



Atom-Based Software Engineering



Lua Guide

Lua Guide



Read some details [about Lua](#).

This guide will help you get up to speed with the Lua programming language, and how you can apply it to your ABSE models. You just need a small subset of Lua for most tasks. To get up to speed with Lua in just 5 minutes, read this page : [Lua in ABSE](#).

For a more detailed reading, read these pages:

[Data Types](#)

[Variables](#)

[Reserved Keywords](#)

[Strings](#)

[Statements](#)

[Functions](#)

[Expressions](#)

[Comments](#)

[Visibility Rules](#)

[Error Handling](#)

[Garbage Collection](#)

[Standard Libraries](#)

[The Complete Lua Syntax](#)

Ultimately, you should refer to the Lua [official site](#).

About Lua

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Lua is intended to be used as a powerful, light-weight scripting language. It's also a very fast one, make it suitable for long, intensive programs like those internally generated by AtomWeaver.

Being an extension language, Lua has no notion of a *main* program: it only works embedded in a host client, called the *embedding program* or simply the *host*. AtomWeaver is Lua's host. The host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code.

Lua in ABSE

To implement your Atom Templates, you need to do it using the Lua language. Lua by itself is already a language with simple syntax and just a few keywords.

But to use Lua in ABSE Atoms is even simpler than that. What you need to learn can be summed up to:

Function calls

Because ABSE makes its features available as a function library, you just need to make calls to [functions](#). Some examples:

```
abstract()
param_text("def", "my_param", "My Parameter", "This is my parameter")
line("Generated text")
```

Variables

Besides ABSE variables, you can use Lua [variables](#) to perform additional processing or string preparation tasks. You should always use local variables (through the use of the *local* qualifier) so that they can be released when an Atom finishes its job.

```
local myVar = "The Atom is set to " .. var("my_atom_var")
```

Control structures

You will probably need to make some decisions and some loops to generate code, therefore learning Lua's [control structures](#) is useful. Some examples:

```
if var("has_flag") then
    line("flag = 0;")
end

for i = 1,10 do
    line("This is line number " .. i)
end
```

Data Types

Lua is a dynamically typed language. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

Because all values in Lua are first-class values, they can be stored in variables, passed as arguments to other functions, and returned as results.

There are eight basic types in Lua: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread*, and *table*.

Tables, functions, threads, and (full) userdata values are "objects": variables do not actually contain these values, only references to them. Therefore, assignment, parameter passing, and function returns always manipulate references to such values: these operations do not copy data around.

The function *type* returns a string describing the type of a given value.

Userdata Type

The type *userdata* is provided to allow arbitrary C data to be stored in Lua variables. This type corresponds to a block of raw memory and has no pre-defined operations in Lua, except assignment and identity test.

Userdata values cannot be created or modified in Lua, only through its C API. This guarantees the integrity of data owned by AtomWeaver.

Thread type

The type *thread* represents independent threads of execution and it is used to implement coroutines.

This type is not useful in an ABSE model.

Table type

The type *table* implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any value except *nil*. Tables can be heterogeneous; that is, they can contain values of all types except *nil*.

Tables are the only data structure mechanism in Lua. They may be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc. To represent records, Lua uses the field name as an index. The language supports this representation by providing *a.name* as syntactic sugar for *a["name"]*.

Because functions are first-class values, table fields may contain functions.

Variables

Variables are places that store values. There are three kinds of variables in Lua: global variables, local variables, and table fields.

There is no variable that has no value. Even uninitialized variables have a value. The default value for variables is *nil*, which indicates the absence of a value. Valid variable identifiers consist of letters, digits, and underscores. Identifiers may not start with a digit.

Any variable is assumed to be global unless explicitly declared as a local using the *local* qualifier:

```
var1 = 0 -- Global variable
local var2 = 0 -- Local variable
```

On Atoms you should use local variables whenever possible.

Reserved Keywords

The following keywords are reserved by the language and cannot be used as names:

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

You can, however, have Atom Template parameters with any of the above keywords.

Strings

Because all code generation inside AtomWeaver using ABSE is done through strings, this is one important topic.

Literal strings can be delimited by matching single or double quotes, and can contain the following C-like escape sequences: `\a` (bell), `\b` (backspace), `\f` (form feed), `\n` (new line), `\r` (carriage return), `\t` (horizontal tab), `\v` (vertical tab), `\\` (backslash), `\"` (quotation mark [double quote]), and `\'` (apostrophe [single quote]).

A backslash followed by a real new line results in a new line in the string. A character in a string may also be specified by its numerical value using the escape sequence `\ddd`, where *ddd* is a sequence of up to three decimal digits. If a numerical digit follows, then it must be expressed using three digits. Strings in Lua may contain any 8-bit value, including embedded zeros, which can be specified as `\0`.

