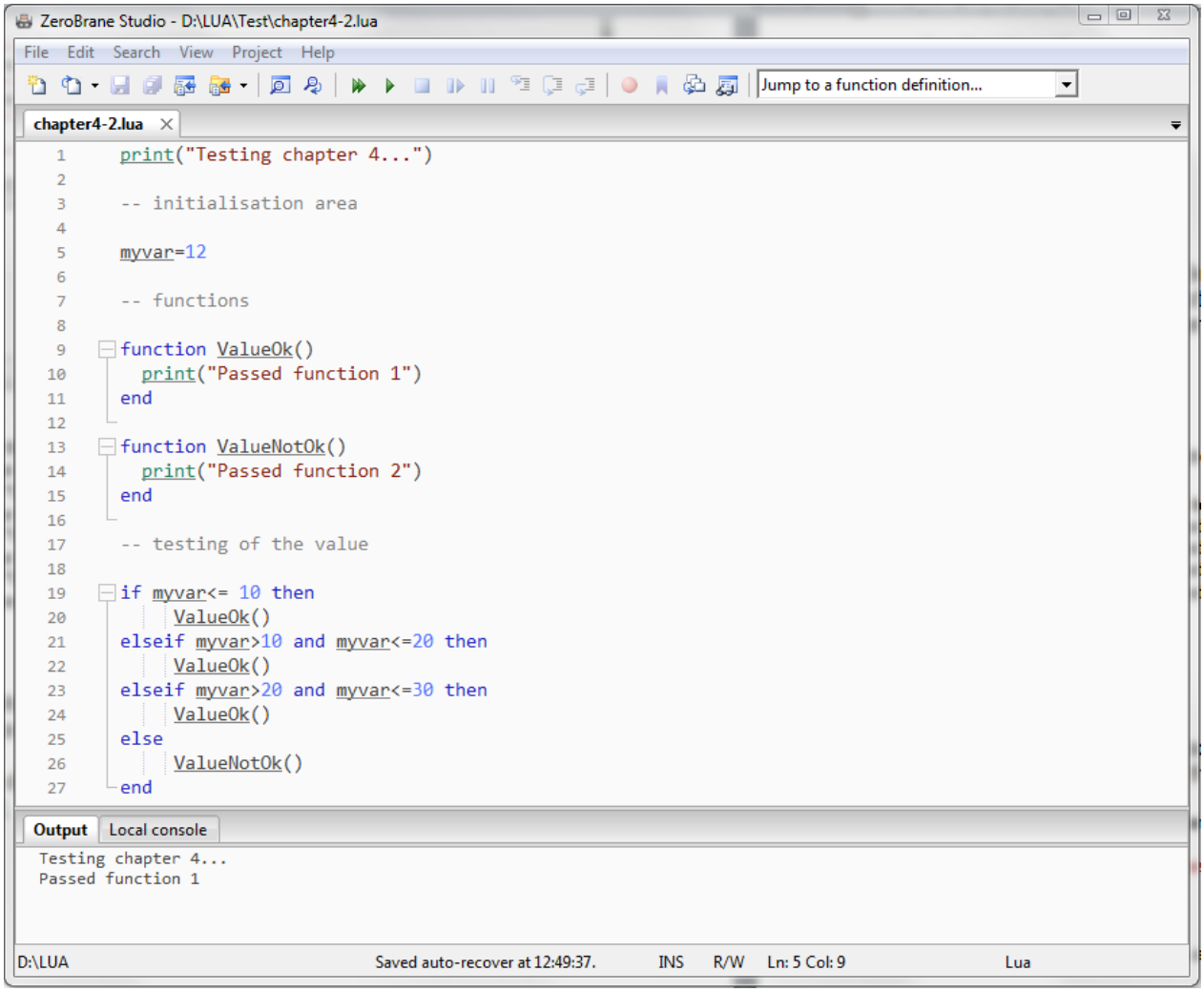
Notice that there are always two brackets () at the end of the function definition ! If you don't send (a) value(s) to the function, there is nothing between the brackets, otherwise mostly one or more variable names are placed between the brackets (see in a later chapter).

Example :

Depending on a value a certain function is called, in the function is a print command :

value	**text to be printed**
between 1 and 10	Passed function 1
between 11 and 20	Passed function 1
between 21 and 30	Passed function 1
else	Passed function 2

By using a function, we reduce the code. The same block can be executed in the first 3 possible situations :



This is kind of a stupid example, but otherwise, suppose if you want to change the text to print, you have only to change it in the function block once, and it is active for the 3 possible situations that are calling the function.

1.c. Loops

Finally there is a third kind of block structures, called loops. A loop executes a block of code several times, depending on the type of loop you want :

Execute a block depending from one value to another and by using a certain step :

```
for variable = value_start, value_end, step do
    block
end
```

The same, but in a table :

```
for variable, name_in_table do
    block
end
```

Execute a block while a condition is true :

```
while condition do
    block
end
```

Execute a block until a condition is true :

```
repeat
    block
until condition
```

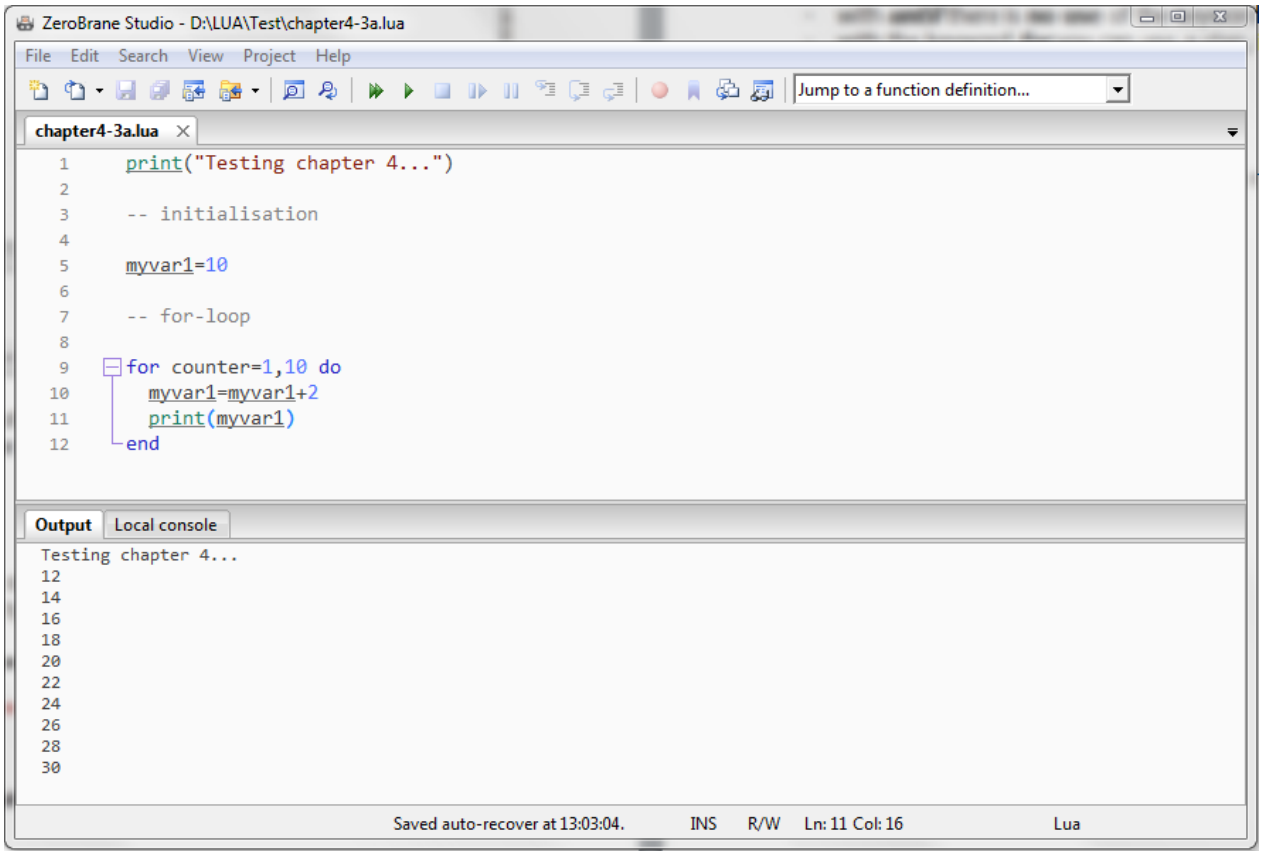
Notice :

- with ***until*** there is **no use** of the keyword ***end***
- with the keyword ***for*** you can use a step, if you don't use it LUA assumes the stepvalue is 1

Example :

