## 1 2 - The Language

This section describes the lexis, the syntax, and the semantics of Lua. In other words, this section describes which tokens are valid, how they can be combined, and what their combinations mean.
The language constructs will be explained using the usual extended BNF notation, in which *a* means 0 or more *a*'s, and [*a*] means an optional *a*. Non-terminals are shown like non-terminal, keywords are shown like **kword**, and other terminal symbols are shown like `= . The complete syntax of Lua can be found in §8 at the end of this manual.

### 1.1 2.1 - Lexical Conventions

*Names* (also called *identifiers*) in Lua can be any string of letters, digits, and underscores, not beginning with a digit. This coincides with the definition of names in most languages. (The definition of letter depends on the current locale: any character considered alphabetic by the current locale can be used in an identifier.) Identifiers are used to name variables and table fields.
The following *keywords* are reserved 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
```

Lua is a case-sensitive language: `and` is a reserved word, but `And` and `AND` are two different, valid names. As a convention, names starting with an underscore followed by uppercase letters (such as `_VERSION`) are reserved for internal global variables used by Lua.
The following strings denote other tokens:

```
     +     -     *     /     %     ^     #
     ==    ~=    <=    >=    <     >     =
     (     )     {     }     [     ]
     ;     :     ,     .     ..    ...
```

*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`' (newline), '`\r`' (carriage return), '`\t`' (horizontal tab), '`\v`' (vertical tab), '`\\`' (backslash), '`\"`' (quotation mark [double quote]), and '`\'`' (apostrophe [single quote]). Moreover, a backslash followed by a real newline results in a newline in the string. A character in a string can also be specified by its numerical value using

the escape sequence `\ddd`, where *ddd* is a sequence of up to three decimal digits. (Note that if a numerical escape is to be followed by a digit, it must be expressed using exactly three digits.) Strings in Lua can contain any 8-bit value, including embedded zeros, which can be specified as '`\0`'.
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 can run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. They can contain anything except a closing bracket of the proper level.
For convenience, when the opening long bracket is immediately followed by a newline, the newline is not included in the string. As an example, in a system using ASCII (in which '`a`' is coded as 97, newline is coded as 10, and '`1`' is coded as 49), the five literal strings below denote the same string:

```
a = 'alo\n123"'
     a = "alo\n123\""
     a = '\97lo\10\04923"'
     a = [[alo
     123"]]
     a = [==[
     alo
     123"]==]
```

A *numerical constant* can be written with an optional decimal part and an optional decimal exponent. Lua also accepts integer hexadecimal constants, by prefixing them with `0x`. Examples of valid numerical constants are

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

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.

## 1.2 2.2 - Values and 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.
All values in Lua are *first-class values*. This means that all values 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*. *Nil* is the type of the value **nil**, whose main property is to be different from any other value; it usually represents the absence of a useful value. *Boolean* is the type of the values **false** and **true**. Both **nil** and **false** make a condition false; any other value makes it true. *Number* represents real (double-precision floating-point) numbers. (It is easy to build Lua interpreters that use other internal representations for numbers, such as single-precision float or long integers; see file `luaconf.h`.) *String* represents arrays of characters.
Lua is 8-bit clean: strings can contain any 8-bit character, including embedded zeros ('`\0`') (see `§2.1`).
Lua can call (and manipulate) functions written in Lua and functions written in C (see `§2.5.8`).
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. However, by using *metatables*, the programmer can define operations for userdata values (see `§2.8`). Userdata values cannot be created or modified in Lua, only through the C API. This guarantees the integrity of data owned by the host program.
The type *thread* represents independent threads of execution and it is used to implement coroutines (see `§2.11`). Do not confuse Lua threads with operating-system threads. Lua supports coroutines on all systems, even those that do not support threads.
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 sole data structuring mechanism in Lua; they can 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"]`. There are several convenient ways to create tables in Lua (see `§2.5.7`).

Like indices, the value of a table field can be of any type (except **nil**). In particular, because functions are first-class values, table fields can contain functions. Thus tables can also carry *methods* (see §2.5.9).

Tables, functions, threads, and (full) userdata values are *objects*: variables do not actually *contain* these values, only *references* to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.

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

### 1.2.1 2.2.1 - Coercion

Lua provides automatic conversion between string and number values at run time. Any arithmetic operation applied to a string tries to convert this string to a number, following the usual conversion rules. Conversely, whenever a number is used where a string is expected, the number is converted to a string, in a reasonable format. For complete control over how numbers are converted to strings, use the `format` function from the string library (see `string.format`).

## 1.3 2.3 - Variables

Variables are places that store values.

There are three kinds of variables in Lua: global variables, local variables, and table fields.

A single name can denote a global variable or a local variable (or a function's formal parameter, which is a particular kind of local variable):

```
var ::= Name
```

Name denotes identifiers, as defined in §2.1.

Any variable is assumed to be global unless explicitly declared as a local (see §2.4.7). Local variables are *lexically scoped*: local variables can be freely accessed by functions defined inside their scope (see §2.6).

Before the first assignment to a variable, its value is **nil**.

Square brackets are used to index a table:

```
var ::= prefixexp `{\bf [}  exp `{\bf ]}
```

The meaning of accesses to global variables and table fields can be changed via metatables. An access to an indexed variable `t[i]` is equivalent to a call `gettable_event(t,i)`. (See §2.8 for a complete description of the `gettable_event`

function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)
The syntax `var.Name` is just syntactic sugar for `var["Name"]`:

```
var ::= prefixexp `{\bf .}  Name
```

All global variables live as fields in ordinary Lua tables, called *environment tables* or simply *environments* (see §2.9). Each function has its own reference to an environment, so that all global variables in this function will refer to this environment table. When a function is created, it inherits the environment from the function that created it. To get the environment table of a Lua function, you call `getfenv`. To replace it, you call `setfenv`. (You can only manipulate the environment of C functions through the debug library; (see §5.9).)
An access to a global variable `x` is equivalent to `_env.x`, which in turn is equivalent to

```
gettable_event(_env, "x")
```

where `_env` is the environment of the running function. (See §2.8 for a complete description of the `gettable_event` function. This function is not defined or callable in Lua. Similarly, the `_env` variable is not defined in Lua. We use them here only for explanatory purposes.)

## 1.4 2.4 - Statements

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

### 1.4.1 2.4.1 - Chunks

The unit of execution of Lua is called a *chunk*. A chunk is simply a sequence of statements, which are executed sequentially. Each statement can be optionally followed by a semicolon:

```
chunk ::= {stat [`{\bf ;} ]}
```

There are no empty statements and thus '`;;`' is not legal.
Lua handles a chunk as the body of an anonymous function with a variable number of arguments (see §2.5.9). As such, chunks can define local variables, receive arguments, and return values.

A chunk can be stored in a file or in a string inside the host program. To execute a chunk, Lua first pre-compiles the chunk into instructions for a virtual machine, and then it executes the compiled code with an interpreter for the virtual machine.
Chunks can also be pre-compiled into binary form; see program `luac` for details. Programs in source and compiled forms are interchangeable; Lua automatically detects the file type and acts accordingly.

### 1.4.2 2.4.2 - Blocks

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

```
block ::= chunk
```

A block can be explicitly delimited to produce a single statement:

```
stat ::= {\bf do} block {\bf end}
```

Explicit blocks are useful to control the scope of variable declarations. Explicit blocks are also sometimes used to add a **return** or **break** statement in the middle of another block (see §2.4.4).

### 1.4.3 2.4.3 - Assignment

Lua allows multiple assignments. 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:

```
stat ::= varlist `{\bf =}  explist
       varlist ::= var {`{\bf ,}  var}
       explist ::= exp {`{\bf ,}  exp}
```

Expressions are discussed in §2.5.
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 that call enter the list of values, before the adjustment (except when the call is enclosed in parentheses; see §2.5).
The assignment statement first evaluates all its expressions and only then are the assignments performed. Thus the code

```
i = 3
     i, a[i] = i+1, 20
```

sets `a[3]` to 20, without affecting `a[4]` because the `i` in `a[i]` is evaluated (to 3) before it is assigned 4. Similarly, the line

```
x, y = y, x
```

exchanges the values of `x` and `y`, and

```
x, y, z = y, z, x
```

cyclically permutes the values of `x`, `y`, and `z`.
The meaning of assignments to global variables and table fields can be changed via metatables. An assignment to an indexed variable `t[i] = val` is equivalent to `settable_event(t,i,val)`. (See §2.8 for a complete description of the `settable_event` function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)
An assignment to a global variable `x = val` is equivalent to the assignment `_env.x = val`, which in turn is equivalent to

```
settable_event(_env, "x", val)
```

where `_env` is the environment of the running function. (The `_env` variable is not defined in Lua. We use it here only for explanatory purposes.)

### 1.4.4 2.4.4 - Control Structures

The control structures **if**, **while**, and **repeat** have the usual meaning and familiar syntax:

```
stat ::= {\bf while} exp {\bf do} block {\bf end}
       stat ::= {\bf repeat} block {\bf until} exp
       stat ::= {\bf if} exp {\bf then} block {{\bf elseif} exp {\bf
then} block} [{\bf else} block] {\bf end}
```

Lua also has a **for** statement, in two flavors (see §2.4.5).
The condition expression of a control structure can return any value. Both **false** and **nil** are considered false. All values different from **nil** and **false** are considered true (in particular, the number 0 and the empty string are also true).
In the **repeatůuntil** 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 **return** statement is used to return values from a function or a chunk (which is just a function).
Functions and chunks can return more than one value, and so the syntax for the **return** statement is

```
stat ::= {\bf return} [explist]
```

The **break** statement is used to terminate the execution of a **while**, **repeat**, or **for** loop, skipping to the next statement after the loop:

```
stat ::= {\bf break}
```

A **break** ends the innermost enclosing loop.
The **return** and **break** statements can only be written as the *last* statement of a block. If it is really necessary to **return** or **break** in the middle of a block, then an explicit inner block can be used, as in the idioms `do return end` and `do break end`, because now **return** and **break** are the last statements in their (inner) blocks.

### 1.4.5 2.4.5 - 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. It has the following syntax:

```
stat ::= {\bf for} Name `{\bf =}  exp `{\bf ,}  exp [`{\bf ,}  exp]
{\bf do} block {\bf end}
```

The *block* is repeated for *name* starting at the value of the first *exp*, until it passes the second *exp* by steps of the third *exp*. More precisely, a **for** statement like

```
for v = {\em e1}, {\em e2}, {\em e3} do {\em block} end
```

is equivalent to the code:

```
do
       local var, limit, step = tonumber({\em e1}), tonumber({\em
e2}), tonumber({\em e3})
       if not (var and limit and step) then error() end
       while (step > 0 and var <= limit) or (step <= 0 and var >=
limit) do
         local v = var
         {\em block}
```

```
        var = var + step
      end
    end
```

Note the following:

- All three control expressions are evaluated only once, before the loop starts. They must all result in numbers.
- `var`, `limit`, and `step` are invisible variables. The names shown here are for explanatory purposes only.
- If the third expression (the step) is absent, then a step of 1 is used.
- You can use **break** to exit a **for** loop.
- The loop variable `v` is local to the loop; you cannot use its value after the **for** ends or is broken. 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**. The generic **for** loop has the following syntax:

```
stat ::= {\bf for} namelist {\bf in} explist {\bf do} block {\bf end}
       namelist ::= Name {`{\bf ,}  Name}
```

A **for** statement like

```
for {\em var_1}, , {\em var_n} in explist do {\em block} end
```

is equivalent to the code:

```
do
       local f, s, var = explist
       while true do
         local {\em var_1}, , {\em var_n} = f(s, var)
         var = {\em var_1}
         if var == nil then break end
         {\em block}
       end
     end
```

Note the following:

- `explist` is evaluated only once. Its results are an *iterator* function, a *state*, and an initial value for the first *iterator variable*.
- `f`, `s`, and `var` are invisible variables. The names are here for explanatory purposes only.
- You can use **break** to exit a **for** loop.
- The loop variables `var_i` are local to the loop; you cannot use their values after the **for** ends. If you need these values, then assign them to other variables before breaking or exiting the loop.

### 1.4.6 2.4.6 - Function Calls as Statements

To allow possible side-effects, function calls can be executed as statements:

```
stat ::= functioncall
```

In this case, all returned values are thrown away. Function calls are explained in §2.5.8.

### 1.4.7 2.4.7 - Local Declarations

Local variables can be declared anywhere inside a block. The declaration can include an initial assignment:

```
stat ::= {\bf local} namelist [`{\bf =}  explist]
```

If present, an initial assignment has the same semantics of a multiple assignment (see §2.4.3). Otherwise, all variables are initialized with **nil**.
A chunk is also a block (see §2.4.1), and so local variables can be declared in a chunk outside any explicit block. The scope of such local variables extends until the end of the chunk.
The visibility rules for local variables are explained in §2.6.

## 1.5 2.5 - Expressions

The basic expressions in Lua are the following:

```
exp ::= prefixexp
       exp ::= {\bf nil} | {\bf false} | {\bf true}
       exp ::= Number
       exp ::= String
       exp ::= function
       exp ::= tableconstructor
       exp ::= `{\bf ...}
       exp ::= exp binop exp
       exp ::= unop exp
       prefixexp ::= var | functioncall | `{\bf (}  exp `{\bf )}
```

Numbers and literal strings are explained in §2.1; variables are explained in §2.3; function definitions are explained in §2.5.9; function calls are explained in §2.5.8; table constructors are explained in §2.5.7. Vararg expressions, denoted by three dots ('...'), can only be used when directly inside a vararg function; they are explained in §2.5.9.
Binary operators comprise arithmetic operators (see §2.5.1), relational operators (see §2.5.2), logical operators (see §2.5.3), and the concatenation operator (see §2.5.4). Unary operators comprise the unary minus (see §2.5.1), the unary **not** (see §2.5.3), and the unary *length operator* (see §2.5.5).
Both function calls and vararg expressions can result in multiple values. If an expression is used as a statement (only possible for function calls (see §2.4.6)), then its return list is adjusted to zero elements, thus discarding all returned values. If an expression is used as the last (or the only) element of a list of expressions, then no adjustment is made (unless the call is enclosed in parentheses). In all other contexts, Lua adjusts the result list to one element, discarding all values except the first one.
Here are some examples:

```
f()                -- adjusted to 0 results
     g(f(), x)          -- f() is adjusted to 1 result
     g(x, f())          -- g gets x plus all results from f()
     a,b,c = f(), x     -- f() is adjusted to 1 result (c gets nil)
     a,b = ...          -- a gets the first vararg parameter, b gets
                        -- the second (both a and b can get nil if
there
                        -- is no corresponding vararg parameter)

     a,b,c = x, f()     -- f() is adjusted to 2 results
     a,b,c = f()        -- f() is adjusted to 3 results
     return f()         -- returns all results from f()
     return ...          -- returns all received vararg parameters
     return x,y,f()      -- returns x, y, and all results from f()
     {f()}               -- creates a list with all results from f()
     {...}               -- creates a list with all vararg parameters
     {f(), nil}          -- f() is adjusted to 1 result
```

Any expression enclosed in parentheses always results in only one value. Thus, `(f(x,y,z))` is always a single value, even if `f` returns several values. (The value of `(f(x,y,z))` is the first value returned by `f` or **nil** if `f` does not return any values.)

### 1.5.1 2.5.1 - 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 (see §2.2.1), then all operations have the usual meaning. Exponentiation works for any exponent. For instance, `x^(-0.5)` computes the inverse of the square root of `x`. Modulo is defined as

