1 5 - 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., `table.sort`).
All libraries are implemented through the official C API and are provided as separate C modules. Currently, Lua has the following standard libraries:

- basic library, which includes the coroutine sub-library;
- package library;
- string manipulation;
- table manipulation;
- mathematical functions (sin, log, etc.);
- input and output;
- operating system facilities;
- debug facilities.

Except for the basic and package libraries, each library provides all its functions as fields of a global table or as methods of its objects.
To have access to these libraries, the C host program should call the `luaL_openlibs` function, which opens all standard libraries. Alternatively, it can open them individually by calling `luaopen_base` (for the basic library), `luaopen_package` (for the package library), `luaopen_string` (for the string library), `luaopen_table` (for the table library), `luaopen_math` (for the mathematical library), `luaopen_io` (for the I/O library), `luaopen_os` (for the Operating System library), and `luaopen_debug` (for the debug library). These functions are declared in `lualib.h` and should not

be called directly: you must call them like any other Lua C function, e.g., by using `lua_call`.

1.1 5.1 - Basic Functions

The basic library provides some core functions to Lua. If you do not include this library in your application, you should check carefully whether you need to provide implementations for some of its facilities.

1.1.1 `assert (v [, message])`

Issues an error when the value of its argument `v` is false (i.e., **nil** or **false**); otherwise, returns all its arguments. `message` is an error message; when absent, it defaults to "assertion failed!"

1.1.2 `collectgarbage (opt [, arg])`

This function is a generic interface to the garbage collector. It performs different functions according to its first argument, `opt`:

- **"stop":** stops the garbage collector.
- **"restart":** restarts the garbage collector.
- **"collect":** performs a full garbage-collection cycle.
- **"count":** returns the total memory in use by Lua (in Kbytes).
- **"step":** performs a garbage-collection step. The step "size" is controlled by `arg` (larger values mean more steps) in a non-specified way. If you want to control the step size you must experimentally tune the value of `arg`. Returns **true** if the step finished a collection cycle.
- **"setpause":** sets `arg` as the new value for the *pause* of the collector (see §2.10). Returns the previous value for *pause*.
- **"setstepmul":** sets `arg` as the new value for the *step multiplier* of the collector (see §2.10). Returns the previous value for step.

1.1.3 `dofile (filename)`

Opens the named file and executes its contents as a Lua chunk. When called without arguments, `dofile` executes the contents of the standard input (`stdin`). Returns

all values returned by the chunk. In case of errors, `dofile` propagates the error to its caller (that is, `dofile` does not run in protected mode).

1.1.4 `error (message [, level])`

Terminates the last protected function called and returns `message` as the error message. Function `error` never returns.
Usually, `error` adds some information about the error position at the beginning of the message. The `level` argument specifies how to get the error position. With level 1 (the default), the error position is where the `error` function was called. Level 2 points the error to where the function that called `error` was called; and so on. Passing a level 0 avoids the addition of error position information to the message.

1.1.5 `_G`

A global variable (not a function) that holds the global environment (that is, `_G._G = _G`). Lua itself does not use this variable; changing its value does not affect any environment, nor vice-versa. (Use `setfenv` to change environments.)

1.1.6 `getfenv ([f])`

Returns the current environment in use by the function. `f` can be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling `getfenv`. If the given function is not a Lua function, or if `f` is 0, `getfenv` returns the global environment. The default for `f` is 1.

1.1.7 `getmetatable (object)`

If `object` does not have a metatable, returns **nil**. Otherwise, if the object's metatable has a `"__metatable"` field, returns the associated value. Otherwise, returns the metatable of the given object.

1.1.8 `ipairs (t)`

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

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

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

1.1.9 `load (func [, chunkname])`

Loads a chunk using function `func` to get its pieces. Each call to `func` must return a string that concatenates with previous results. A return of an empty string, **nil**, or no value signals the end of the chunk.
If there are no errors, returns the compiled chunk as a function; otherwise, returns **nil** plus the error message. The environment of the returned function is the global environment.
`chunkname` is used as the chunk name for error messages and debug information. When absent, it defaults to `"=(load)"`.

1.1.10 `loadfile ([filename])`

Similar to `load`, but gets the chunk from file `filename` or from the standard input, if no file name is given.

1.1.11 `loadstring (string [, chunkname])`

Similar to `load`, but gets the chunk from the given string.
To load and run a given string, use the idiom

```
assert(loadstring(s))()
```

When absent, `chunkname` defaults to the given string.