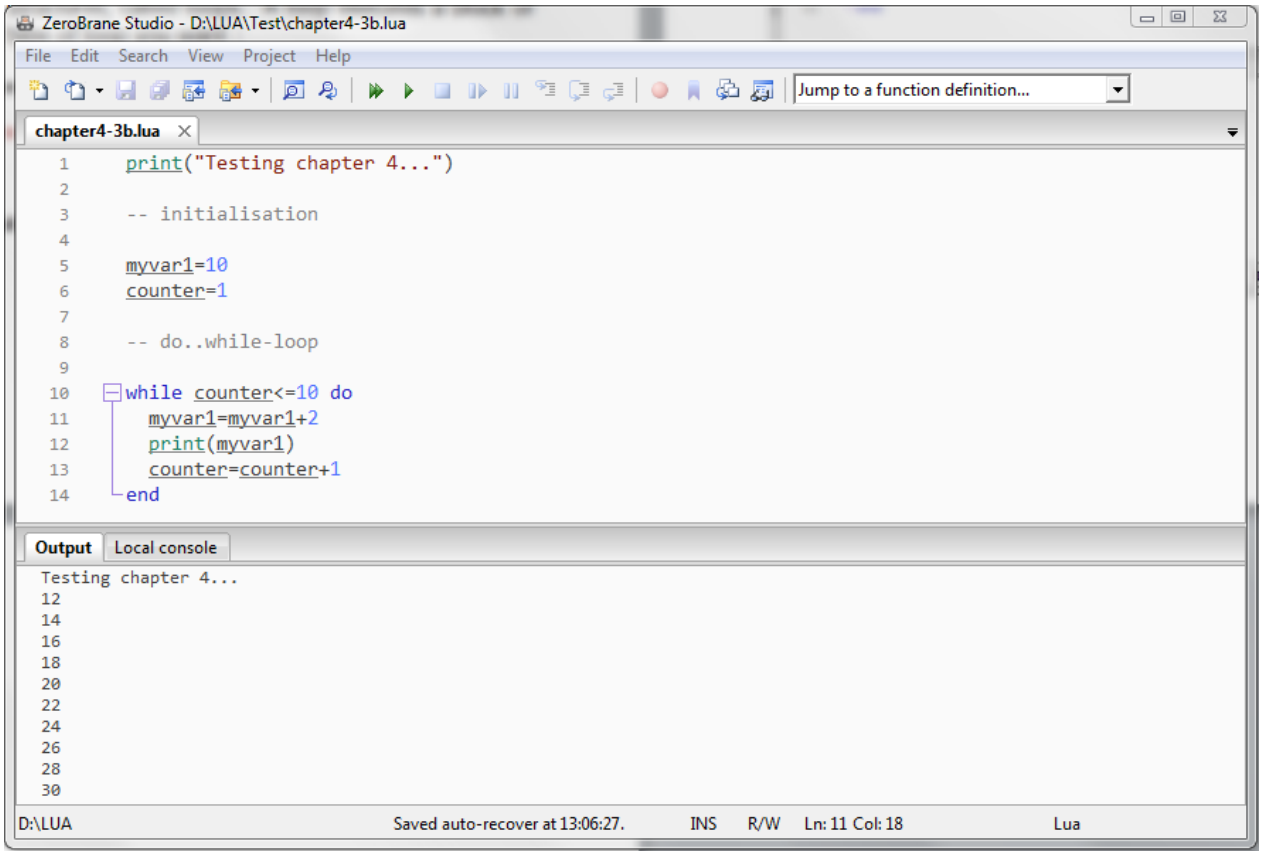
Start with a value of 10, add 10 times 2, so the last result has to be 30.

Using for :



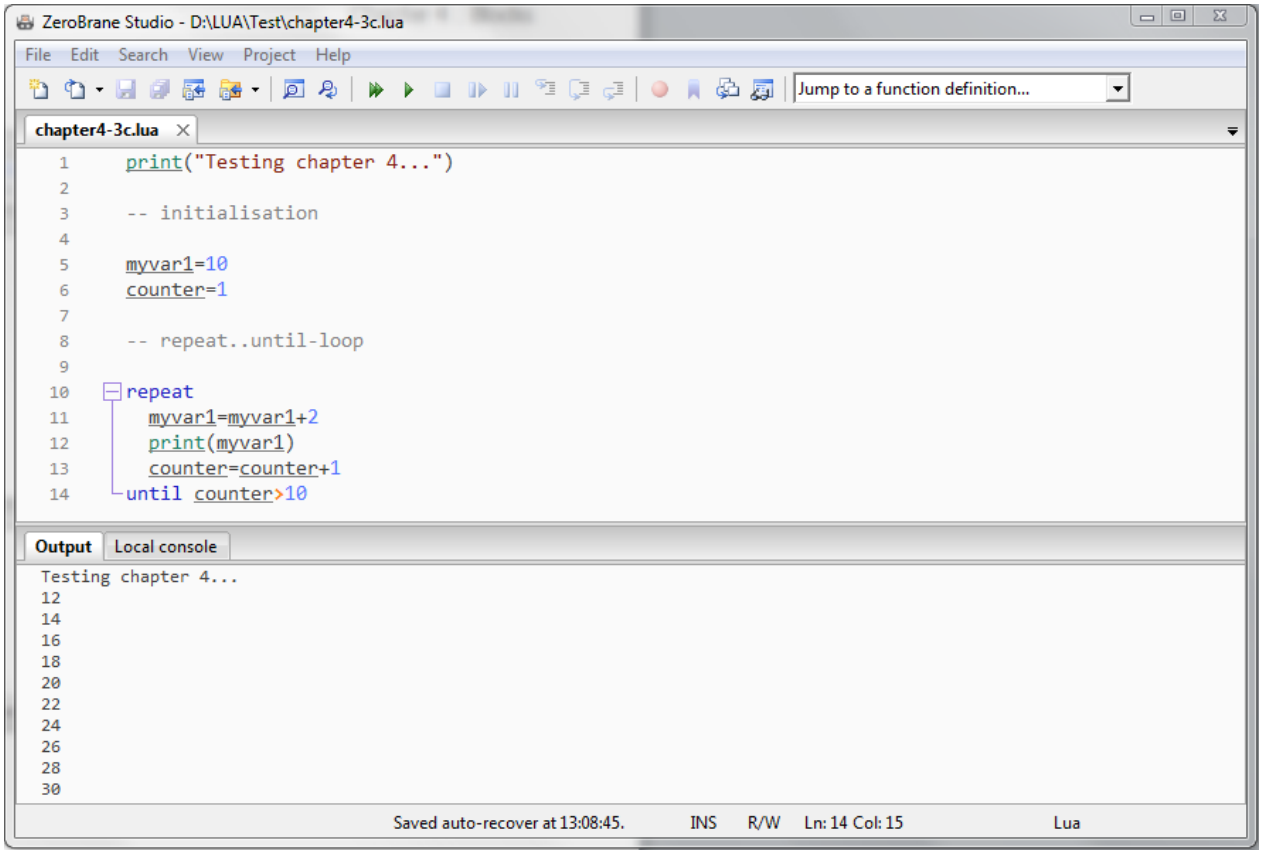
Using do..while :

- using a counter that ends the loop if the value is smaller then or equal to 10



Using repeat..until :

- using a counter that ends the loop if it gets a value greater then 10



As you can notice, the result is 3 times the same, but the code differs in many ways.

2. Leaving blocks

Normally you leave a block when a condition is fullfilled, that's called "elegant and logical programming". In the last LUA-versions the ***goto***-statement is added, this statement will allow you to leave a loop immediately if needed.

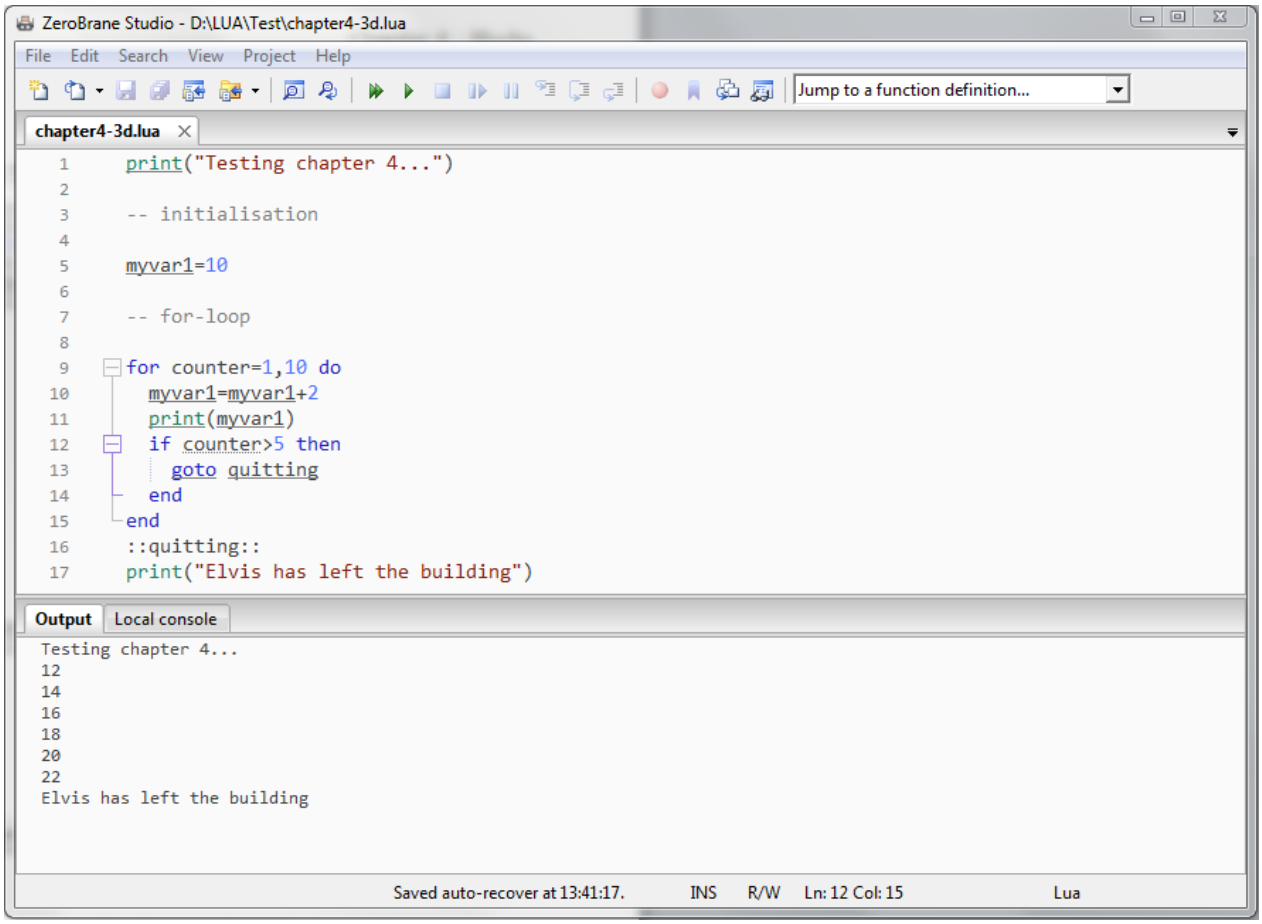
A goto-statement has to point to a *label*, otherwise your program doesn't know where to go to…

Some rules to follow :

- a label is visible in the entire block where it is defined (including nested blocks, but not nested functions).
- a goto may jump to any visible label as long as it does not enter into the scope of a local variable.
- a label starts en ends with ::

Example :

Exit a loop when that loop is executed 5 times :



Again, a stupid example, but this gives you an idea how ***goto*** works.