These strings are equivalent and represent:

```
hello
123"
```

```
a = 'hello\n123''
a = "hello\n123\""
```

```
a = '\104ello\10\04923''
```

Lua has a standard [library for string manipulation](#).

Long Brackets

Literal strings can also be defined using a long format enclosed by *long brackets*. We define an opening long bracket of level n as an opening square bracket followed by n equal signs followed by another opening square bracket.

So, an opening long bracket of level 0 is written as `[[`, an opening long bracket of level 1 is written as `[=[`, and so on. A closing long bracket is defined similarly; for instance, a closing long bracket of level 4 is written as `]====]`. A long string starts with an opening long bracket of any level and ends at the first closing long bracket of the same level.

Literals in this bracketed form may run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. They may contain anything except a closing bracket of the proper level.

These strings are equivalent and represent:

```
hello
123"
```

```
a = [[hello
123"]]
```

```
a = [=[
hello
123"]=]
```

What if you wanted to print two close square brackets inside a long string?

```
print([[ Close ]] brackets]])
```

This will generate an error. You can solve this by using long brackets of level 1:

```
print([=[ Close ]] brackets]=])
```

Long brackets are very useful in ABSE, because you can use a large block of source code in one function call. The `code()` command was specially created to use long brackets:

```
code([[
  class @class_name : public $base_class
  {
    private:
```

```

    {{class_members}}
    public:
    {{class_functions}}
  };
11)

```

Statements

Lua supports an almost conventional set of statements, similar to those in Pascal or C. This set includes assignment, control structures, function calls, and variable declarations.

Chunks

The unit of execution of Lua is called a chunk. A chunk is simply a sequence of statements, which are executed sequentially.

Lua handles a chunk as the body of an anonymous function with a variable number of arguments. As such, chunks can define local variables, receive arguments, and return values.

A chunk may be stored in a file or in a string inside the host program. When a chunk is executed, first it is pre-compiled into instructions for a virtual machine, and then the compiled code is executed by an interpreter for the virtual machine.

When you generate code in AtomWeaver, it loads all your Atom Templates into the Lua engine for pre-compilation. Then, it executes every Atom Instance in the model by calling the appropriate sections of their Atom Templates.

Blocks

A block is a list of statements; syntactically, a block is the same as a chunk.

```

do
    ... -- any number of statements here
end

```

Explicit blocks are useful to control the scope of variable declarations. See [Visibility Rules](#). Explicit blocks are also sometimes used to add a return or break statement in the middle of another block.

Assignment

Lua allows multiple assignment. Therefore, the syntax for assignment defines a list of variables on the left side and a list of expressions on the right side. The elements in both lists are separated by commas.

Before the assignment, the list of values is adjusted to the length of the list of variables. If there are more values than needed, the excess values are thrown away. If there are fewer values than needed, the list is extended with as many *nil*'s as needed. If the list of expressions ends with a function call, then all values returned by this call enter in the list of values, before the adjustment (except when the call is enclosed in parentheses).

The assignment statement first evaluates all its expressions and only then are the assignments performed. The code

```
i = 10
i, a[i] = i+1, 33
```

sets `a[10]` to 33, without affecting `a[11]` because the `i` in `a[i]` is evaluated (to 10) before it is assigned 11. Similarly, the line

```
a, b = b, a
```

exchanges the values of `a` and `b`.

Control Structures

The control structures *if*, *while*, and *repeat* have the usual meaning and familiar syntax. Lua also has a *for* statement, in two distinct forms (see [For Statement](#)).

```
while expression do
    ...
end

repeat
    ...
until expression

if expression then
    ...
elseif expression then
    ...
else
    ...
end
```

In the *repeat–until* loop, the inner block does not end at the *until* keyword, but only after the condition. So, the condition can refer to local variables declared inside the loop block.

The *break* statement is used to terminate the execution of a *while*, *repeat*, or *for* loop, skipping to the next statement after the loop. A *break* statement ends the innermost enclosing loop.

The *break* statement can only be written as the last statement of a block.

Return statement

The *return* statement is used to return values from a function or a chunk (which is just a function). Functions and chunks may return more than one value:

```
...
return 1
```

```
...
return 1, a, "text"
```

The *return* statement can only be written as the last statement of a block.

For Statement

The *for* statement has two forms: one numeric and one generic.

The numeric *for* loop repeats a block of code while a control variable runs through an arithmetic progression:

```
-- get even numbers
for i = 0, 100, 2 do
  print i
end
```

- All three control expressions are evaluated only once, before the loop starts. They must all result in numbers.
- If the third expression (the step) is absent, then a step of 1 is used.
- The loop variable *i* is local to the loop: you cannot use its value outside the *for* loop. If you need this value, assign it to another variable before breaking or exiting the loop.