1.1.12 `next (table [, index])`

Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. `next` returns the next index of the table and its associated value. When called with **nil** as its second argument, `next` returns an initial index and its associated value. When called with the last index, or with **nil** in an empty table, `next` returns **nil**. If the second argument is absent, then it is interpreted as **nil**. In particular, you can use `next(t)` to check whether a table is empty.
The order in which the indices are enumerated is not specified, *even for numeric indices*. (To traverse a table in numeric order, use a numerical **for** or the `ipairs` function.)

The behavior of `next` is *undefined* if, during the traversal, you assign any value to a non-existent field in the table. You may however modify existing fields. In particular, you may clear existing fields.

1.1.13 `pairs (t)`

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

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

will iterate over all keyůvalue pairs of table `t`.
See function `next` for the caveats of modifying the table during its traversal.

1.1.14 `pcall (f, arg1, )`

Calls function `f` with the given arguments in *protected mode*. This means that any error inside `f` is not propagated; instead, `pcall` catches the error and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, `pcall` also returns all results from the call, after this first result. In case of any error, `pcall` returns **false** plus the error message.

1.1.15 `print ()`

Receives any number of arguments, and prints their values to `stdout`, using the `tostring` function to convert them to strings. `print` is not intended for formatted

output, but only as a quick way to show a value, typically for debugging. For formatted output, use `string.format`.

1.1.16 `rawequal (v1, v2)`

Checks whether `v1` is equal to `v2`, without invoking any metamethod. Returns a boolean.

1.1.17 `rawget (table, index)`

Gets the real value of `table[index]`, without invoking any metamethod. `table` must be a table; `index` may be any value.

1.1.18 `rawset (table, index, value)`

Sets the real value of `table[index]` to `value`, without invoking any metamethod. `table` must be a table, `index` any value different from **nil**, and `value` any Lua value. This function returns `table`.

1.1.19 `select (index, )`

If `index` is a number, returns all arguments after argument number `index`. Otherwise, `index` must be the string `"#"`, and `select` returns the total number of extra arguments it received.

1.1.20 `setfenv (f, table)`

Sets the environment to be used by the given function. `f` can be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling `setfenv`. `setfenv` returns the given function.
As a special case, when `f` is 0 `setfenv` changes the environment of the running thread. In this case, `setfenv` returns no values.

1.1.21 `setmetatable (table, metatable)`

Sets the metatable for the given table. (You cannot change the metatable of other types from Lua, only from C.) If `metatable` is **nil**, removes the metatable of the given table. If the original metatable has a `"__metatable"` field, raises an error.

This function returns `table`.

1.1.22 `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 can have a decimal part, as well as an optional exponent part (see §2.1). In other bases, only unsigned integers are accepted.

1.1.23 `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.

1.1.24 `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"`.

1.1.25 `unpack (list [, i [, j]])`

Returns the elements from the given table. This function is equivalent to

```
return list[i], list[i+1], , list[j]
```

except that the above code can be written only for a fixed number of elements. By default, `i` is 1 and `j` is the length of the list, as defined by the length operator (see §2.5.5).

1.1.26 _VERSION

A global variable (not a function) that holds a string containing the current interpreter version. The current contents of this variable is "`Lua 5.1`".

1.1.27 xpcall (f, err)

This function is similar to `pcall`, except that you can set a new error handler.
`xpcall` calls function `f` in protected mode, using `err` as the error handler. Any error inside `f` is not propagated; instead, `xpcall` catches the error, calls the `err` function with the original error object, and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In this case, `xpcall` also returns all results from the call, after this first result. In case of any error, `xpcall` returns **false** plus the result from `err`.

1.2 5.2 - Coroutine Manipulation

The operations related to coroutines comprise a sub-library of the basic library and come inside the table `coroutine` . See §2.11 for a general description of coroutines.

1.2.1 coroutine.create (f)

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns this new coroutine, an object with type "`thread`".

1.2.2 coroutine.resume (co [, val1,])

Starts or continues the execution of coroutine `co`. The first time you resume a coroutine, it starts running its body. The values `val1`, are passed as the arguments to the body function. If the coroutine has yielded, `resume` restarts it; the values `val1`, are passed as the results from the yield.
If the coroutine runs without any errors, `resume` returns **true** plus any values passed to `yield` (if the coroutine yields) or any values returned by the body function (if

the coroutine terminates). If there is any error, `resume` returns **false** plus the error message.