1. What is a string ?

A string is a concatenation of characters and numbers, mostly defined as text. To use a string in a variable, the text has to be placed between single quotes, double quotes, or double square brackets :

```
player1=”Charles”
player1='Charles'
player1=[[Charles]]
```

The purpose of those different types is to allow you to use them as part of the string :

```
answer='My name is "nobody"'
```

Remarks :

- you can't make calculations with strings that contain other characters then numbers.
- always use the same symbol to end or start your string with, if you use double quotes then end with double quotes
- if a string contains numbers only, it can be used as a number (see chapter 1)

2. The string library

LUA has a "library" of prebuild functions to perform certain actions on strings. Not all the functions will be explained here, we have chosen for the most used.

2.a. String.find(s, pattern [,index [,plain]])

Find the first occurrence of a pattern in the string. If an instance of the pattern is found a value representing the start of the string is returned. If the pattern cannot be found nil is returned :

```
string.find("Hello LUA user", "LUA")
7
string.find("Hello LUA user", "abcde")
nil
```

We can optionally specify where to start the search with a third argument. The argument may also be negative which means we count back from the end of the string and start the search.

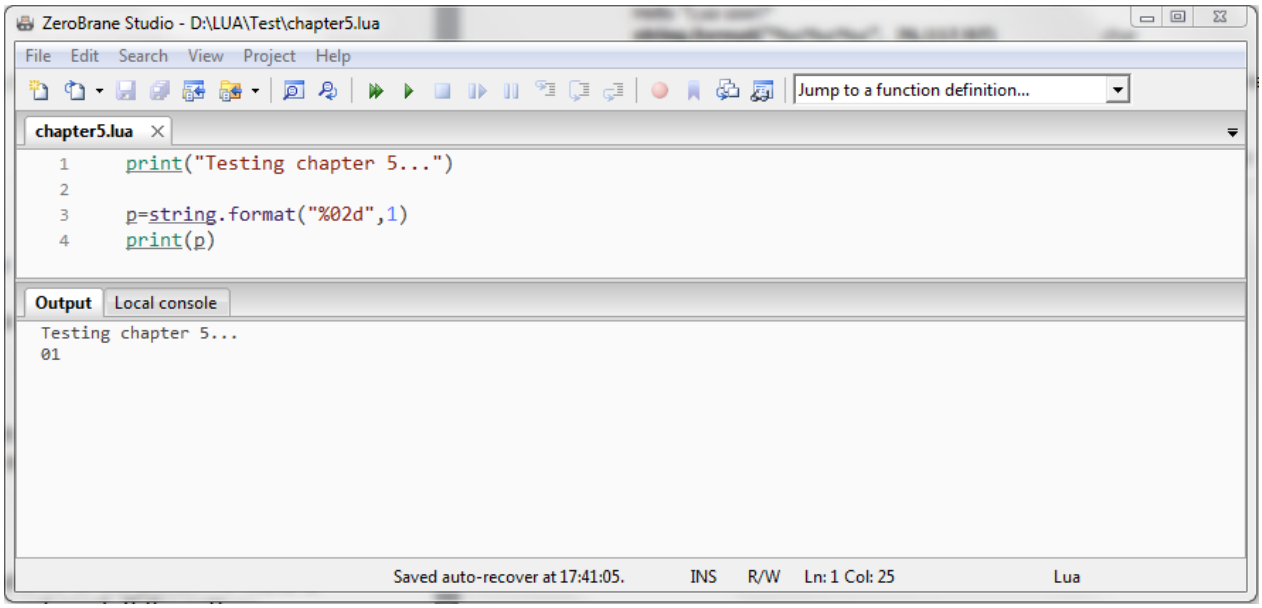
```
string.find("Hello LUA user", "LUA", 1)      start at first character
7 9
string.find("Hello LUA user", "LUA", 8)      "LUA" not found again after character 8
nil
string.find("Hello LUA user", "e", -5)       first "e" 5 characters from the end
13
```

2.b. String.format(s, e1, e2, ...)

Create a formatted string from the format and arguments provided. An additional option %q puts quotes around a string. Arguments start with % and can be c, d, E, e, f, g, G, i, o, u, X, and x all expect a number, arguments q and s expect a string.

. **string.format("%s %q", "Hello", "LUA user")** string and quoted string
Hello "Lua user"
string.format("%c%c%c", 76,85,65) returns characters from ansi-codes
LUA
string.format("%d, %i, %u", -100,-100,-100) signed, signed, unsigned integer
-100, -100, 4294967196

It's possible to add a numeric value to some arguments. Suppose you want to format a number (for example 1) by placing leading zero's to it and make it 2 positions wide :



2.c. String.gsub(s,pattern, replace [,n])

This is a very powerful function and can be used in multiple ways. Used simply it can replace all instances of the pattern provided with the replacement. A pair of values is returned, the modified string and the number of substitutions made. The optional fourth argument n can be used to limit the number of substitutions made :

string.gsub("Hello banana", "banana", "LUA user")
Hello LUA user
string.gsub("banana", "a", "A", 2) limit substitutions made to 2
bAnAna

2.d. String.len(s)

Returns the length of the string passed.

```
string.len("LUA")
3
string.len("")
0
```

2.e. String.lower(s)

Make uppercase characters lowercase.

string.lower("Hello, LUA user")
hello, lua user

2.f. String.rep(s, n)

Generate a string n copies of the string passed and concatenate them.

string.rep("LUA ",5)
LUA LUA LUA LUA LUA

2.g. String.reverse(s)

Reverses a string.

string.reverse("LUA")
AUL

2.h. String.sub(s, i [,j])

Return a substring of the string passed. The substring starts at i. If the third argument j is not given, the substring will end at the end of the string. If the third argument is given, the substring ends at and includes j.

string.sub("Hello LUA user", 7) from character 7 until the end
LUA user
string.sub("Hello LUA user", 7, 9) from character 7 until and including 9
LUA
string.sub("Hello LUA user", -8) 8 from the end until the end
LUA user

2.i. String.upper(s)

Make lowercase characters uppercase.

string.upper("Hello, LUA user")
HELLO, LUA USER

3. Special pattern characters

This is a list of special pattern characters that can be used in LUA :

-.: (a dot) represents all characters.

- %a: represents all letters.

- %c: represents all control characters.

- %d: represents all digits.

- %l: represents all lowercase letters.

- %p: represents all punctuation characters.

- %s: represents all space characters.

- %u: represents all uppercase letters.

- %w: represents all alphanumeric characters.

- %x: represents all hexadecimal digits.

- %z: represents the character with representation 0.

- %x: (where x is any non-alphanumeric character) represents the character x

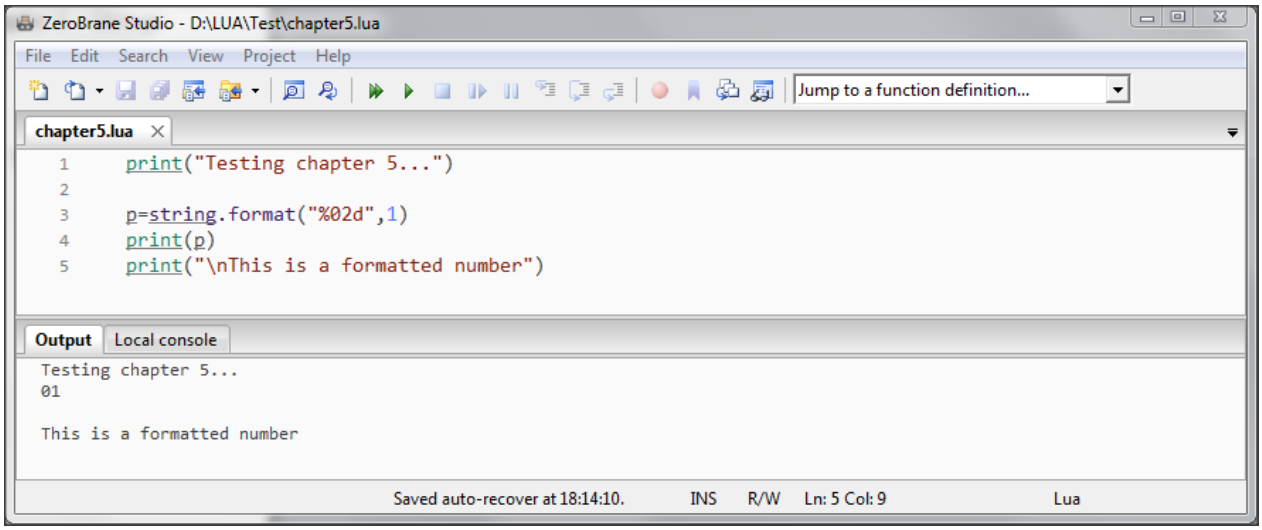
4. Special string characters

LUA includes also a set of special characters to result in certain "actions". They are called ***escape sequences***. These are the most used :

- \a (bell),
- \b (backspace),
- \f (form feed)
- \n (newline)
- \r (carriage return)
- \t (horizontal tab)
- \v (vertical tab)
- \\ (backslash)
- \" (double quote
- \' (single quote)

Example :

Suppose we print a value first and then a text with a blank line between them :



Line #

3 format the number 1 to a leading-zero format of two digits
4 print the formatted number
5 print the text, \n in front sends a carriage return before printing the text

1. What is a table ?

Tables are the most comfortable part of LUA. It's a little difficult to understand, but if you get the table-point (!?), you'll use it very much. In fact, LUA itself "exists" in a table.

Everything what is known in LUA is located in the table _G. If something is not included in _G it does not exist or it is somewhere local.

Several standard functions are available in some LUA-tables. This helps us, because we do not have to develop a lot of functions as they are already existing. And these functions are much faster as well, because they have mostly been written in the programming language "C", which is very close to the operating system. This language is not easy to program, also not easy to learn but it is about 3 times faster than LUA.