The generic *for* statement works over functions, called iterators. On each iteration, the iterator function is called to produce a new value, stopping when this new value is nil.

```
tn = {100, 200, 300}
for price in values(tn) do
  print(price)
end
```

- *values(tn)* is evaluated only once. Its results are an iterator function, a state, and an initial value for the first iterator variable.
 - The *price* variable is local to the loop: you cannot use its value outside the *for* loop. If you need this value, assign it to another variable before breaking or exiting the loop.
-

Functions

Functions are the main mechanism for abstraction of statements and expressions in Lua.

A function definition has a conventional syntax:

```
function add_list(a)
    local sum = 0
    for index, value in ipairs(a) do
        sum = sum + value
    end
    return sum
end
```

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

Function Parameters

Parameters work exactly as local variables, initialized with the values of the arguments passed in the function call. You can call a function with a number of arguments different from its number of parameters. Lua adjusts the number of arguments to the number of parameters, as it does in a multiple assignment:

extra arguments are thrown away; extra parameters get *nil*. For instance, if you have a function like

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

you will get the following mapping from arguments to parameters:

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

Although this behavior can lead to programming errors, it is also useful, especially for default arguments.

Multiple Return Values

An unconventional, but quite convenient feature of Lua is that functions may return multiple results. Several pre-defined functions in Lua return multiple values. An example is the *string.find* function, which locates a pattern in a string. This function returns two indices when it finds the pattern: the index of the character where the pattern match starts and the one where it ends. A multiple assignment allows the program to get both results:

```
start_pos, end_pos = string.find("ABSE likes Lua", "likes")
```

Functions written in Lua also can return multiple results, by listing them all after the return keyword. For instance, a function to find the highest integer in an array can return both the maximum value and its location:

```
function max_value(list)
    local maxi = 1
    local max = list[maxi]
    for index, value in ipairs(list) do
        if value > max then
            maxi = index
            max = value
        end
    end
    return max, maxi
end
```

Lua always adjusts the number of results from a function to the circumstances of the call. When we call a function as a statement, Lua discards all results from the function. When we use a call as an expression, Lua keeps only the first result. We get all results only when the call is the last (or the only) expression in a list of expressions. These lists appear in four constructions in Lua: multiple assignments, arguments to function calls, table constructors, and return statements. To illustrate all these cases, we will assume the following definitions for the next examples:

```
function res0 () end    -- returns no results
function res1 () return "a" end    -- returns 1 result
function res2 () return "a","b" end    -- returns 2 results
```

In a multiple assignment, a function call as the last (or only) expression produces as many results as needed to match the variables:

```
x,y = res2()    -- x="a", y="b"
x = res2()     -- x="a", "b" is discarded
x,y,z = 10,res2()    -- x=10, y="a", z="b"
```

If a function has no results, or not as many results as we need, Lua produces *nils* for the missing values:

```
x,y = res0()    -- x=nil, y=nil
x,y = res1()    -- x="a", y=nil
x,y,z = res2()  -- x="a", y="b", z=nil
```

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

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

```
t = {res0()}    -- t = {} (an empty table)
```

```
t = {res1()}    -- t = {"a"}
t = {res2()}    -- t = {"a", "b"}
```

As always, this behavior happens only when the call is the last in the list; calls in any other position produce exactly one result:

Finally, a statement like `return myFunc()` returns all values returned by function `myFunc`.

Variable Number of Arguments

Functions in Lua can receive a variable number of arguments. The `print` function is one example.

The following function returns the result of summing all its arguments:

```
function sum_args(...)
    local sum = 0
    for index, value in ipairs{...} do
        sum = sum + value
    end
    return sum
end

print(sum_args(5, 9, 10, 2))    --> 26
```

The three dots (...) in the parameter list indicate that the function accepts a variable number of arguments. When this function is called, all its arguments are collected internally; these collected arguments are called the *varargs* (*variable arguments*) of the function. A function can access its *varargs* using the three dots as an expression. In the above example, the expression {...} results in an array with all collected arguments. The function then traverses the array to add its elements.

The expression ... behaves like a [multiple return function](#) returning all *varargs* of the current function.

Expressions

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

Arithmetic Operators

Lua supports the usual arithmetic operators: the binary + (addition), - (subtraction), * (multiplication), / (division), % (modulo), and ^ (exponentiation); and unary - (negation). If the operands are numbers, or strings that can be converted to numbers, then all operations have the usual meaning. Exponentiation works for any exponent.