```
a % b == a - math.floor(a/b)*b
```

That is, it is the remainder of a division that rounds the quotient towards minus infinity.

### 1.5.2 2.5.2 - Relational Operators

The relational operators in Lua are

```
==     ~=     <     >     <=     >=
```

These operators always result in **false** or **true**.
Equality (==) first compares the type of its operands. If the types are different, then the result is **false**. Otherwise, the values of the operands are compared. Numbers and strings are compared in the usual way. Objects (tables, userdata, threads, and functions) are compared by *reference*: two objects are considered equal only if they are the *same* object. Every time you create a new object (a table, userdata, thread, or function), this new object is different from any previously existing object.
You can change the way that Lua compares tables and userdata by using the "eq" metamethod (see §2.8).
The conversion rules of §2.2.1 *do not* apply to equality comparisons. Thus, `"0"==0` evaluates to **false**, and `t[0]` and `t["0"]` denote different entries in a table.

The operator `~=` is exactly the negation of equality (`==`).
The order operators work as follows. If both arguments are numbers, then they are compared as such. Otherwise, if both arguments are strings, then their values are compared according to the current locale. Otherwise, Lua tries to call the "lt" or the "le" metamethod (see §2.8). A comparison `a > b` is translated to `b < a` and `a >= b` is translated to `b <= a`.

### 1.5.3 2.5.3 - Logical Operators

The logical operators in Lua are **and**, **or**, and **not**. Like the control structures (see §2.4.4), all logical operators consider both **false** and **nil** as false and anything else as true.
The negation operator **not** always returns **false** or **true**. The conjunction operator **and** returns its first argument if this value is **false** or **nil**; otherwise, **and** returns its second argument. The disjunction operator **or** returns its first argument if this value is different from **nil** and **false**; otherwise, **or** returns its second argument. Both **and** and **or** use short-cut evaluation; that is, the second operand is evaluated only if necessary. Here are some examples:

```
10 or 20            --> 10
     10 or error()       --> 10
     nil or "a"          --> "a"
     nil and 10          --> nil
     false and error()   --> false
     false and nil       --> false
     false or nil        --> nil
     10 and 20           --> 20
```

(In this manual, `-->` indicates the result of the preceding expression.)

### 1.5.4 2.5.4 - Concatenation

The string concatenation operator in Lua is denoted by two dots ('`..`'). If both operands are strings or numbers, then they are converted to strings according to the rules mentioned in §2.2.1. Otherwise, the "concat" metamethod is called (see §2.8).

### 1.5.5 2.5.5 - 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 of a table `t` is defined to be any integer index `n` such that `t[n]` is not **nil** and `t[n+1]` is **nil**; moreover, if `t[1]` is **nil**, `n` can be zero. For a regular array, with non-nil values from 1 to a given `n`, its length is exactly that `n`, the index of its last value. If the array has "holes" (that is, **nil** values between other non-nil values), then `#t` can be any of the indices that directly precedes a **nil** value (that is, it may consider any such **nil** value as the end of the array).

## 1.5.6 2.5.6 - 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 precedences of an expression. The concatenation ('`..`') and exponentiation ('`^`') operators are right associative. All other binary operators are left associative.

## 1.5.7 2.5.7 - Table Constructors

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

```
tableconstructor ::= `{\bf {}  [fieldlist] `{\bf }}
       fieldlist ::= field {fieldsep field} [fieldsep]
       field ::= `{\bf [}  exp `{\bf ]}  `{\bf =}  exp | Name `{\bf
=}  exp | exp
       fieldsep ::= `{\bf ,}  | `{\bf ;}
```

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
```

If the last field in the list has the form `exp` and the expression is a function call or a vararg expression, then all values returned by this expression enter the list consecutively (see §2.5.8). To avoid this, enclose the function call or the vararg expression in parentheses (see §2.5).
The field list can have an optional trailing separator, as a convenience for machine-generated code.

### 1.5.8 2.5.8 - Function Calls

A function call in Lua has the following syntax:

```
functioncall ::= prefixexp args
```

In a function call, first prefixexp and args are evaluated. If the value of prefixexp has type *function*, then this function is called with the given arguments. Otherwise, the prefixexp "call" metamethod is called, having as first parameter the value of prefixexp, followed by the original call arguments (see §2.8).
The form

```
functioncall ::= prefixexp `{\bf :}  Name args
```

can be used to call "methods". A call `v:name(args)` is syntactic sugar for `v.name(v,args)`, except that `v` is evaluated only once.