1.2.3 `coroutine.running ()`

Returns the running coroutine, or **nil** when called by the main thread.

1.2.4 `coroutine.status (co)`

Returns the status of coroutine `co`, as a string: `"running"`, if the coroutine is running (that is, it called `status`); `"suspended"`, if the coroutine is suspended in a call to `yield`, or if it has not started running yet; `"normal"` if the coroutine is active but not running (that is, it has resumed another coroutine); and `"dead"` if the coroutine has finished its body function, or if it has stopped with an error.

1.2.5 `coroutine.wrap (f)`

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns a function that resumes the coroutine each time it is called. Any arguments passed to the function behave as the extra arguments to `resume`. Returns the same values returned by `resume`, except the first boolean. In case of error, propagates the error.

1.2.6 `coroutine.yield ()`

Suspends the execution of the calling coroutine. The coroutine cannot be running a C function, a metamethod, or an iterator. Any arguments to `yield` are passed as extra results to `resume`.

1.3 5.3 - Modules

The package library provides basic facilities for loading and building modules in Lua. It exports two of its functions directly in the global environment: `require` and `module`. Everything else is exported in a table `package` .

1.3.1 `module (name [, ])`

Creates a module. If there is a table in `package.loaded[name]`, this table is the module. Otherwise, if there is a global table `t` with the given name, this table is the

module. Otherwise creates a new table `t` and sets it as the value of the global `name` and the value of `package.loaded[name]`. This function also initializes `t._NAME` with the given name, `t._M` with the module (`t` itself), and `t._PACKAGE` with the package name (the full module name minus last component; see below). Finally, `module` sets `t` as the new environment of the current function and the new value of `package.loaded[name]`, so that `require` returns `t`.
If `name` is a compound name (that is, one with components separated by dots), `module` creates (or reuses, if they already exist) tables for each component. For instance, if `name` is `a.b.c`, then `module` stores the module table in field `c` of field `b` of global `a`.
This function can receive optional *options* after the module name, where each option is a function to be applied over the module.

1.3.2 `require (modname)`

Loads the given module. The function starts by looking into the `package.loaded` table to determine whether `modname` is already loaded. If it is, then `require` returns the value stored at `package.loaded[modname]`. Otherwise, it tries to find a *loader* for the module.
To find a loader, `require` is guided by the `package.loaders` array. By changing this array, we can change how `require` looks for a module. The following explanation is based on the default configuration for `package.loaders`.
First `require` queries `package.preload[modname]`. If it has a value, this value (which should be a function) is the loader. Otherwise `require` searches for a Lua loader using the path stored in `package.path`. If that also fails, it searches for a C loader using the path stored in `package.cpath`. If that also fails, it tries an *all-in-one* loader (see `package.loaders`).
Once a loader is found, `require` calls the loader with a single argument, `modname`. If the loader returns any value, `require` assigns the returned value to `package.loaded[modname]`. If the loader returns no value and has not assigned any value to `package.loaded[modname]`, then `require` assigns **true** to this entry. In any case, `require` returns the final value of `package.loaded[modname]`.
If there is any error loading or running the module, or if it cannot find any loader for the module, then `require` signals an error.

1.3.3 `package.cpath`

The path used by `require` to search for a C loader.

Lua initializes the C path `package.cpath` in the same way it initializes the Lua path `package.path`, using the environment variable `LUA_CPATH` or a default path defined in `luaconf.h`.

1.3.4 `package.loaded`

A table used by `require` to control which modules are already loaded. When you require a module `modname` and `package.loaded[modname]` is not false, `require` simply returns the value stored there.

1.3.5 `package.loaders`

A table used by `require` to control how to load modules.
Each entry in this table is a *searcher function*. When looking for a module, `require` calls each of these searchers in ascending order, with the module name (the argument given to `require`) as its sole parameter. The function can return another function (the module *loader*) or a string explaining why it did not find that module (or **nil** if it has nothing to say). Lua initializes this table with four functions.
The first searcher simply looks for a loader in the `package.preload` table.
The second searcher looks for a loader as a Lua library, using the path stored at `package.path`. A path is a sequence of *templates* separated by semicolons. For each template, the searcher will change each interrogation mark in the template by `filename`, which is the module name with each dot replaced by a "directory separator" (such as `"/"` in Unix); then it will try to open the resulting file name. So, for instance, if the Lua path is the string

```
"./?.lua;./?.lc;/usr/local/?/init.lua"
```