What we have know about tables :

- it is a variable
- as every variable it has a name
- a table definition starts with { and finishes with }
- a table can include other variables (all types of variables and also other tables)
- there are 2 different kind of tables :
  - it can be of a numerical type : myTable[4]
  - or it can have a name : myTable.myVariable

2. Creating a table

First it's important to tell LUA that we are creating a table. That is done this way :

```
myTable = {}
```

Now LUA knows that everything that follows and has this name belongs to this table.

We'll explain the use of tables with some examples.

2.a. Table with names

2.a.a. Table with names with standalone values

Suppose we have these variables defined :

```
player1Name ="Tom"
player1Gold=1000
player1IsAlive= true
```

```
player2Name="Jerry"
player2Gold=1500
player2IsAlive=true
```

Let's define a table for each "player" :

```
player1 = {}
player2 = {}
```

We "transfer" the variable naming to the 2 defined tables. This can be done in 2 "versions" :

Version 1

```
(for Player 1)   player1.Name="Tom"
                 player1.Gold=1000
                 player1.IsAlive=true

(for Player 2)   player2.Name="Jerry"
                 player2.Gold=1500
                 player2.IsAlive=true
```

Version 2

```
(for Player 1)   player1 = {Name="Tom",Gold=1000,IsAlive=true}
(for Player 2)   player2 = {Name="Jerry",Gold=1500,IsAlive=true}
```

Both versions are equivalent. Don't forget to seperate the values with a comma !

2.a.b. Table with names with another table with names included

Sorry for the weird title…

A table can include another table. So a value is extended to another "group" of values stored in another table that's part of the "mother" table. The "child" table can again include other tables. At this point the "child" table becomes the "daughter" and so on…

Naturally it's not preferable to make a lot of "generations" of tables, but it's possible.

Suppose we add a new tables in our previous example : a table with 2 values X and Y :

```
player1.Position={X=120,Y=360}
player2.Position={X=180,Y=270}
```

Our versions will be :

Version 1

```
(for Player 1)   player1.Name="Tom"
                 player1.Gold=1000
                 player1.IsAlive=true
                 player1.Position={X=120,Y=360}

(for Player 2)   player2.Name="Jerry"
                 player2.Gold=1500
                 player2.IsAlive=true
                 player2.Position={X=180,Y=270}
```

Version 2

```
(for Player 1)   player1={Name="Tom",Gold=1000,IsAlive=true,Position={X=120,Y=360}}
(for Player 2)   player2={Name="Jerry",Gold=1500,IsAlive=true,Position={X=180,Y=270}}
```

Again, both versions are equivalent.

2.a.c. Accessing values in a table with names

To access a value in a table with names, simply separate the "family" part wanted by a point. Every "generation" is separated from the previous one with a point.

Suppose we have a family where "Sue" is the mother, "Ellen" is the daughter and "Tom" and "Jerry" are children of "Ellen" :

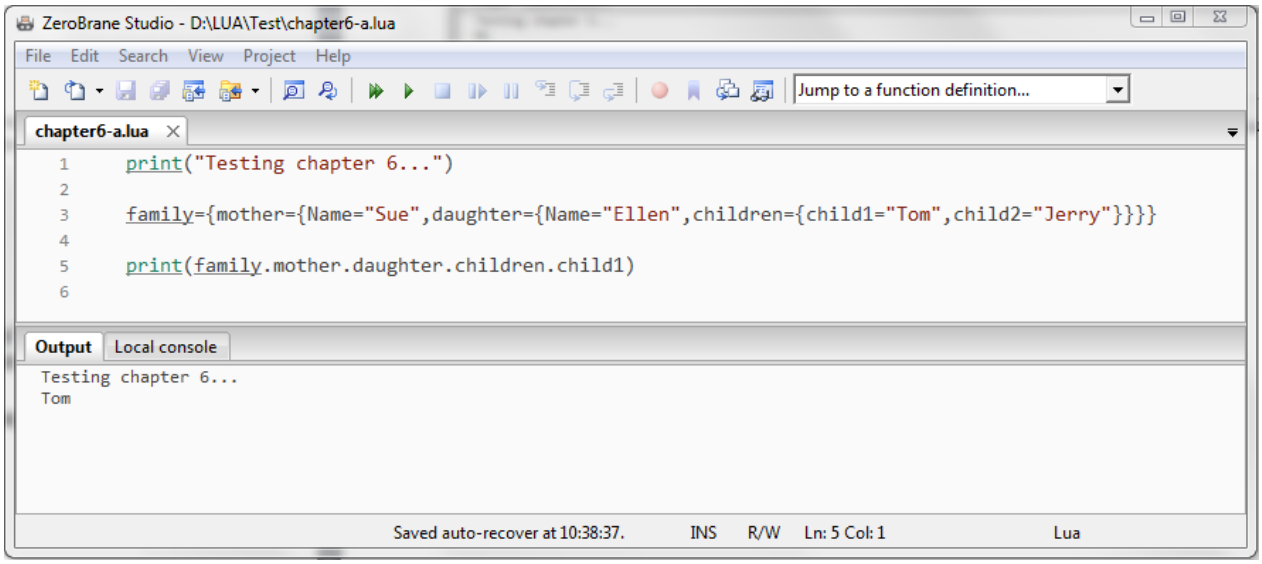
```
mother = "Sue"
daughter = "Ellen"
child1 = "Tom"
child2 = "Jerry"
```

The table definition will be :

```
family={mother={Name="Sue",daughter={Name="Ellen",children={child1="Tom",child2="Jerry"}}}}
```

To access the value of this family-table, we have to use the names to get some result.

Suppose we want to know the name of the first child, this will be our code :



Line #

3	table definition
5	print the wanted content

This looks quite complicated, but logical… we have 3 tables inside the family-table :

- one for the children with 2 items (the two children)
- one for the daughter with 2 items (daughter name and children table)
- one for the mother with 3 items (mother name, daughter name and children)

Notice that at the end we have 4 times } !

2.b. Table with numbers

This kind of table assigns numbers for the children and therefor a combination with loops can be done easier.

First of all, the table has to be defined in general :

```
player={}
```

Again, our example with the 2 players :

Version 1

```
(for Player 1)   player1.[1]="Tom"
                 player1.[2]=1000
                 player1.[3]=true

(for Player 2)   player2.[1]="Jerry"
                 player2.[2]=1500
                 player2.[3]=true
```

Version 2

```
(for Player 1)   player1 = {[1]="Tom",[2]=1000,[3]=true}
(for Player 2)   player2 = {[1]="Jerry",[2]=1500,[3]=true}
```

This way is OK, but not quite clear to read. We would have to keep in mind that [1] is Name, [2] is Gold and [3] is IsAlive.

If we would be able to mix it with the theory about tables with names, it would be better, and fortunately, this is possible.

2.c. Table with numbers and names combined

Again, the table has to be defined in general :

```
player={}
```

Mixing both type now gives this as result :

Version 1

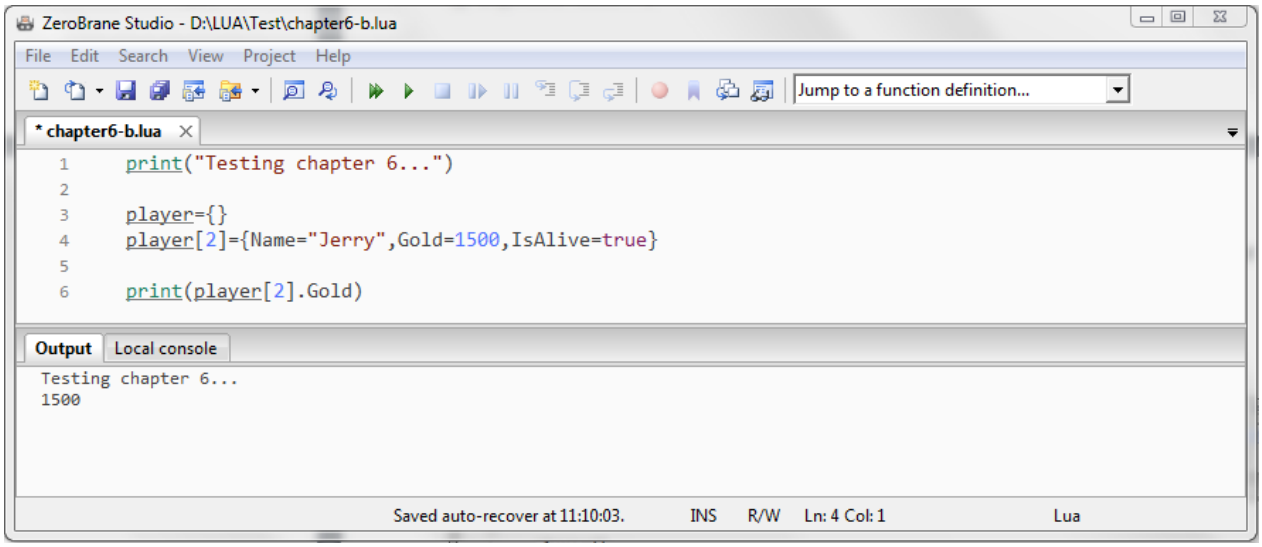
```
(for Player 1)   player[1].Name="Tom"
                 player[1].Gold=1000
                 player[1].IsAlive=true

(for Player 2)   player[2].Name="Jerry"
                 player[2].Gold=1500
                 player[2].IsAlive=true
```

Version 2

```
(for Player 1)   player[1] = {Name="Tom",Gold=1000,IsAlive=true}
(for Player 2)   player[2] = {Name="Jerry",Gold=1500,IsAlive=true}
```

Notice that the “indexes” (between [and]) differ from the previous example where only numbers are used.