Arguments have the following syntax:

```
args ::= `{\bf (}  [explist] `{\bf )}
       args ::= tableconstructor
       args ::= String
```

All argument expressions are evaluated before the call. A call of the form `f{fields}` is syntactic sugar for `f({fields})`; that is, the argument list is a single new table. A call of the form `f'string'` (or `f"string"` or `f[[string]]`) is syntactic sugar for `f('string')`; that is, the argument list is a single literal string.
As an exception to the free-format syntax of Lua, you cannot put a line break before the '`(`' in a function call. This restriction avoids some ambiguities in the language. If you write

```
a = f
     (g).x(a)
```

Lua would see that as a single statement, `a = f(g).x(a)`. So, if you want two statements, you must add a semi-colon between them. If you actually want to call `f`, you must remove the line break before `(g)`.
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. However, a tail call erases any debug information about the calling function. Note that a tail call only happens with a particular syntax, where the **return** has one single function call as argument; this syntax makes the calling function return exactly the returns of the called function. So, none of the following examples are tail calls:

```
return (f(x))        -- results adjusted to 1
     return 2 * f(x)
     return x, f(x)       -- additional results
     f(x); return         -- results discarded
     return x or f(x)     -- results adjusted to 1
```

### 1.5.9 2.5.9 - Function Definitions

The syntax for function definition is

```
function ::= {\bf function} funcbody
       funcbody ::= `{\bf (}  [parlist] `{\bf )}  block {\bf end}
```

The following syntactic sugar simplifies function definitions:

```
stat ::= {\bf function} funcname funcbody
       stat ::= {\bf local} {\bf function} Name funcbody
       funcname ::= Name {`{\bf .}  Name} [`{\bf :}  Name]
```

The statement

```
function f () {\em body} end
```

translates to

```
f = function () {\em body} end
```

The statement

```
function t.a.b.c.f () {\em body} end
```

translates to

```
t.a.b.c.f = function () {\em body} end
```

The statement

```
local function f () {\em body} end
```

translates to

```
local f; f = function () {\em body} end
```

*not* to

```
local f = function () {\em body} end
```

(This only makes a difference when the body of the function contains references to `f`.)
A function definition is an executable expression, whose value has type *function*. When Lua pre-compiles a chunk, all its function bodies are pre-compiled too. Then, whenever Lua executes the function definition, the function is *instantiated* (or *closed*). This function instance (or *closure*) is the final value of the expression. Different instances of the same function can refer to different external local variables and can have different environment tables.
Parameters act as local variables that are initialized with the argument values:

```
parlist ::= namelist [`{\bf ,}  `{\bf ...} ] | `{\bf ...}
```

When a function is called, the list of arguments is adjusted to the length of the list of parameters, unless the function is a variadic or *vararg function*, which is indicated by three dots ('...') at the end of its parameter list. A vararg function does not adjust its argument list; instead, it collects all extra arguments and supplies them to the function through a *vararg expression*, which is also written as three dots. The value of this expression is a list of all actual extra arguments, similar to a function with multiple results. If a vararg expression is used inside another expression or in the middle of a list of expressions, then its return list is adjusted to one element. If the expression is used as the last element of a list of expressions, then no adjustment is made (unless that last expression is enclosed in parentheses).
As an example, consider the following definitions:

```
function f(a, b) end
     function g(a, b, ...) end
     function r() return 1,2,3 end
```

Then, we have the following mapping from arguments to parameters and to the vararg expression:

```
CALL            PARAMETERS

     f(3)             a=3, b=nil
     f(3, 4)          a=3, b=4
     f(3, 4, 5)       a=3, b=4
     f(r(), 10)       a=1, b=10
     f(r())           a=1, b=2

     g(3)             a=3, b=nil, ... -->  (nothing)
     g(3, 4)          a=3, b=4,   ... -->  (nothing)
     g(3, 4, 5, 8)    a=3, b=4,   ... -->  5  8
     g(5, r())        a=5, b=1,   ... -->  2  3
```

Results are returned using the **return** statement (see §2.4.4). If control reaches the end of a function without encountering a **return** statement, then the function returns with no results.
The *colon* syntax is used for defining *methods*, that is, functions that have an implicit extra parameter `self`. Thus, the statement

```
function t.a.b.c:f ({\em params}) {\em body} end
```

is syntactic sugar for

```
t.a.b.c.f = function (self, {\em params}) {\em body} end
```

## 1.6 2.6 - Visibility Rules

Lua is a lexically scoped language. 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 one)
```

Notice that, 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.
Because of the lexical scoping rules, local variables can be freely accessed by functions defined inside their scope. A local variable used by an inner function is called an *upvalue*, or *external local variable*, inside the inner function.
Notice that each execution of a **local** statement defines new local variables. Consider the following example:

```
a = {}
     local x = 20
     for i=1,10 do
       local y = 0
       a[i] = function () y=y+1; return x+y end
     end
```

The loop creates ten closures (that is, ten instances of the anonymous function). Each of these closures uses a different `y` variable, while all of them share the same `x`.

## 1.7 2.7 - Error Handling

Because Lua is an embedded extension language, all Lua actions start from C code in the host program calling a function from the Lua library (see `lua_pcall`). 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. If you need to catch errors in Lua, you can use the `pcall` function.

## 1.8 2.8 - Metatables

Every value in Lua can have a *metatable*. This *metatable* is an ordinary Lua table that defines the behavior of the original value under certain special operations. You can change several aspects of the behavior of operations over a value by setting specific fields in its metatable. For instance, when a non-numeric value is the operand of an addition, Lua checks for a function in the field `"__add"` in its metatable. If it finds one, Lua calls this function to perform the addition.
We call the keys in a metatable *events* and the values *metamethods*. In the previous example, the event is `"add"` and the metamethod is the function that performs the addition.
You can query the metatable of any value through the `getmetatable` function.
You can replace the metatable of tables through the `setmetatable` function. You cannot change the metatable of other types from Lua (except by using the debug library); you must use the C API for that.
Tables and full userdata have individual metatables (although multiple tables and userdata can share their metatables). Values of all other types share one single metatable per type; that is, there is one single metatable for all numbers, one for all strings, etc.
A metatable controls how an object behaves in arithmetic operations, order comparisons, concatenation, length operation, and indexing. A metatable also can define a function to be called when a userdata is garbage collected. For each of these operations Lua associates a specific key called an *event*. When Lua performs one of these operations over a value, it checks whether this value has a metatable with the corresponding event. If so, the value associated with that key (the metamethod) controls how Lua will perform the operation.

Metatables control the operations listed next. Each operation is identified by its corresponding name. The key for each operation is a string with its name prefixed by two underscores, '`__`'; for instance, the key for operation "add" is the string `"__add"`. The semantics of these operations is better explained by a Lua function describing how the interpreter executes the operation.
The code shown here in Lua is only illustrative; the real behavior is hard coded in the interpreter and it is much more efficient than this simulation. All functions used in these descriptions (`rawget`, `tonumber`, etc.) are described in `§5.1`. In particular, to retrieve the metamethod of a given object, we use the expression

```
metatable(obj)[event]
```

This should be read as

```
rawget(getmetatable(obj) or {}, event)
```

That is, the access to a metamethod does not invoke other metamethods, and the access to objects with no metatables does not fail (it simply results in **nil**).

- **"add":** the `+` operation.
  The function `getbinhandler` below defines how Lua chooses a handler for a binary operation. First, Lua tries the first operand. If its type does not define a handler for the operation, then Lua tries the second operand.

  ```
  function getbinhandler (op1, op2, event)
         return metatable(op1)[event] or metatable(op2)[event]
       end
  ```

  By using this function, the behavior of the `op1 + op2` is

  ```
  function add_event (op1, op2)
         local o1, o2 = tonumber(op1), tonumber(op2)
         if o1 and o2 then  -- both operands are numeric?
           return o1 + o2   -- '+' here is the primitive 'add'
         else  -- at least one of the operands is not numeric
           local h = getbinhandler(op1, op2, "__add")
           if h then
             -- call the handler with both operands
             return (h(op1, op2))
           else  -- no handler available: default behavior
             error()
           end
  ```

```
          end
        end
```

- **"sub":** the `-` operation.
  Behavior similar to the "add" operation.

- **"mul":** the `*` operation.
  Behavior similar to the "add" operation.

- **"div":** the `/` operation.
  Behavior similar to the "add" operation.

- **"mod":** the `%` operation.
  Behavior similar to the "add" operation, with the operation `o1 - floor(o1/o2)*o2` as the primitive operation.

- **"pow":** the `^` (exponentiation) operation.
  Behavior similar to the "add" operation, with the function `pow` (from the C math library) as the primitive operation.

- **"unm":** the unary `-` operation.

  ```
  function unm_event (op)
         local o = tonumber(op)
         if o then  -- operand is numeric?
           return -o  -- '-' here is the primitive 'unm'
         else  -- the operand is not numeric.
           -- Try to get a handler from the operand
           local h = metatable(op).__unm
           if h then
             -- call the handler with the operand
             return (h(op))
           else  -- no handler available: default behavior
             error()
           end
         end
       end
  ```

- **"concat":** the `..` (concatenation) operation.

  ```
  function concat_event (op1, op2)
         if (type(op1) == "string" or type(op1) == "number") and
            (type(op2) == "string" or type(op2) == "number") then
  ```

```
          return op1 .. op2  -- primitive string concatenation
        else
          local h = getbinhandler(op1, op2, "__concat")
          if h then
            return (h(op1, op2))
          else
            error()
          end
        end
      end
```

- **"len":** the # operation.

  ```
  function len_event (op)
         if type(op) == "string" then
           return strlen(op)         -- primitive string length
         elseif type(op) == "table" then
           return #op                -- primitive table length
         else
           local h = metatable(op).__len
           if h then
             -- call the handler with the operand
             return (h(op))
           else  -- no handler available: default behavior
             error()
           end
         end
       end
  ```

  See §2.5.5 for a description of the length of a table.

- **"eq":** the == operation.
  The function `getcomphandler` defines how Lua chooses a metamethod for comparison operators. A metamethod only is selected when both objects being compared have the same type and the same metamethod for the selected operation.