Relational Operators

Lua provides the following relational operators:

< > <= >= == ~=

These operators always result in *true* or *false*.

The operator `==` tests for equality. The operator `~=` is the negation of equality. We can use these operators to compare any two values. If the values have different types, Lua will consider them as not equal. Otherwise, Lua compares them according to their type.

nil is equal only to itself.

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

Logical Operators

The logical operators in Lua are *and*, *or*, and *not*. Like control structures, all logical operators consider both *false* and *nil* as false and anything else as true.

Both *and* and *or* use short-cut evaluation; that is, the second operand is evaluated only if necessary.

String Concatenation

The string concatenation operator in Lua is denoted by two dots (`..`). If both operands are strings or numbers, they are converted to strings.

Strings in Lua are immutable values, and therefore the concatenation operator always creates a new string, without any modification to its operands.

The Length Operator

The length operator is denoted by the unary operator `#`. The length of a string is its number of bytes, that is, the usual meaning of string length when each character is one byte.

The length operator ``#'` returns the last index (or the size) of an array or list.

Precedence

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

```

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

```

As usual, you can use parentheses to change the precedence in an expression. The concatenation ('..') and exponentiation ('^') operators are right associative. All other binary operators are left associative.

Table Constructors

Table constructors are expressions that create tables. Every time a constructor is evaluated, a new table is created. Constructors can be used to create empty tables, or to create a table and initialize some of its fields.

```
t = {} -- create a table and store its reference in 't'
```

Each field of the form $[exp1] = exp2$ adds to the new table an entry with key $exp1$ and value $exp2$. A field of the form $name = exp$ is equivalent to $["name"] = exp$. Finally, fields of the form exp are equivalent to $[i] = exp$, where i are consecutive numerical integers, starting with 1. Fields in the other formats do not affect this counting. For example,

```
a = { [f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45 }
```

is equivalent to

```

do
  local t = {}
  t[f(1)] = g
  t[1] = "x"      -- 1st exp
  t[2] = "y"      -- 2nd exp
  t.x = 1         -- t["x"] = 1
  t[3] = f(x)     -- 3rd exp
  t[30] = 23
  t[4] = 45       -- 4th exp
  a = t
end

```

Function Calls

Function calls can be executed as statements:

```
function my_function()
    ...
    return a, b
end

my_function()
```

In this case, all returned values are thrown away.

A call of the form *return functioncall()* is called a *tail call*. Lua implements proper tail calls (or proper tail recursion): in a tail call, the called function reuses the stack entry of the calling function. Therefore, there is no limit on the number of nested tail calls that a program can execute.

Comments

A comment starts with a double hyphen (--) anywhere outside a string. If the text immediately after -- is not an opening long bracket, the comment is a short comment, which runs until the end of the line. Otherwise, it is a long comment, which runs until the corresponding closing long bracket. Long comments are frequently used to disable code temporarily.

```
-- Short comment

--[
...
Long comment
...
--]
```

Visibility Rules

When a function is written enclosed in another function, it has full access to local variables from the enclosing function; this feature is called lexical scoping. The scope of variables begins at the first statement after their declaration and lasts until the end of the innermost block that includes the declaration. Consider the following example:

```
x = 10                -- global variable
do                   -- new block
    local x = x       -- new 'x', with value 10
    print(x)          --> 10
    x = x+1
    do               -- another block
        local x = x+1 -- another 'x'
        print(x)     --> 12
    end
    print(x)         --> 11
end
print(x)            --> 10 (the global 'x')
```

In a declaration like *local x = x*, the new *x* being declared is not in scope yet, and so the second *x* refers to the outside variable.

Error Handling

Because Lua is an embedded extension language, all Lua actions start from C code in AtomWeaver calling a function from the Lua library. Whenever an error occurs during Lua compilation or execution, control returns to C, which can take appropriate measures (such as printing an error message).

Lua code can explicitly generate an error by calling the *error* function.

Garbage Collection

Lua performs automatic memory management. This means that you have to worry neither about allocating memory for new objects nor about freeing it when the objects are no longer needed.

Lua manages memory automatically by running a *garbage collector* from time to time to collect all *dead objects* (that is, those objects that are no longer accessible from Lua).

All objects in Lua are subject to automatic management: tables, userdata, functions, threads, and strings.

Standard Libraries

The standard Lua libraries provide useful functions that are implemented directly through the C API. Some of these functions provide essential services to the language (e.g., *type* and *getmetatable*); others provide access to "outside" services (e.g., I/O); and others could be implemented in Lua itself, but are quite useful or have critical performance requirements that deserve an implementation in C (e.g., *sort*).