If we want to know the value "Gold" of player 2, this is the code :



Line #

3	table definition
4	table content definition
6	print the wanted content

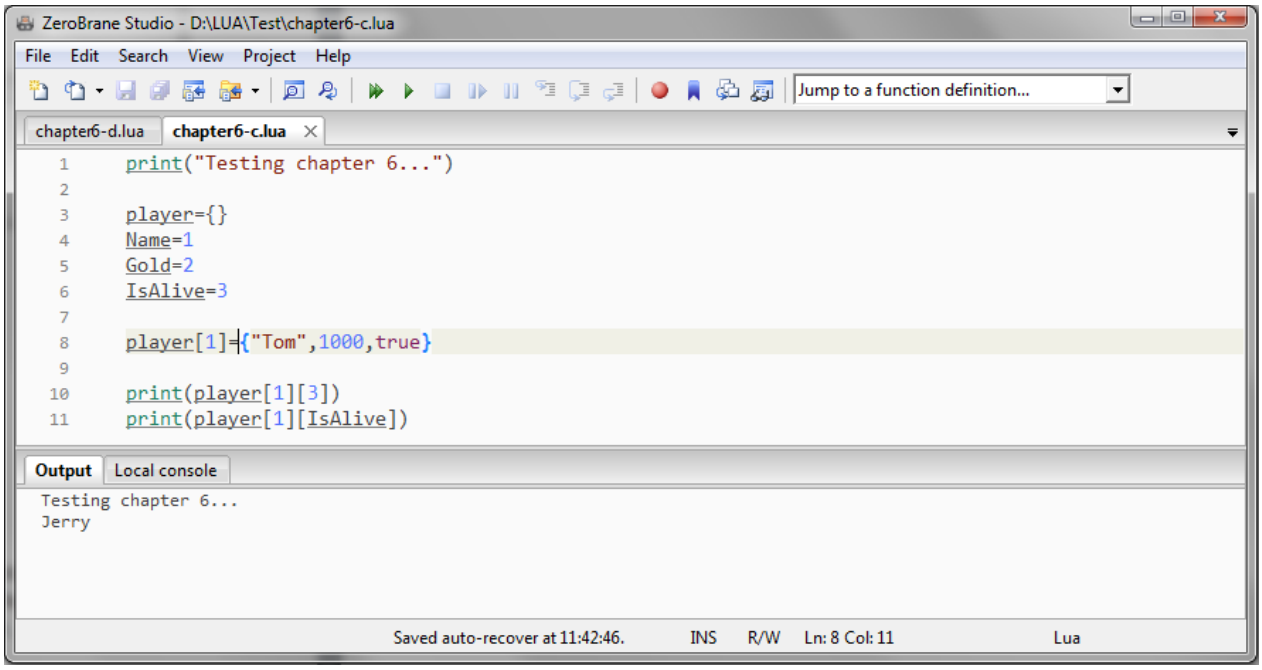
2.d. Table with numbers and names combined (pointer version)

This is the same as the above item, but you can define the content names of the tables separately by using "***pointers***".

This will be :

```
player={}
Name=1
Gold=2
IsAlive=3

(for Player 1)    player[1] = {"Tom",1000,true}
(for Player 2)    player[2] = {"Jerry",1500,true}
```

If we want to know the value "IsAlive" of player 1, this is the code :



Line #	
3	table definition
4-6	value name definition (pointers)
8	table content definition (only real value, no value names)
10	print the wanted content by value numbered
11	print the wanted content by value named

(lines 10 and 11 are giving the same result here)

3. Loops and tables

In this part we'll explain how a table can be accessed by using a loop.

Suppose we want to show the family tree. First we create our tables :

```
family={}
child={}
```

Now we define some pointers to make our code more readable (the numbers here are positions in the table) :

for the family table	mothername=1
	amount_of_children=2
for the child table	childname=1
	amount_of_grandchildren=2

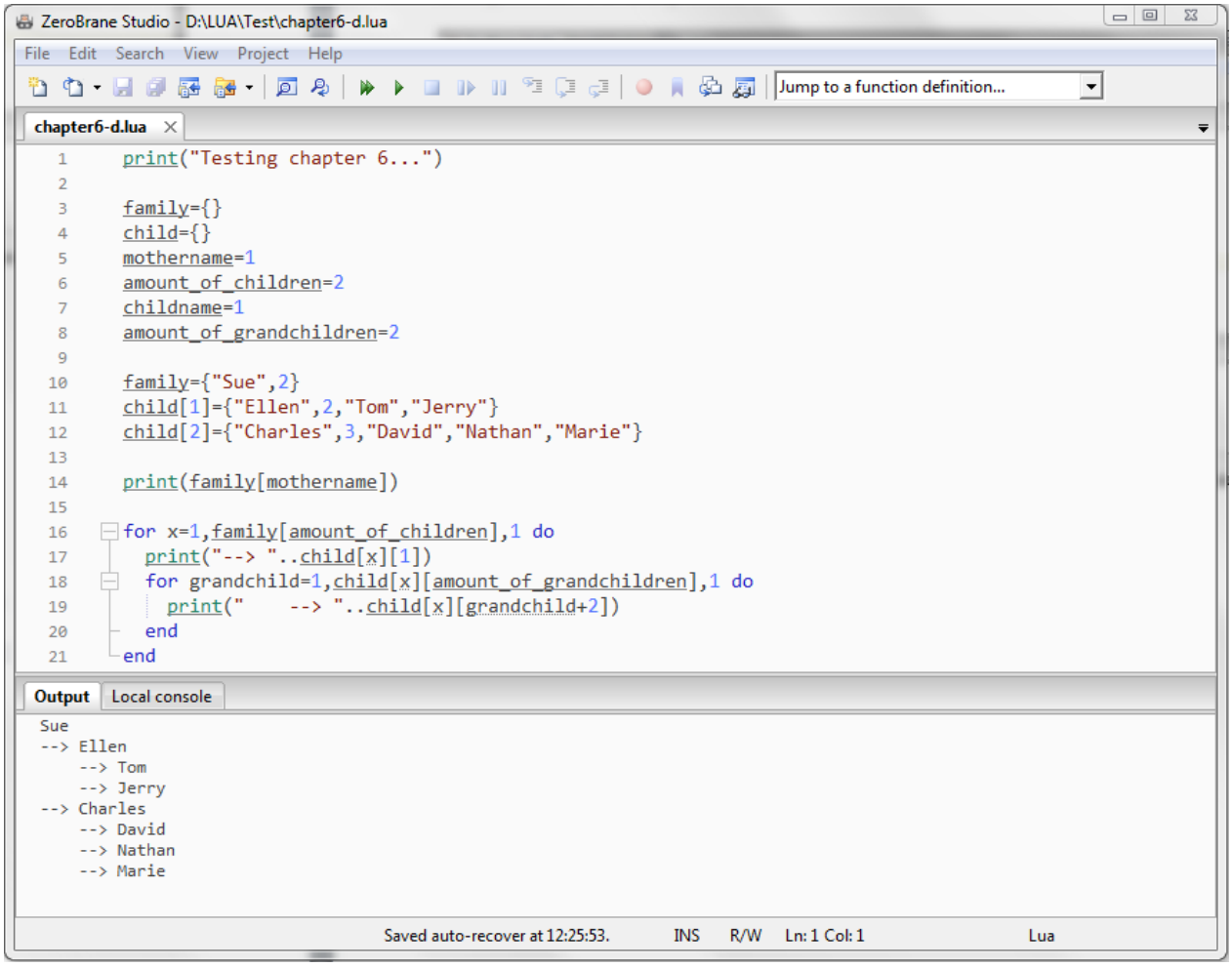
And we define our table data :

```
family={"Sue",2}
child[1]={"Ellen",2,"Tom","Jerry"}
child[2]={"Charles",3,"David","Nathan","Marie"}
```

We can add a value in our tables that indicates how much children exist :

for the ”family” table	mothername=”Sue”
	amount_of_children=2
for the “child” table	childname=”Ellen” and “Charles”
	amount_of_grandchildren=2 for “Ellen” and 3 for “Charles”

Let’s print out the family-tree :



Line #

3-4 table definition

5-8 value name definition (pointers)

10 "family" table content definition
- first the mother's name
- then the amount of children she has

11-12 "children" table content definitions
- first the mothers name of the grandchildren
- then the amount of children she or he has
- then the names of the grandchildren (as many as defined)

14 print the name of the mother

16 outer loop : execute this as many times as there are children defined in the "family" table (in this case 2)

17 print the name of the mother's children " --> ".. gives some extra space in front of the name

18 inner loop : execute this as many times as there are children defined in each of the the "child" tables (in this case 2 for "Ellen" and 3 for "Charles")

19 print the name of the grandchildren " --> ".. gives some extra space in front of the name
notice : for the names of all grandchildren we use the position grandchild+2, the reason is that we have to skip the first 2 values (name of the child and "amount of grandchildren"-indicator, otherwise the result would be for "Ellen" :
--> Ellen
--> 3
--> Tom
and for "Charles" :
--> Charles
--> 3
--> David