  ```
  function getcomphandler (op1, op2, event)
         if type(op1) ~= type(op2) then return nil end
         local mm1 = metatable(op1)[event]
         local mm2 = metatable(op2)[event]
  ```

```
         if mm1 == mm2 then return mm1 else return nil end
       end
```

The "eq" event is defined as follows:

```
function eq_event (op1, op2)
         if type(op1) ~= type(op2) then  -- different types?
           return false   -- different objects
         end
         if op1 == op2 then   -- primitive equal?
           return true   -- objects are equal
         end
         -- try metamethod
         local h = getcomphandler(op1, op2, "__eq")
         if h then
           return (h(op1, op2))
         else
           return false
         end
       end
```

`a ~= b` is equivalent to `not (a == b)`.

- **"lt":** the `<` operation.

```
function lt_event (op1, op2)
         if type(op1) == "number" and type(op2) == "number" then
           return op1 < op2   -- numeric comparison
         elseif type(op1) == "string" and type(op2) == "string" then
           return op1 < op2   -- lexicographic comparison
         else
           local h = getcomphandler(op1, op2, "__lt")
           if h then
             return (h(op1, op2))
           else
             error()
           end
         end
       end
```

`a > b` is equivalent to `b < a`.

- **"le":** the `<=` operation.

  ```
  function le_event (op1, op2)
         if type(op1) == "number" and type(op2) == "number" then
           return op1 <= op2   -- numeric comparison
         elseif type(op1) == "string" and type(op2) == "string" then
           return op1 <= op2   -- lexicographic comparison
         else
           local h = getcomphandler(op1, op2, "__le")
           if h then
             return (h(op1, op2))
           else
             h = getcomphandler(op1, op2, "__lt")
             if h then
               return not h(op2, op1)
             else
               error()
             end
           end
         end
       end
  ```

  `a >= b` is equivalent to `b <= a`. Note that, in the absence of a "le" metamethod, Lua tries the "lt", assuming that `a <= b` is equivalent to `not (b < a)`.

- **"index":** The indexing access `table[key]`.

  ```
  function gettable_event (table, key)
         local h
         if type(table) == "table" then
           local v = rawget(table, key)
           if v ~= nil then return v end
           h = metatable(table).__index
           if h == nil then return nil end
         else
           h = metatable(table).__index
           if h == nil then
             error()
           end
         end
         if type(h) == "function" then
           return (h(table, key))     -- call the handler
  ```

```
       else return h[key]           -- or repeat operation on it
       end
     end
```

- **"newindex":** The indexing assignment `table[key] = value`.

```
function settable_event (table, key, value)
       local h
       if type(table) == "table" then
         local v = rawget(table, key)
         if v ~= nil then rawset(table, key, value); return end
         h = metatable(table).__newindex
         if h == nil then rawset(table, key, value); return end
       else
         h = metatable(table).__newindex
         if h == nil then
           error()
         end
       end
       if type(h) == "function" then
         h(table, key,value)           -- call the handler
       else h[key] = value             -- or repeat operation on
it
       end
     end
```

- **"call":** called when Lua calls a value.

```
function function_event (func, ...)
       if type(func) == "function" then
         return func(...)   -- primitive call
       else
         local h = metatable(func).__call
         if h then
           return h(func, ...)
         else
           error()
         end
```

```
      end
    end
```

## 1.9 2.9 - Environments

Besides metatables, objects of types thread, function, and userdata have another table associated with them, called their *environment*. Like metatables, environments are regular tables and multiple objects can share the same environment.

Threads are created sharing the environment of the creating thread. Userdata and C functions are created sharing the environment of the creating C function. Non-nested Lua functions (created by `loadfile`, `loadstring` or `load`) are created sharing the environment of the creating thread. Nested Lua functions are created sharing the environment of the creating Lua function.

Environments associated with userdata have no meaning for Lua. It is only a convenience feature for programmers to associate a table to a userdata.

Environments associated with threads are called *global environments*. They are used as the default environment for threads and non-nested Lua functions created by the thread and can be directly accessed by C code (see `§3.3`).

The environment associated with a C function can be directly accessed by C code (see `§3.3`). It is used as the default environment for other C functions and userdata created by the function.

Environments associated with Lua functions are used to resolve all accesses to global variables within the function (see `§2.3`). They are used as the default environment for nested Lua functions created by the function.

You can change the environment of a Lua function or the running thread by calling `setfenv`. You can get the environment of a Lua function or the running thread by calling `getfenv`. To manipulate the environment of other objects (userdata, C functions, other threads) you must use the C API.

## 1.10 2.10 - 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, objects that are no longer accessible from Lua). All memory used by Lua is subject to automatic management: tables, userdata, functions, threads, strings, etc.

Lua implements an incremental mark-and-sweep collector. It uses two numbers to control its garbage-collection cycles: the *garbage-collector pause* and the

*garbage-collector step multiplier*. Both use percentage points as units (so that a value of 100 means an internal value of 1).
The garbage-collector pause controls how long the collector waits before starting a new cycle. Larger values make the collector less aggressive. Values smaller than 100 mean the collector will not wait to start a new cycle. A value of 200 means that the collector waits for the total memory in use to double before starting a new cycle.
The step multiplier controls the relative speed of the collector relative to memory allocation. Larger values make the collector more aggressive but also increase the size of each incremental step. Values smaller than 100 make the collector too slow and can result in the collector never finishing a cycle. The default, 200, means that the collector runs at "twice" the speed of memory allocation.
You can change these numbers by calling `lua_gc` in C or `collectgarbage` in Lua. With these functions you can also control the collector directly (e.g., stop and restart it).

### 1.10.1 2.10.1 - Garbage-Collection Metamethods

Using the C API, you can set garbage-collector metamethods for userdata (see §2.8). These metamethods are also called *finalizers*. Finalizers allow you to coordinate Lua's garbage collection with external resource management (such as closing files, network or database connections, or freeing your own memory).
Garbage userdata with a field `__gc` in their metatables are not collected immediately by the garbage collector. Instead, Lua puts them in a list. After the collection, Lua does the equivalent of the following function for each userdata in that list:

```
function gc_event (udata)
       local h = metatable(udata).__gc
       if h then
         h(udata)
       end
     end
```

At the end of each garbage-collection cycle, the finalizers for userdata are called in *reverse* order of their creation, among those collected in that cycle. That is, the first finalizer to be called is the one associated with the userdata created last in the program. The userdata itself is freed only in the next garbage-collection cycle.

### 1.10.2 2.10.2 - Weak Tables

A *weak table* is a table whose elements are *weak references*. A weak reference is ignored by the garbage collector. In other words, if the only references to an object are weak references, then the garbage collector will collect this object.

A weak table can have weak keys, weak values, or both. A table with weak keys allows the collection of its keys, but prevents the collection of its values. A table with both weak keys and weak values allows the collection of both keys and values. In any case, if either the key or the value is collected, the whole pair is removed from the table. The weakness of a table is controlled by the `__mode` field of its metatable. If the `__mode` field is a string containing the character '`k`', the keys in the table are weak. If `__mode` contains '`v`', the values in the table are weak.
After you use a table as a metatable, you should not change the value of its `__mode` field. Otherwise, the weak behavior of the tables controlled by this metatable is undefined.

## 1.11 2.11 - Coroutines

Lua supports coroutines, also called *collaborative multithreading*. A coroutine in Lua represents an independent thread of execution. Unlike threads in multithread systems, however, a coroutine only suspends its execution by explicitly calling a yield function.
You create a coroutine with a call to `coroutine.create`. Its sole argument is a function that is the main function of the coroutine. The `create` function only creates a new coroutine and returns a handle to it (an object of type *thread*); it does not start the coroutine execution.
When you first call `coroutine.resume`, passing as its first argument a thread returned by `coroutine.create`, the coroutine starts its execution, at the first line of its main function. Extra arguments passed to `coroutine.resume` are passed on to the coroutine main function. After the coroutine starts running, it runs until it terminates or *yields*.
A coroutine can terminate its execution in two ways: normally, when its main function returns (explicitly or implicitly, after the last instruction); and abnormally, if there is an unprotected error. In the first case, `coroutine.resume` returns **true**, plus any values returned by the coroutine main function. In case of errors, `coroutine.resume` returns **false** plus an error message.
A coroutine yields by calling `coroutine.yield`. When a coroutine yields, the corresponding `coroutine.resume` returns immediately, even if the yield happens inside nested function calls (that is, not in the main function, but in a function directly or indirectly called by the main function). In the case of a yield, `coroutine.resume` also returns **true**, plus any values passed to `coroutine.yield`. The next time you resume the same coroutine, it continues its execution from the point where it yielded, with the call to `coroutine.yield` returning any extra arguments passed to `coroutine.resume`.

Like `coroutine.create`, the `coroutine.wrap` function also creates a coroutine, but instead of returning the coroutine itself, it returns a function that, when called, resumes the coroutine. Any arguments passed to this function go as extra arguments to `coroutine.resume`. `coroutine.wrap` returns all the values returned by `coroutine.resume`, except the first one (the boolean error code). Unlike `coroutine.resume`, `coroutine.wrap` does not catch errors; any error is propagated to the caller.
As an example, consider the following code:

```
function foo (a)
       print("foo", a)
       return coroutine.yield(2*a)
     end

     co = coroutine.create(function (a,b)
           print("co-body", a, b)
           local r = foo(a+1)
           print("co-body", r)
           local r, s = coroutine.yield(a+b, a-b)
           print("co-body", r, s)
           return b, "end"
     end)

     print("main", coroutine.resume(co, 1, 10))
     print("main", coroutine.resume(co, "r"))
     print("main", coroutine.resume(co, "x", "y"))
     print("main", coroutine.resume(co, "x", "y"))
```

When you run it, it produces the following output:

```
co-body 1       10
     foo     2

     main    true    4
     co-body r
     main    true    11      -9
     co-body x       y
     main    true    10      end
     main    false   cannot resume dead coroutine
```