the search for a Lua file for module `foo` will try to open the files `./foo.lua`, `./foo.lc`, and `/usr/local/foo/init.lua`, in that order.
The third searcher looks for a loader as a C library, using the path given by the variable `package.cpath`. For instance, if the C path is the string

```
"./?.so;./?.dll;/usr/local/?/init.so"
```

the searcher for module `foo` will try to open the files `./foo.so`, `./foo.dll`, and `/usr/local/foo/init.so`, in that order. Once it finds a C library, this searcher first uses a dynamic link facility to link the application with the library. Then it tries to find a C function inside the library to be used as the loader. The name of this C function is the string `"luaopen_"` concatenated with a copy of the module name where each dot is replaced by an underscore. Moreover, if the module name has a

hyphen, its prefix up to (and including) the first hyphen is removed. For instance, if the module name is `a.v1-b.c`, the function name will be `luaopen_b_c`.
The fourth searcher tries an *all-in-one loader*. It searches the C path for a library for the root name of the given module. For instance, when requiring `a.b.c`, it will search for a C library for `a`. If found, it looks into it for an open function for the submodule; in our example, that would be `luaopen_a_b_c`. With this facility, a package can pack several C submodules into one single library, with each submodule keeping its original open function.

1.3.6 `package.loadlib (libname, funcname)`

Dynamically links the host program with the C library `libname`. Inside this library, looks for a function `funcname` and returns this function as a C function. (So, `funcname` must follow the protocol (see `lua_CFunction`)).
This is a low-level function. It completely bypasses the package and module system. Unlike `require`, it does not perform any path searching and does not automatically adds extensions. `libname` must be the complete file name of the C library, including if necessary a path and extension. `funcname` must be the exact name exported by the C library (which may depend on the C compiler and linker used).
This function is not supported by ANSI C. As such, it is only available on some platforms (Windows, Linux, Mac OS X, Solaris, BSD, plus other Unix systems that support the `dlfcn` standard).

1.3.7 `package.path`

The path used by `require` to search for a Lua loader.
At start-up, Lua initializes this variable with the value of the environment variable `LUA_PATH` or with a default path defined in `luaconf.h`, if the environment variable

is not defined. Any `";;"` in the value of the environment variable is replaced by the default path.

1.3.8 `package.preload`

A table to store loaders for specific modules (see `require`).

1.3.9 `package.seeall (module)`

Sets a metatable for `module` with its `__index` field referring to the global environment, so that this module inherits values from the global environment. To be used as an option to function `module`.

1.4 5.4 - 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)`.
The string library assumes one-byte character encodings.

1.4.1 `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`.
Note that numerical codes are not necessarily portable across platforms.

1.4.2 `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.

Note that numerical codes are not necessarily portable across platforms.

1.4.3 `string.dump (function)`

Returns a string containing a binary representation of the given function, so that a later `loadstring` on this string returns a copy of the function. `function` must be a Lua function without upvalues.

1.4.4 `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 can 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.

1.4.5 `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, newlines, embedded zeros, and backslashes in the string are correctly escaped when written. For instance, the call

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

will produce the string:

```
"a string with \"quotes\" and \
      new line"
```

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.

1.4.6 `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. The next example collects all pairs `key=value` from the given string into a table:

```
t = {}
     s = "from=world, to=Lua"
     for k, v in string.gmatch(s, "(%w+)=(%w+)") do
       t[k] = v
     end
```

For this function, a '`^`' at the start of a pattern does not work as an anchor, as this would prevent the iteration.

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

Returns a copy of `s` in which all (or the first `n`, if given) occurrences of the `pattern` have been replaced by a replacement string specified by `repl`, which can be a string, a table, or a function. `gsub` also returns, as its second value, the total number of matches that occurred.
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).
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"