20 end statement of the inner loop

21 end statement of outer loop

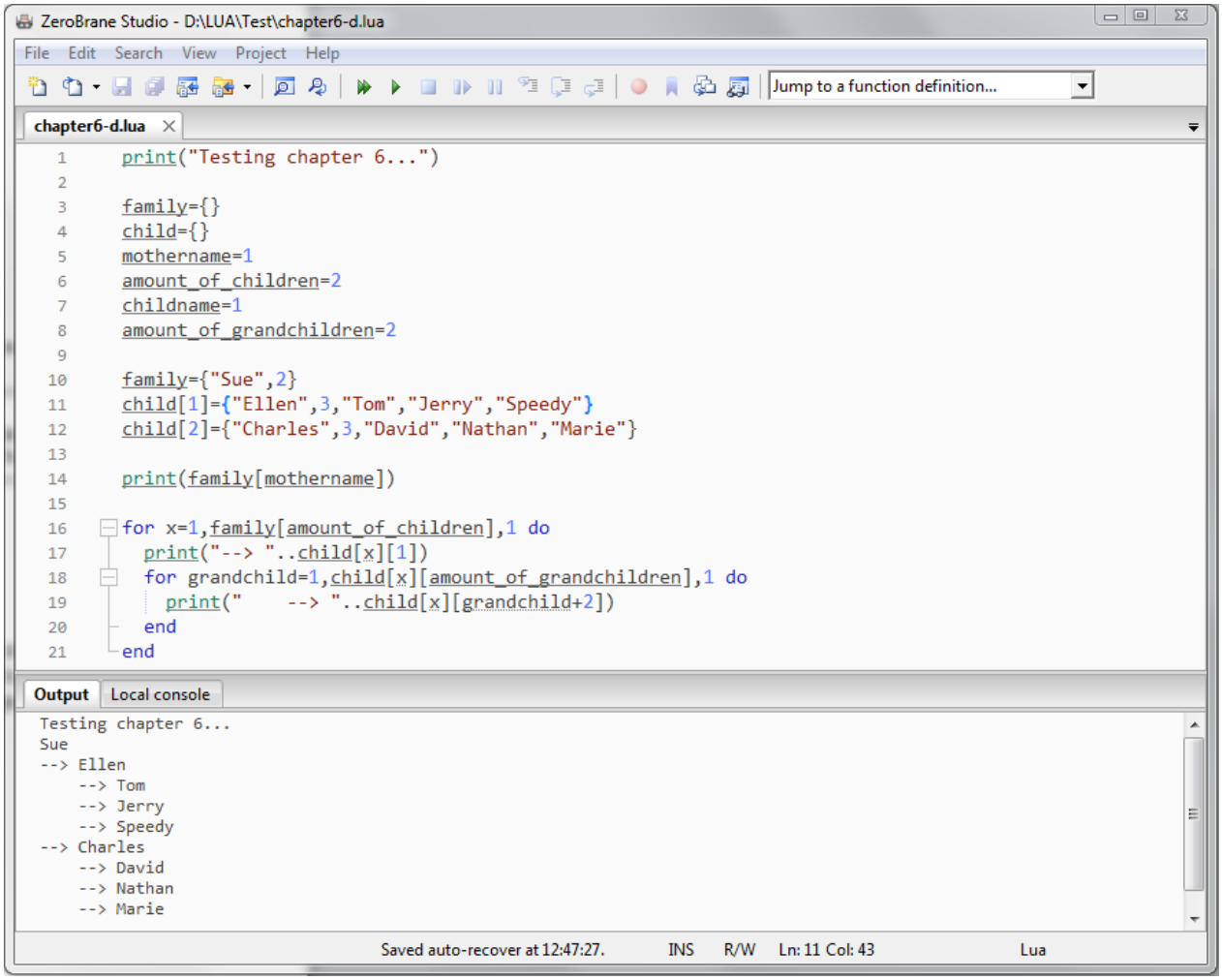
The for-loop can be replaced by a while- or repeat-loop.

In this example we used a counter to indicate how many children or grandchildren are present. This gives you a safe way to execute your loop. You have to update this data if there is changed something in the table data, for example if "Ellen" gives birth to another child, you need to change line 11 :

```
child[1]={"Ellen",3,"Tom","Jerry",”Speedy”}
```

Once you have structured your data, it's quite easy to make changes, and mostly your "execute" code has to remain unchanged.

Finally we will test our latest change :



There is also a way to work with a table without a "safe-counter", but this will not be explained here.

1. Remember some theory…

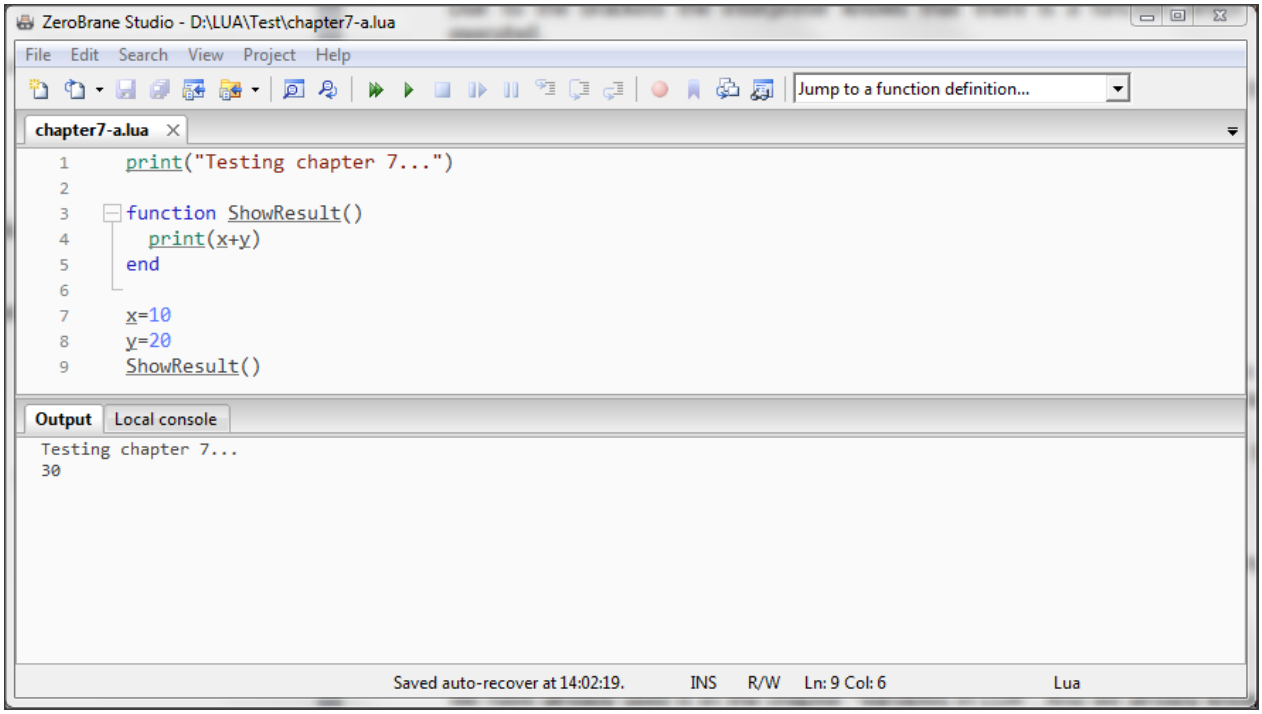
In a previous chapter functions are allready explained. This is what is allready known :

- they have a name
- the name is followed by brackets ()
- they have to be called explicitly
- they execute something (if called)
- they include a block of code (which can also include several blocks again)
- they are closed with the keyword "**end**"

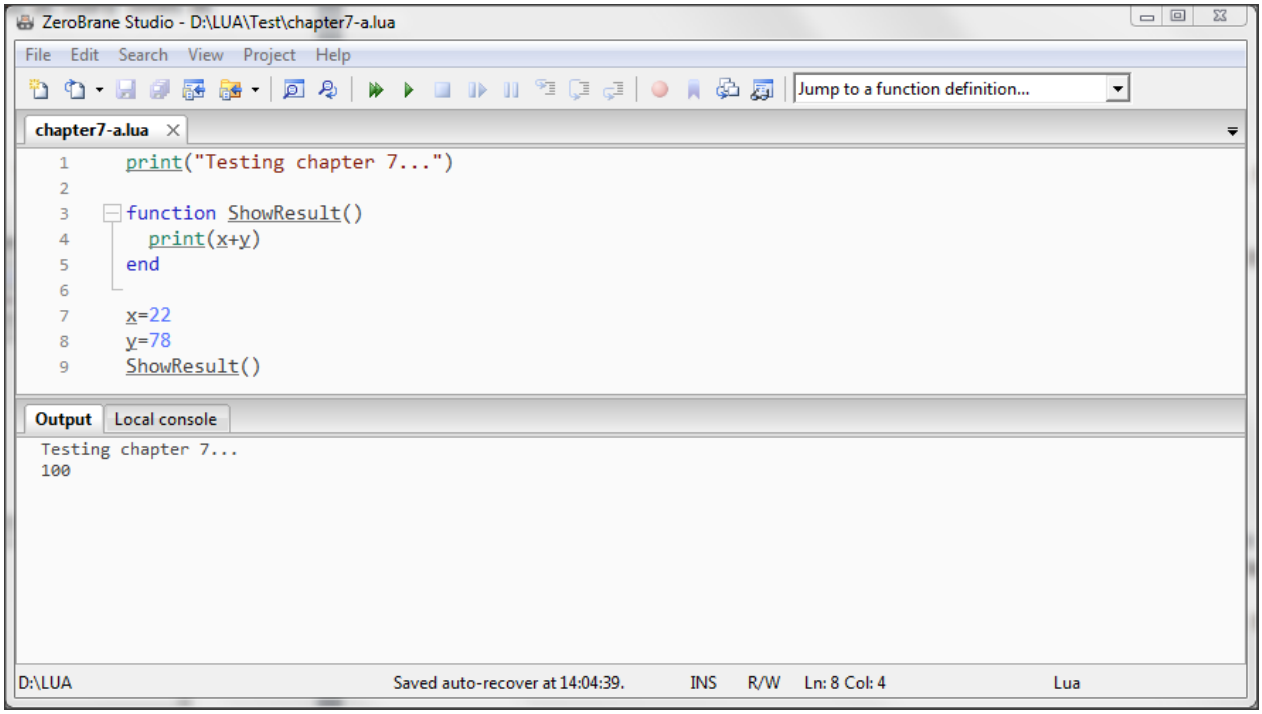
The main goal of a function is to combine code that can be executed as many times as called, even with changing incoming values and returning values if necessary.