All libraries are implemented through the official C API and are provided as separate C modules. Useful standard libraries are:

[Basic library](#)

[String manipulation](#)

[Table manipulation](#)

[Mathematical functions](#)

[Input and output](#)

[Operating system facilities](#)

Except for the Basic library, each library provides all its functions as fields of a global table or as methods of its objects.

Basic Functions

The basic library provides some core functions to Lua.

dofile (filename)

Opens the named file and executes its contents as a Lua chunk. Returns all values returned by the chunk. In case of errors, *dofile* propagates the error to its caller.

ipairs (t)

Returns three values: an iterator function, the table *t*, and 0, so that the construction

```
for i,v in ipairs(t) do
    ...
end
```

will iterate over the pairs $(1,t[1])$, $(2,t[2])$, ..., up to the first integer key absent from the table.

pairs (t)

Returns three values: the next function, the table *t*, and nil, so that the construction

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

will iterate over all key–value pairs of table *t*.

tonumber (e [, base])

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then *tonumber* returns this number; otherwise, it returns *nil*.

An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter 'A' (in either upper or lower case) represents 10, 'B' represents 11, and so forth, with 'Z' representing 35. In base 10 (the default), the number may have a decimal part, as well as an optional exponent part. In other bases, only unsigned integers are accepted.

tostring (e)

Receives an argument of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use `string.format`.

If the metatable of *e* has a `__tostring` field, then *tostring* calls the corresponding value with *e* as argument, and uses the result of the call as its result.

type (v)

Returns the type of its only argument, coded as a string. The possible results of this function are "nil" (a string, not the value nil), "number", "string", "boolean", "table", "function", "thread", and "userdata".

String Manipulation

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Lua, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position -1, and so on.

The string library provides all its functions inside the table `string`. It also sets a metatable for strings where the `__index` field points to the string table. Therefore, you can use the string functions in object-oriented style. For instance, `string.byte(s, i)` can be written as `s:byte(i)`.

string.byte (s [, i [, j]])

Returns the internal numerical codes of the characters `s[i]`, `s[i+1]`, ..., `s[j]`. The default value for `i` is 1; the default value for `j` is `i`. Please note that numerical codes are not necessarily portable across platforms.

string.char (...)

Receives zero or more integers. Returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its corresponding argument. Please note that numerical codes are not necessarily portable across platforms.

string.find (s, pattern [, init [, plain]])

Looks for the first match of `pattern` in the string `s`. If it finds a match, then `find` returns the indices of `s` where this occurrence starts and ends; otherwise, it returns `nil`. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and may be negative. A value of `true` as a fourth, optional argument `plain` turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in `pattern` being considered "magic". Note that if `plain` is given, then `init` must be given as well.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

string.format (formatstring, ...)

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the `printf` family of standard C functions. The only differences are that the options/modifiers `*`, `l`, `L`, `n`, `p`, and `h` are not supported and that there is an extra option, `q`. The `q` option formats a string in a form suitable to be safely read back by the Lua interpreter: the string is written between double quotes, and all double quotes, new lines, embedded zeros, and backslashes in the string are correctly escaped when written.

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string.

This function does not accept string values containing embedded zeros, except as arguments to the `q` option.

string.gmatch (s, pattern)

Returns an iterator function that, each time it is called, returns the next captures from `pattern` over string `s`. If `pattern` specifies no captures, then the whole match is produced in each call.