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

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

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

     local t = {name="lua", version="5.1"}
     x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
     --> x="lua-5.1.tar.gz"
```

1.4.8 `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.

1.4.9 `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.

1.4.10 `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 can be negative.

1.4.11 `string.rep (s, n)`

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

1.4.12 `string.reverse (s)`

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

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

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` can 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`.

1.4.14 `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.

1.4.15 5.4.1 - Patterns

1.4.15.1 Character Class:

A *character class* is used to represent a set of characters. The following combinations are allowed in describing a character class:

- **x:** (where *x* is not one of the *magic characters* `^$()%.[]*+-?`) represents the character *x* itself.
- `.`: (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*. This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a '`%`' when used to represent itself in a pattern.
- `[set]`: represents the class which is the union of all characters in *set*. A range of characters can be specified by separating the end characters of the range with a '`-`'. All classes `%`*x* described above can also be used as components in *set*. All other characters in *set* represent themselves. For example, `[%w_]` (or `[_%w]`) represents all alphanumeric characters plus the underscore, `[0-7]` represents the octal digits, and `[0-7%l%-]` represents the octal digits plus the lowercase letters plus the '`-`' character.
  The interaction between ranges and classes is not defined. Therefore, patterns like `[%a-z]` or `[a-%%]` have no meaning.
- `[^set]`: represents the complement of *set*, where *set* is interpreted as above.

For all classes represented by single letters (`%a`, `%c`, etc.), the corresponding uppercase letter represents the complement of the class. For instance, `%S` represents all non-space characters.

The definitions of letter, space, and other character groups depend on the current locale. In particular, the class `[a-z]` may not be equivalent to `%l`.

1.4.15.2 Pattern Item:

A *pattern item* can be

- a single character class, which matches any single character in the class;
- a single character class followed by '`*`', which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '`+`', which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '`-`', which also matches 0 or more repetitions of characters in the class. Unlike '`*`', these repetition items will always match the *shortest* possible sequence;
- a single character class followed by '`?`', which matches 0 or 1 occurrence of a character in the class;
- `%n`, for *n* between 1 and 9; such item matches a substring equal to the *n*-th captured string (see below);
- `%bxy`, where *x* and *y* are two distinct characters; such item matches strings that start with *x*, end with *y*, and where the *x* and *y* are *balanced*. This means that, if one reads the string from left to right, counting *+1* for an *x* and *-1* for a *y*, the ending *y* is the first *y* where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.

1.4.15.3 Pattern:

A *pattern* is a sequence of pattern items. A '`^`' at the beginning of a pattern anchors the match at the beginning of the subject string. A '`$`' at the end of a pattern anchors

the match at the end of the subject string. At other positions, '^' and '$' have no special meaning and represent themselves.

1.4.15.4 Captures:

A pattern can contain sub-patterns enclosed in parentheses; they describe *captures*. When a match succeeds, the substrings of the subject string that match captures are stored (*captured*) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern `"(a*(.)%w(%s*))"`, the part of the string matching `"a*(.)%w(%s*)"` is stored as the first capture (and therefore has number 1); the character matching `"."` is captured with number 2, and the part matching "`%s*`" has number 3.
As a special case, the empty capture `()` captures the current string position (a number). For instance, if we apply the pattern `"()aa()"` on the string `"flaaap"`, there will be two captures: 3 and 5.
A pattern cannot contain embedded zeros. Use `%z` instead.

1.5 5.5 - 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.

1.5.1 `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.

1.5.2 `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 (see §2.5.5), so that a call `table.insert(t,x)` inserts `x` at the end of table `t`.

1.5.3 `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.)

1.5.4 `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 a call `table.remove(t)` removes the last element of table `t`.

1.5.5 `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.

1.6 5.6 - Mathematical Functions

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

1.6.1 `math.abs (x)`

Returns the absolute value of `x`.

1.6.2 `math.acos (x)`

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

1.6.3 `math.asin (x)`

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

1.6.4 `math.atan (x)`

Returns the arc tangent of `x` (in radians).

1.6.5 `math.atan2 (y, x)`

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

zero.)

1.6.6 `math.ceil (x)`

Returns the smallest integer larger than or equal to `x`.

1.6.7 `math.cos (x)`

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

1.6.8 `math.cosh (x)`

Returns the hyperbolic cosine of `x`.

1.6.9 `math.deg (x)`

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

1.6.10 `math.exp (x)`

Returns the value e^x.

1.6.11 `math.floor (x)`

Returns the largest integer smaller than or equal to `x`.

1.6.12 `math.fmod (x, y)`

Returns the remainder of the division of `x` by `y` that rounds the quotient towards zero.

1.6.13 `math.frexp (x)`

Returns `m` and `e` such that $x = \mathrm{m}2^e$, `e` is an integer and the absolute value of `m` is in

the range *[0.5, 1)* (or zero when `x` is zero).

1.6.14 `math.huge`

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

1.6.15 `math.ldexp (m, e)`

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

1.6.16 `math.log (x)`

Returns the natural logarithm of `x`.

1.6.17 `math.log10 (x)`

Returns the base-10 logarithm of `x`.

1.6.18 `math.max (x, )`

Returns the maximum value among its arguments.

1.6.19 `math.min (x, )`

Returns the minimum value among its arguments.

1.6.20 `math.modf (x)`

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

1.6.21 `math.pi`

The value of *pi*.

1.6.22 `math.pow (x, y)`

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

1.6.23 `math.rad (x)`

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

1.6.24 `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 uniform pseudo-random real number in the range *[0,1)*. When called with an integer number `m`, `math.random` returns a uniform pseudo-random integer in the range *[1, m]*. When called with two integer numbers `m` and `n`, `math.random` returns a uniform pseudo-random integer in the range *[m, n]*.

1.6.25 `math.randomseed (x)`

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

1.6.26 `math.sin (x)`

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

1.6.27 `math.sinh (x)`

Returns the hyperbolic sine of `x`.

1.6.28 `math.sqrt (x)`

Returns the square root of `x`. (You can also use the expression `x^0.5` to compute

this value.)

1.6.29 `math.tan (x)`

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

1.6.30 `math.tanh (x)`

Returns the hyperbolic tangent of `x`.

1.7 5.7 - Input and Output Facilities

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` . The I/O library never closes these files.
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.

1.7.1 `io.close ([file])`

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

1.7.2 `io.flush ()`

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

1.7.3 `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 errors this function raises the error, instead of returning an error code.

1.7.4 `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 body 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.

1.7.5 `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 can 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`.

1.7.6 `io.output ([file])`

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

1.7.7 `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.

1.7.8 `io.read ()`

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

1.7.9 `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.

1.7.10 `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.

1.7.11 `io.write ()`

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

1.7.12 `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.

1.7.13 `file:flush ()`

Saves any written data to `file`.

1.7.14 `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 body end
```