2. Calling a function without a value

It's best to use an example for explaining the calling of a function. So let's make some code that calls a function that is printing a result of a calculation :

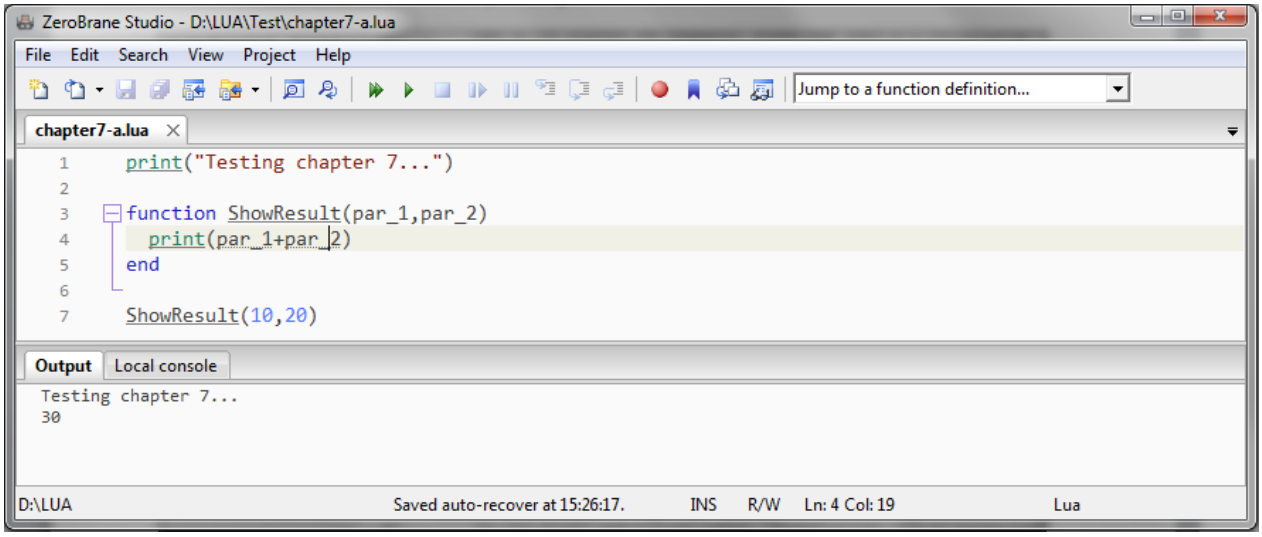


Suppose we change the values of X and Y, the same function is called without changing it's content, and prints the result as wanted :



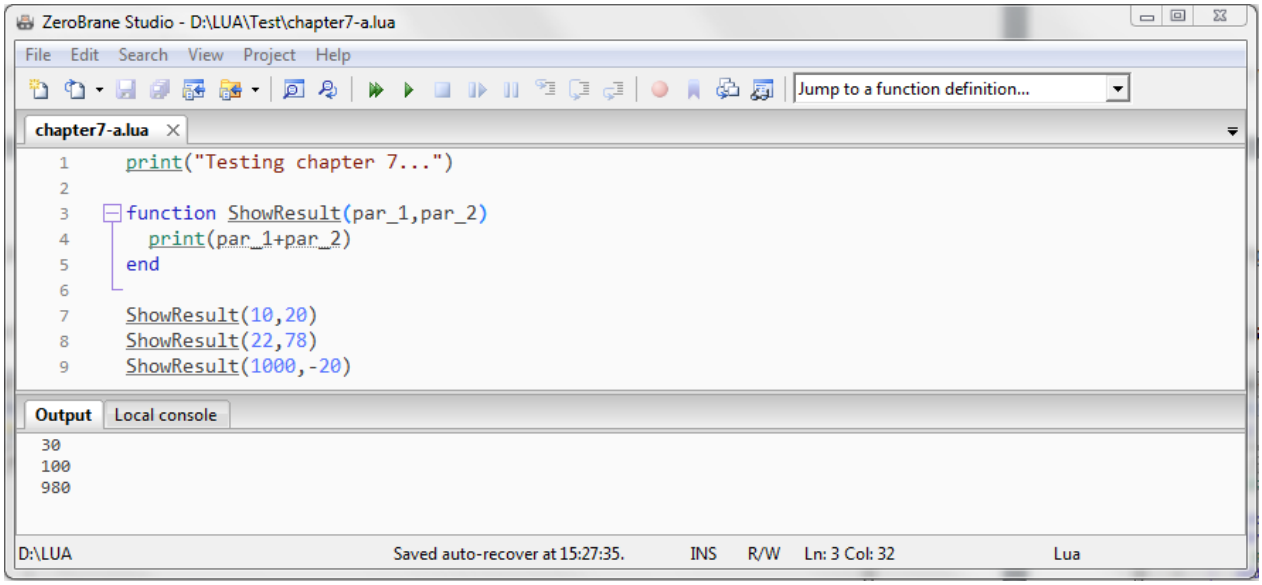
3. Calling a function with (a) value(s)

A function can accept values, or called "parameters'". The parameters are local, and the value is used inside the function.



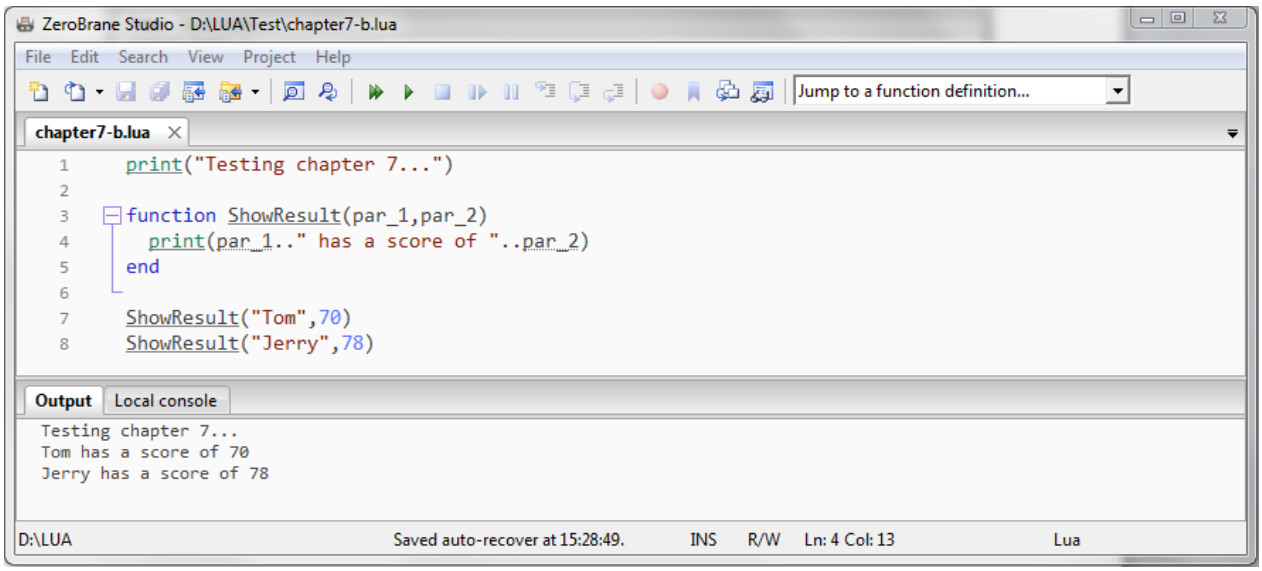
Here you can see that we removed x and y, and call the function with two numbers, separated by a comma. They are transferred to the function and "*adopted*" by the variables par_1 and par_2. In the result they are added to each other (par_1+par_2). Again, if you change those values, par_1 and par_2 will change when calling the function.

In this example we call the same function 3 times with different values :

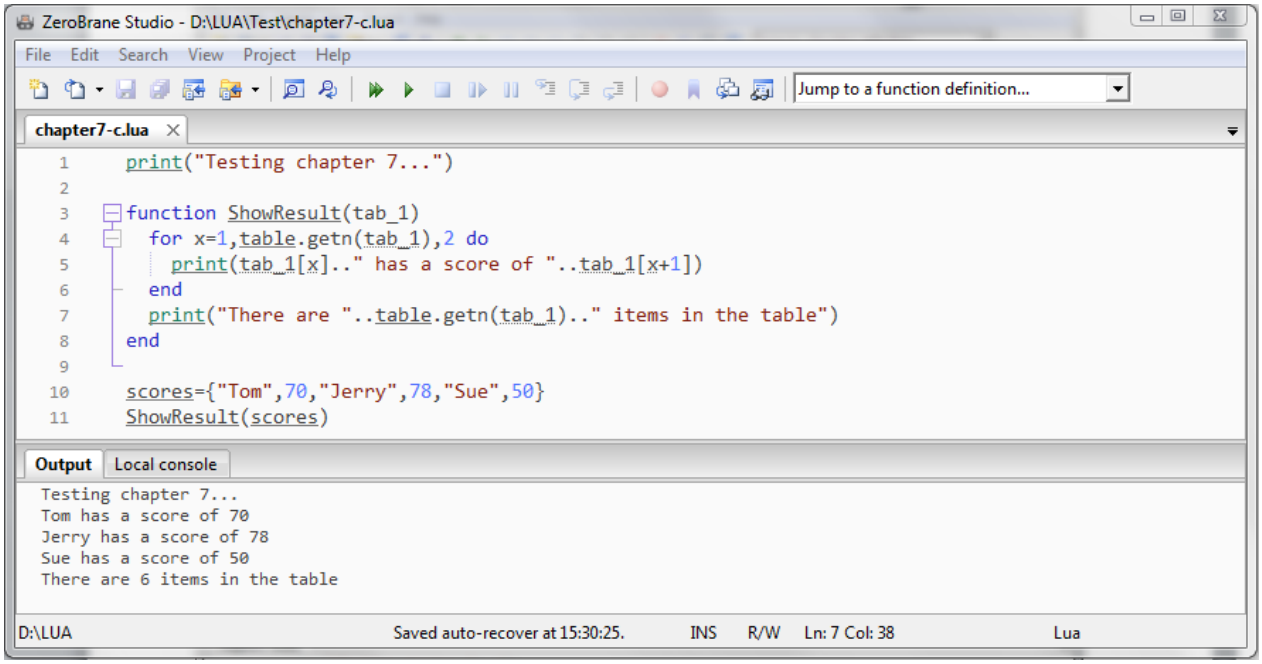


You can see that the code in the function itself remains the same.

In this example we used numbers as parameters, but you can also use text (and for example a combination of both) :



Even tables are allowed :



Here we define a table in line 9 and call the function in line 10. The whole table is passed to parameter tab_1.