As an example, the following loop

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
    print(w)
end
```

will iterate over all the words from string `s`, printing one per line.

string.gsub (s, pattern, repl [, n])

Returns a copy of `s` in which all occurrences of the `pattern` have been replaced by a replacement string specified by `repl`, which may be a string, a table, or a function. `gsub` also returns, as its second value, the total number of substitutions made.

If `repl` is a string, then its value is used for replacement. The `%` character works as an escape character: any sequence in `repl` of the form `%n`, with `n` between 1 and 9, stands for the value of the `n`-th captured substring (see below). The sequence `%0` stands for the whole match. The sequence `%%` stands for a single `%`.

If `repl` is a table, then the table is queried for every match, using the first capture as the key; if the `pattern` specifies no captures, then the whole match is used as the key.

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the `pattern` specifies no captures, then the whole match is passed as a sole argument.

If the value returned by the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is `false` or `nil`, then there is no replacement (that is, the original match is kept in the string).

The optional last parameter `n` limits the maximum number of substitutions to occur. For instance, when `n` is 1 only the first occurrence of `pattern` is replaced.

Here are some examples:

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

string.len (s)

Receives a string and returns its length. The empty string "" has length 0. Embedded zeros are counted, so "a\000bc\000" has length 5.

string.lower (s)

Receives a string and returns a copy of this string with all uppercase letters changed to lowercase. All other characters are left unchanged. The definition of what an uppercase letter is depends on the current locale.

string.match (s, pattern [, init])

Looks for the first match of *pattern* in the string *s*. If it finds one, then *match* returns the captures from the pattern; otherwise it returns *nil*. If *pattern* specifies no captures, then the whole match is returned. A third, optional numerical argument *init* specifies where to start the search; its default value is 1 and may be negative.

string.rep (s, n)

Returns a string that is the concatenation of *n* copies of the string *s*.

string.reverse (s)

Returns a string that is the string *s* reversed.

string.sub (s, i [, j])

Returns the substring of *s* that starts at *i* and continues until *j*; *i* and *j* may be negative. If *j* is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call *string.sub(s, 1, j)* returns a prefix of *s* with length *j*, and *string.sub(s, -i)* returns a suffix of *s* with length *i*.

string.upper (s)

Receives a string and returns a copy of this string with all lowercase letters changed to uppercase. All other characters are left unchanged. The definition of what a lowercase letter is depends on the current locale.

Table Manipulation

This library provides generic functions for table manipulation. It provides all its functions inside the table *table*.

Most functions in the table library assume that the table represents an array or a list. For these functions, when we talk about the "length" of a table we mean the result of the length operator.

table.concat (table [, sep [, i [, j]])

Given an array where all elements are strings or numbers, returns *table[i]..sep..table[i+1] ...*

sep..table[j]. The default value for *sep* is the empty string, the default for *i* is 1, and the default for *j* is the length of the table. If *i* is greater than *j*, returns the empty string.

table.insert (table, [pos,] value)

Inserts element *value* at position *pos* in *table*, shifting up other elements to open space, if necessary. The default value for *pos* is *n+1*, where *n* is the length of the table, so that the call *table.insert(t,x)* inserts *x* at the end of table *t*.

table.maxn (table)

Returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices. To do its job this function does a linear traversal of the whole table, so it can be slow for large tables.

table.remove (table [, pos])

Removes from *table* the element at position *pos*, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for *pos* is *n*, where *n* is the length of the table, so that the call *table.remove(t)* removes the last element of table *t*.

table.sort (table [, comp])

Sorts *table* elements in a given order, in-place, from *table[1]* to *table[n]*, where *n* is the length of the table. If *comp* is given, then it must be a function that receives two table elements, and returns *true* when the first is less than the second (so that *not comp(a[i+1],a[i])* will be true after the sort). If *comp* is not given, then the standard Lua operator *<* is used instead.

The sort algorithm is not stable; that is, elements considered equal by the given order may have their relative positions changed by the sort.

Mathematical Functions

This library is an interface to the standard C math library. It provides all its functions inside the *table math*.

math.abs (x)

Returns the absolute value of *x*.

math.acos (x)

Returns the arc cosine of *x* (in radians).

math.asin (x)

Returns the arc sine of *x* (in radians).

math.atan (x)

Returns the arc tangent of x (in radians).

math.atan2 (x, y)

Returns the arc tangent of x/y (in radians), but uses the signs of both parameters to find the quadrant of the result. (It also handles correctly the case of y being zero.)

math.ceil (x)

Returns the smallest integer larger than or equal to x .

math.cos (x)

Returns the cosine of x (assumed to be in radians).

math.cosh (x)

Returns the hyperbolic cosine of x .

math.deg (x)

Returns the angle x (given in radians) in degrees.

math.exp (x)

Returns the value e^x .

math.floor (x)

Returns the largest integer smaller than or equal to x .

math.fmod (x, y)

Returns the remainder of the division of x by y .

math.frexp (x)

Returns m and e such that $x = m2^e$, e is an integer and the absolute value of m is in the range $[0.5, 1)$ (or zero when x is zero).

math.huge

The value `HUGE_VAL`, a value larger than or equal to any other numerical value.

math.ldexp (m, e)

Returns $m2^e$ (e should be an integer).

math.log (x)

Returns the natural logarithm of x .

math.log10 (x)

Returns the base-10 logarithm of x .

math.max (x, ...)

Returns the maximum value among its arguments.

math.min (x, ...)

Returns the minimum value among its arguments.

math.modf (x)

Returns two numbers, the integral part of x and the fractional part of x .

math.pi

The value of π .

math.pow (x, y)

Returns x to the power of y . (You can also use the expression x^y to compute this value.)

math.rad (x)

Returns the angle x (given in degrees) in radians.

math.random ([m [, n]])

This function is an interface to the simple pseudo-random generator function *rand* provided by ANSI C. (No guarantees can be given for its statistical properties.)

When called without arguments, returns a pseudo-random real number in the range $[0, 1)$. When called with a number m , *math.random* returns a pseudo-random integer in the range $[1, m]$. When called with two numbers m and n , *math.random* returns a pseudo-random integer in the range $[m, n]$.

math.randomseed (x)

Sets x as the "seed" for the pseudo-random generator: equal seeds produce equal sequences of numbers.

math.sin (x)

Returns the sine of x (assumed to be in radians).

math.sinh (x)

Returns the hyperbolic sine of x .

math.sqrt (x)

Returns the square root of x . (You can also use the expression $x^{0.5}$ to compute this value.)

math.tan (x)

Returns the tangent of x (assumed to be in radians).

math.tanh (x)

Returns the hyperbolic tangent of x .

Input and Output

The I/O library provides two different styles for file manipulation. The first one uses implicit file descriptors; that is, there are operations to set a default input file and a default output file, and all input/output operations are over these default files. The second style uses explicit file descriptors.

When using implicit file descriptors, all operations are supplied by table *io*. When using explicit file descriptors, the operation *io.open* returns a file descriptor and then all operations are supplied as methods of the file descriptor.

The table *io* also provides three predefined file descriptors with their usual meanings from C: *io.stdin*, *io.stdout*, and *io.stderr*.

Unless otherwise stated, all I/O functions return *nil* on failure (plus an error message as a second result and a system-dependent error code as a third result) and some value different from *nil* on success.

io.close ([file])

Equivalent to *file:close()*. Without a file, closes the default output file.

io.flush ()

Equivalent to *file:flush* over the default output file.

io.input ([file])

When called with a file name, it opens the named file (in text mode), and sets its handle as the default input file. When called with a file handle, it simply sets this file handle as the default input file. When called without parameters, it returns the current default input file.

In case of error this function raises the corresponding error instead of returning an error code.

io.lines ([filename])

Opens the given file name in read mode and returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in io.lines(filename) do
    ...
end
```