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

1.7.15 `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.

1.7.16 `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.

1.7.17 `file:setvbuf (mode [, size])`

Sets the buffering mode for an output file. There are three available modes:

- **"no":** no buffering; the result of any output operation appears immediately.
- **"full":** full buffering; output operation is performed only when the buffer is full (or when you explicitly `flush` the file (see `io.flush`)).
- **"line":** line buffering; output is buffered until a newline is output or there is any input from some special files (such as a terminal device).

For the last two cases, `size` specifies the size of the buffer, in bytes. The default is an appropriate size.

1.7.18 `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`.

1.8 5.8 - Operating System Facilities

This library is implemented through table `os` .

1.8.1 `os.clock ()`

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

1.8.2 `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")`).

1.8.3 `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`.

1.8.4 `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.

1.8.5 `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.

1.8.6 `os.getenv (varname)`

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

1.8.7 `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.

1.8.8 `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.

1.8.9 `os.setlocale (locale [, category])`

Sets the current locale of the program. `locale` is a string specifying a locale; `category` is an optional string describing which category to change: `"all"`, `"collate"`,

"ctype", "monetary", "numeric", or "time"; the default category is "all". The function returns the name of the new locale, or **nil** if the request cannot be honored.
If locale is the empty string, the current locale is set to an implementation-defined native locale. If locale is the string "C", the current locale is set to the standard C locale.
When called with **nil** as the first argument, this function only returns the name of the current locale for the given category.

1.8.10 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.

1.8.11 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.
On some systems (POSIX), this function also creates a file with that name, to avoid security risks. (Someone else might create the file with wrong permissions in the time between getting the name and creating the file.) You still have to open the file to use it and to remove it (even if you do not use it).
When possible, you may prefer to use io.tmpfile, which automatically removes the file when the program ends.

1.9 5.9 - The Debug Library

This library provides the functionality of the debug interface to Lua programs. You should exert care when using this library. The functions provided here should be used exclusively for debugging and similar tasks, such as profiling. Please resist the temptation to use them as a usual programming tool: they can be very slow. Moreover, several of these functions violate some assumptions about Lua code (e.g., that variables local to a function cannot be accessed from outside or that userdata

metatables cannot be changed by Lua code) and therefore can compromise otherwise secure code.
All functions in this library are provided inside the `debug` table. All functions that operate over a thread have an optional first argument which is the thread to operate over. The default is always the current thread.