A special item is used in line 4 :

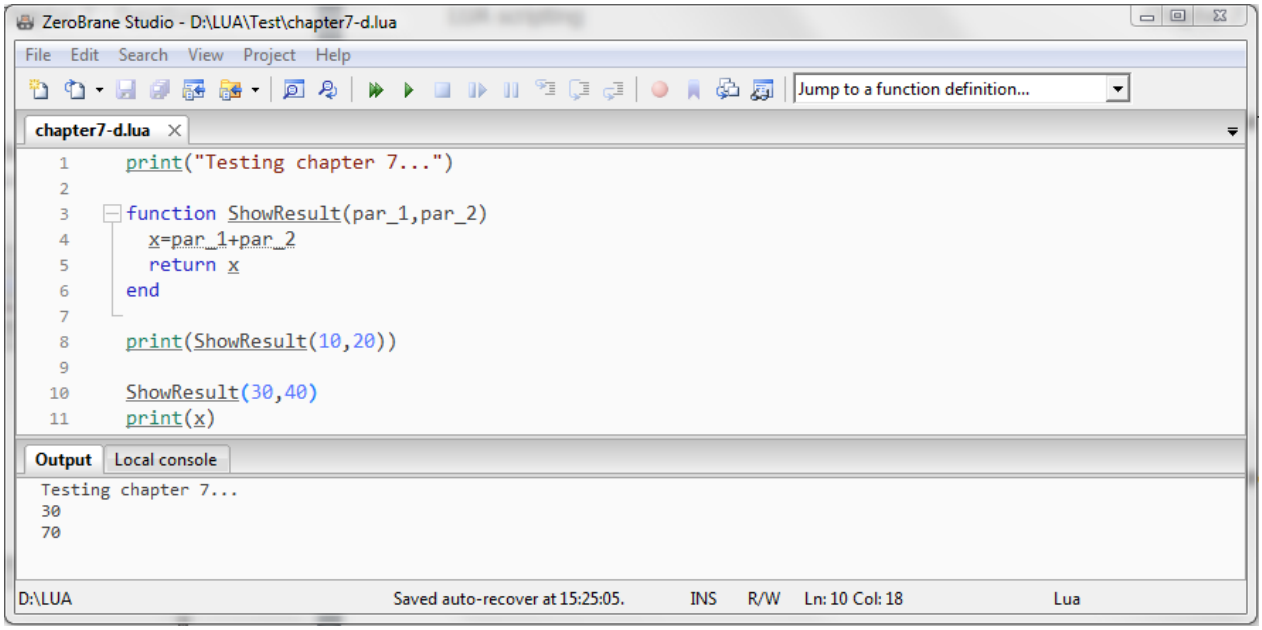
table.getn(tab1**)**

With this LUA-table-function **table.getn(n)** we can get information about the amount of elements in a table. This is again used in line 7 to show the number of items in the table.

To extend the result, e.g. just expand the table definition in line 10 with 2 values, 1 name and 1 score.

4. Returning a value

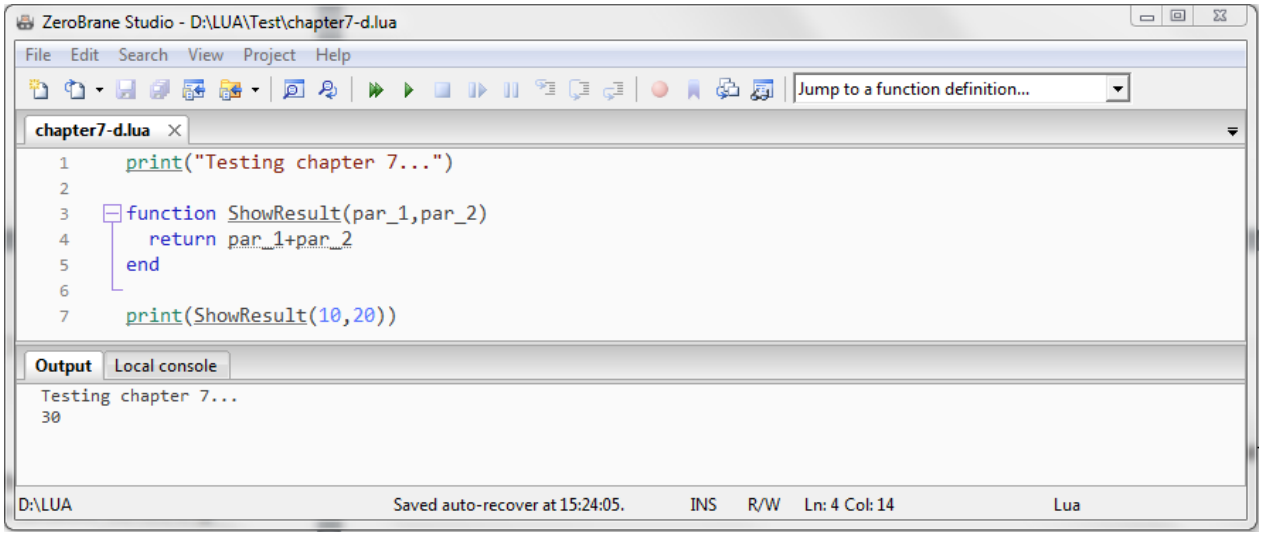
A function can not only accept values, but also return a result. In the previous example the result was printed in the function itself. Suppose we just want to calculate something in a function and send it back as a result :



In the function the sum of par_1 and par_2 is stored in variable x, and in line 8 that variable is send back to the calling line of the function.

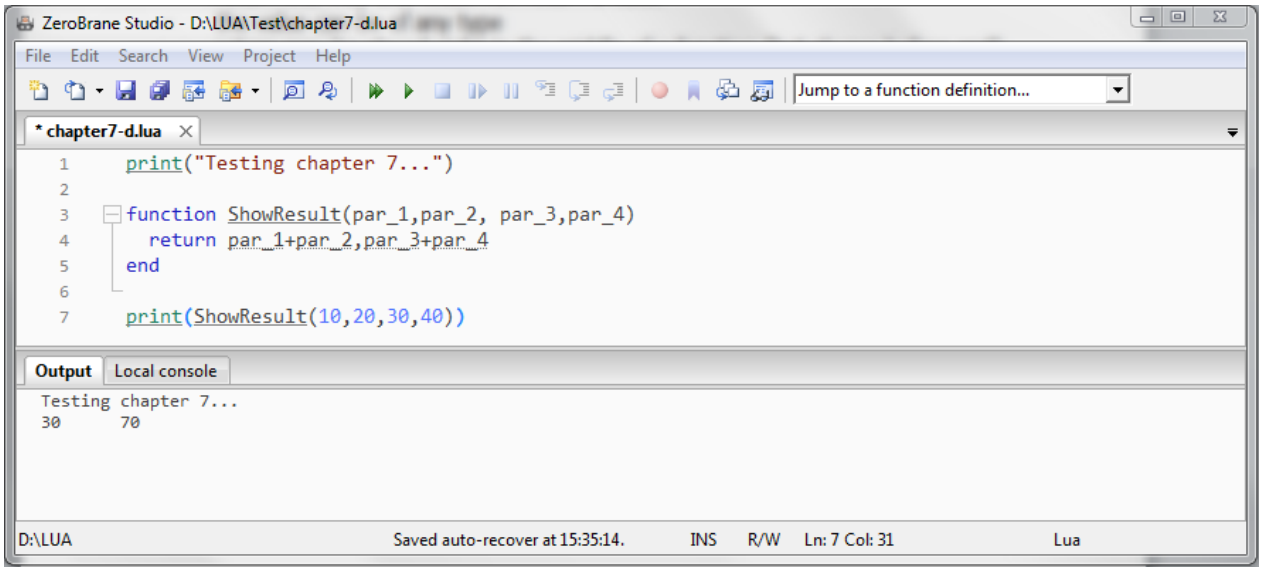
Either line 8 alone or line 9 together with line 10 give the same result of the function on the screen.

Finally you can make some things shorter :

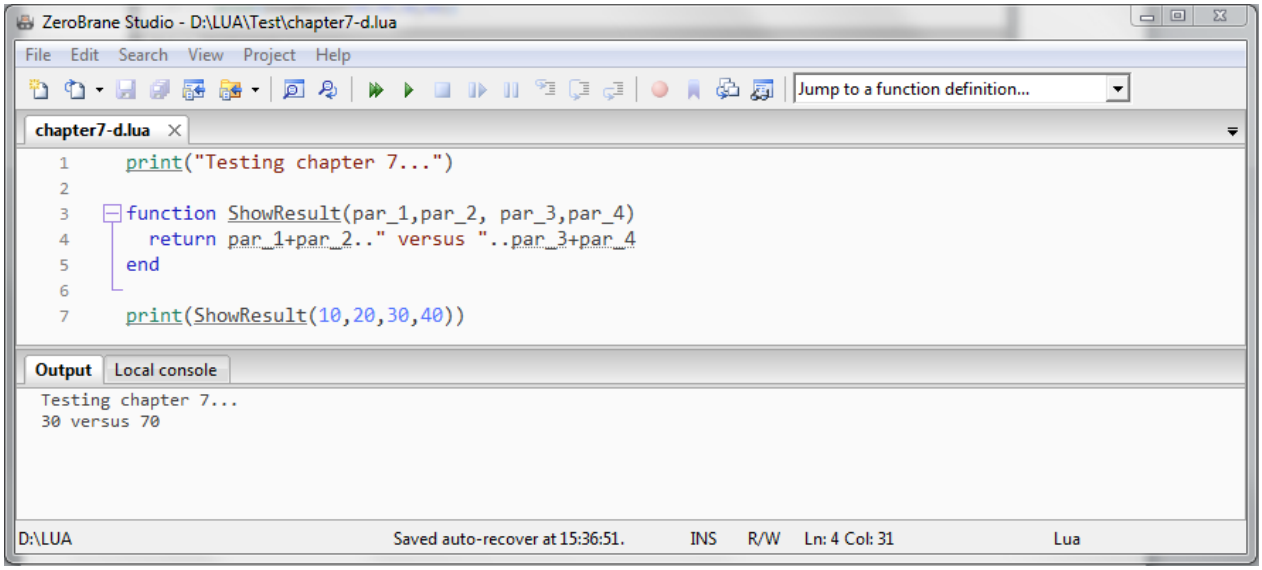


It is also possible to return more than one parameter.

- These values are separated by a comma :



- These values are concatenated :



Remarks :

- return has to be always the last keyword before an end keyword
- return is finishing the function
- return “this” may and must not return a value
- the value can be of any type
- return may be placed in a block in a function (but always before end)