will iterate over all lines of the file. When the iterator function detects the end of file, it returns *nil* (to finish the loop) and automatically closes the file.

The call *io.lines()* (with no file name) is equivalent to *io.input():lines()*; that is, it iterates over the lines of the default input file. In this case it does not close the file when the loop ends.

io.open (filename [, mode])

This function opens a file, in the mode specified in the string mode. It returns a new file handle, or, in case of errors, *nil* plus an error message.

The mode string can be any of the following:

"r": read mode (the default);

"w": write mode;

"a": append mode;

"r+": update mode, all previous data is preserved;

"w+": update mode, all previous data is erased;

"a+": append update mode, previous data is preserved, writing is only allowed at the end of file.

The *mode* string may also have a 'b' at the end, which is needed in some systems to open the file in binary mode. This string is exactly what is used in the standard C function *fopen*.

io.output ([file])

Similar to *io.input*, but operates over the default output file.

io.popen (prog [, mode])

Starts program *prog* in a separated process and returns a file handle that you can use to read data from this program (if mode is "r", the default) or to write data to this program (if mode is "w").

This function is system dependent and is not available on all platforms.

io.read (...)

Equivalent to *io.input():read*.

io.tmpfile ()

Returns a handle for a temporary file. This file is opened in update mode and it is automatically removed when the program ends.

io.type (obj)

Checks whether *obj* is a valid file handle. Returns the string "file" if *obj* is an open file handle, "closed file" if *obj* is a closed file handle, or *nil* if *obj* is not a file handle.

io.write (...)

Equivalent to *io.output():write*.

file:close ()

Closes file. Note that files are automatically closed when their handles are garbage collected, but that takes an unpredictable amount of time to happen.

file:flush ()

Saves any un-written data to file.

file:lines ()

Returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in file:lines() do
  ...
end
```

will iterate over all lines of the file. (Unlike *io.lines*, this function does not close the file when the loop ends.)

file:read (...)

Reads the file *file*, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or *nil* if it cannot read data with the specified format. When called without formats, it uses a default format that reads the entire next line (see below).

The available formats are

"*n": reads a number; this is the only format that returns a number instead of a string.

"*a": reads the whole file, starting at the current position. On end of file, it returns the empty string.

"*l": reads the next line (skipping the end of line), returning *nil* on end of file. This is the default format.

number: reads a string with up to this number of characters, returning *nil* on end of file. If number is zero, it reads nothing and returns an empty string, or *nil* on end of file.

file:seek ([whence] [, offset])

Sets and gets the file position, measured from the beginning of the file, to the position given by offset plus a base specified by the string *whence*, as follows:

"set": base is position 0 (beginning of the file);

"cur": base is current position;

"end": base is end of file;

In case of success, function `seek` returns the final file position, measured in bytes from the beginning of the file. If this function fails, it returns `nil`, plus a string describing the error.

The default value for *whence* is "cur", and for *offset* is 0. Therefore, the call `file:seek()` returns the current file position, without changing it; the call `file:seek("set")` sets the position to the beginning of the file (and returns 0); and the call `file:seek("end")` sets the position to the end of the file, and returns its size.

file:write (...)

Writes the value of each of its arguments to the file. The arguments must be strings or numbers. To write other values, use *tostring* or *string.format* before *write*.

OS Facilities

This library is implemented through table `os`.

os.clock ()

Returns an approximation of the amount in seconds of CPU time used by the program.

os.date ([format [, time]])

Returns a string or a table containing date and time, formatted according to the given string format.

If the *time* argument is present, this is the time to be formatted (see the *os.time* function for a description of this value). Otherwise, *date* formats the current time.

If format starts with '!', then the date is formatted in Coordinated Universal Time. After this optional character, if format is the string `"*t"`, then *date* returns a table with the following fields: year (four digits), month (1--12), day (1--31), hour (0--23), min (0--59), sec (0--61), wday (weekday, Sunday is 1), yday (day of the year), and isdst (daylight saving flag, a boolean).

If format is not `"*t"`, then *date* returns the date as a string, formatted according to the same rules as the C function *strftime*.

When called without arguments, *date* returns a reasonable date and time representation that depends on the host system and on the current locale (that is, `os.date()` is equivalent to `os.date("%c")`).

os.difftime (t2, t1)

Returns the number of seconds from time *t1* to time *t2*. In POSIX, Windows, and some other systems, this value is exactly *t2-t1*.

os.execute ([command])

This function is equivalent to the C function *system*. It passes *command* to be executed by an operating system shell. It returns a status code, which is system-dependent. If *command* is absent, then it returns nonzero if a shell is available and zero otherwise.

os.exit ([code])

Calls the C function *exit*, with an optional *code*, to terminate the host program. The default value for *code* is the success code. As you may guess, using this function inside AtomWeaver is not only pointless but also dangerous as you may lose your work.

os.getenv (varname)

Returns the value of the process environment variable *varname*, or *nil* if the variable is not defined.

os.remove (filename)

Deletes the file or directory with the given name. Directories must be empty to be removed. If this function fails, it returns *nil*, plus a string describing the error.

os.rename (oldname, newname)

Renames file or directory named *oldname* to *newname*. If this function fails, it returns *nil*, plus a string describing the error.

os.time ([table])

Returns the current time when called without arguments, or a time representing the date and time specified by the given table. This table must have fields *year*, *month*, and *day*, and may have fields *hour*, *min*, *sec*, and *isdst* (for a description of these fields, see the *os.date* function).

The returned value is a number, whose meaning depends on your system. In POSIX, Windows, and some other systems, this number counts the number of seconds since some given start time (the "epoch"). In other systems, the meaning is not specified, and the number returned by *time* can be used only as an argument to *date* and *difftime*.

os.tmpname ()

Returns a string with a file name that can be used for a temporary file. The file must be explicitly opened before its use and explicitly removed when no longer needed.

Here is the complete syntax of Lua in extended BNF:

```

chunk ::= {stat [`;`]} [laststat [`;`]]

block ::= chunk

stat ::= varlist1 `=` explist1 |
        functioncall |
        do block end |
        while exp do block end |
        repeat block until exp |
        if exp then block {elseif exp then block} [else block] end |
        for Name `=` exp `,' exp [`,` exp] do block end |
        for namelist in explist1 do block end |
        function funcname funcbody |
        local function Name funcbody |
        local namelist [`= ` explist1]

laststat ::= return [explist1] | break

funcname ::= Name {`.` Name} [``: ` Name]

varlist1 ::= var {`,` var}

var ::= Name | prefixexp `[ ` exp `]` | prefixexp `.` Name

namelist ::= Name {`,` Name}

explist1 ::= {exp ``,`} exp

exp ::= nil | false | true | Number | String | `...` | function |
        prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | `( ` exp `)`

functioncall ::= prefixexp args | prefixexp ``: ` Name args

args ::= `( ` [explist1] `)` | tableconstructor | String

function ::= function funcbody

funcbody ::= `( ` [parlist1] `)` block end

parlist1 ::= namelist [`,` `...`] | `...`

tableconstructor ::= `{ ` [fieldlist] `}`

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= `[ ` exp `]` `=` exp | Name `=` exp | exp

fieldsep ::= ``,` | `;`

binop ::= `+` | `-` | `*` | `/` | `^` | `%` | `..` |
        `<` | `<=` | `>` | `>=` | `==` | `~=` |
        and | or

unop ::= `-` | not | `#`

```