1.9.1 `debug.debug ()`

Enters an interactive mode with the user, running each string that the user enters. Using simple commands and other debug facilities, the user can inspect global and local variables, change their values, evaluate expressions, and so on. A line containing only the word `cont` finishes this function, so that the caller continues its execution.
Note that commands for `debug.debug` are not lexically nested within any function, and so have no direct access to local variables.

1.9.2 `debug.getfenv (o)`

Returns the environment of object `o`.

1.9.3 `debug.gethook ([thread])`

Returns the current hook settings of the thread, as three values: the current hook function, the current hook mask, and the current hook count (as set by the `debug.sethook` function).

1.9.4 `debug.getinfo ([thread,] function [, what])`

Returns a table with information about a function. You can give the function directly, or you can give a number as the value of `function`, which means the function running at level `function` of the call stack of the given thread: level 0 is the current function (`getinfo` itself); level 1 is the function that called `getinfo`; and so on. If `function` is a number larger than the number of active functions, then `getinfo` returns **nil**.
The returned table can contain all the fields returned by `lua_getinfo`, with the string `what` describing which fields to fill in. The default for `what` is to get all information available, except the table of valid lines. If present, the option '`f`' adds a field named `func` with the function itself. If present, the option '`L`' adds a field named `activelines` with the table of valid lines.

For instance, the expression `debug.getinfo(1,"n").name` returns a table with a name for the current function, if a reasonable name can be found, and the expression `debug.getinfo(print)` returns a table with all available information about the `print` function.

1.9.5 `debug.getlocal ([thread,] level, local)`

This function returns the name and the value of the local variable with index `local` of the function at level `level` of the stack. (The first parameter or local variable has index 1, and so on, until the last active local variable.) The function returns **nil** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `debug.getinfo` to check whether the level is valid.)
Variable names starting with '`(`' (open parentheses) represent internal variables (loop control variables, temporaries, and C function locals).

1.9.6 `debug.getmetatable (object)`

Returns the metatable of the given `object` or **nil** if it does not have a metatable.

1.9.7 `debug.getregistry ()`

Returns the registry table (see §3.5).

1.9.8 `debug.getupvalue (func, up)`

This function returns the name and the value of the upvalue with index `up` of the function `func`. The function returns **nil** if there is no upvalue with the given index.

1.9.9 `debug.setfenv (object, table)`

Sets the environment of the given `object` to the given `table`. Returns `object`.

1.9.10 `debug.sethook ([thread,] hook, mask [, count])`

Sets the given function as a hook. The string `mask` and the number `count` describe when the hook will be called. The string mask may have the following characters, with the given meaning:

- `"c"`: the hook is called every time Lua calls a function;
- `"r"`: the hook is called every time Lua returns from a function;
- `"l"`: the hook is called every time Lua enters a new line of code.

With a `count` different from zero, the hook is called after every `count` instructions. When called without arguments, `debug.sethook` turns off the hook.
When the hook is called, its first parameter is a string describing the event that has triggered its call: `"call"`, `"return"` (or `"tail return"`, when simulating a return from a tail call), `"line"`, and `"count"`. For line events, the hook also gets the new line number as its second parameter. Inside a hook, you can call `getinfo` with level 2 to get more information about the running function (level 0 is the `getinfo` function, and level 1 is the hook function), unless the event is `"tail return"`. In this case, Lua is only simulating the return, and a call to `getinfo` will return invalid data.

1.9.11 `debug.setlocal ([thread,] level, local, value)`

This function assigns the value `value` to the local variable with index `local` of the function at level `level` of the stack. The function returns **nil** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `getinfo` to check whether the level is valid.) Otherwise, it returns the name of the local variable.

1.9.12 `debug.setmetatable (object, table)`

Sets the metatable for the given `object` to the given `table` (which can be **nil**).

1.9.13 `debug.setupvalue (func, up, value)`

This function assigns the value `value` to the upvalue with index `up` of the function `func`. The function returns **nil** if there is no upvalue with the given index. Otherwise, it returns the name of the upvalue.

1.9.14 `debug.traceback ([thread,] [message] [, level])`

Returns a string with a traceback of the call stack. An optional `message` string is appended at the beginning of the traceback. An optional `level` number tells at which level to start the traceback (default is 1, the function calling `traceback`).