The Lua language (v5.0)

Reserved identifiers and comments

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

_X... (where **X** = any uppercase letter)
-- Comment to end of line.
--[[... **]]** Multi-line comment
#! Ignored at the start of the first line (to make Unix-executable).

Types

"nil" **"boolean"** **"number"** **"string"** **"table"** **"function"** **"thread"** **"userdata"**
Note: for booleans, **nil** and **false** count as **false**, all the rest is **true** including **0** and **""**.

Strings and escape sequences

' ' **" "** **[[ ]]** string delimiters; **[[ ]]** can be multi-line, escape sequences are ignored.
\a (bell) **\b** (backspace) **\f** (form feed) **\n** (newline) **\r** (return)
\t (horiz. tab) **\v** (vert. tab) **** (backslash) **\"** (d. quote) **\'** (quote)
\[(sq. bracket) **\]** (sq. bracket) **\ddd** (decimal)

Operators, decreasing precedence

^ (right-associative, math lib required)
not **-** (unary)
***** /
+ -
.. (string concatenation, right-associative)
< > <= >= ~= ==
and (stops on false/nil, returns last evaluated value)
or (stops on true (not false/nil), returns last evaluated value)

Assignment and coercion

a = 5 Simple assignment.
a = "hi" Variables are not typed, they can hold different types.
a, b, c = 1, 2, 3 Multiple assignment.
a, b = b, a Swap values, because right side values are evaluated before assignment.
a, b = 4, 5, 6 Too many values, **6** is discarded.
a, b = "there" Too few values, **nil** is assigned to **b**.
a = nil Destroys **a**, its value will be eligible for garbage collection if unreferenced.
a = z If **z** is not defined it is **nil**, so **nil** is assigned to **a** (destroying it).
a = "3" + "2" Numbers expected, strings are converted to numbers (a = 5).
a = 3 .. 2 Strings expected, numbers are converted to strings (a = "32").

Control structures

do *block* **end** Block with local scope.
while *exp* **do** *block* **end** Loop as long as *exp* is true.
repeat *block* **until** *exp* Exits when *exp* becomes true.
if *exp* **then** *block* {**elseif** *exp* **then** *block*} [**else** *block*] **end** Conditional execution.
for *var* = *start*, *end* [, *step*] **do** *block* **end** Counter-based loop.
for *vars* **in** *iterator* **do** *block* **end** Iterator-based loop.
break Exits loop, must be last in block.

Table constructors

t = {} A new empty table.
t = {"yes", "no", "?"} Simple array, elements are **t[1]**, **t[2]**, **t[3]**.
t = {[1] = "yes", [2] = "no", [3] = "?"} Same as line above.
t = {[-900] = 3, [+900] = 4} Sparse array, two elements (no space lost).
t = {x=5, y=10} Hash table, fields are **t["x"]**, **t["y"]** or **t.x**, **t.y**.
t = {x=5, y=10; "yes", "no"} Mixed, fields / elements are **t.x**, **t.y**, **t[1]**, **t[2]**.
t = {msg = "choice", {"yes", "no", "?"}} Table containing a table as field.

Function definition

function *name* **(** *args* **)** *body* [**return** *values*] **end** Global function.
local function *name* **(** *args* **)** *body* [**return** *values*] **end** Function local to chunk.
f = function (*args* **)** *body* [**return** *values*] **end** Anonymous function.
function ([*args*,] **...)** *body* [**return** *values*] **end** Variable args, passed as **arg[], arg.n**.
function *t*.*name* **(** *args* **)** *body* [**return** *values*] **end** Shortcut for *t*.*name* = **function** [...]
function *obj*:*name* **(** *args* **)** *body* [**return** *values*] **end** Object function getting extra arg **self**.

Function call

f (x) Simple call, possibly returning one or more values.
f "hello" Shortcut for **f ("hello")**.
f 'goodbye' Shortcut for **f ('goodbye')**.
f [[see you soon]] Shortcut for **f ([[see you soon]])**.
f {x = 3, y = 4} Shortcut for **f ({x = 3, y = 4})**.
t.f (x) Calling a function stored in the field **f** of table **t**.
x:move (2, -3) Object call, shortcut for **x.move (x, 2, -3)**, **x** will be assigned to **self**.

Metatable operations (basic library required)

setmetatable (t, mt) Sets **mt** as metatable for **t**, unless **t**'s metatable has a **__metatable** field.
getmetatable (t) Returns **__metatable** field of **t**'s metatable, or **t**'s metatable, or **nil**.
rawget (t, i) Gets **t[i]** of a table without invoking metamethods.
rawset (t, i, v) Sets **t[i] = v** on a table without invoking metamethods.
rawequal (t1, t2) Returns boolean **(t1 == t2)** without invoking metamethods.

Metatable fields (for tables and userdata)

__add Sets handler **h (a, b)** for '+'.
__sub Sets handler **h (a, b)** for binary '-'.
__mul Sets handler **h (a, b)** for '*'.
__div Sets handler **h (a, b)** for '/'.
__pow Sets handler **h (a, b)** for '^'.
__unm Sets handler **h (a)** for unary '-'.
__concat Sets handler **h (a, b)** for '..'.
__eq Sets handler **h (a, b)** for '==', '~='.
__lt Sets handler **h (a, b)** for '<', '>' and possibly '<=', '>=' (if no **__le**).
__le Sets handler **h (a, b)** for '<=', '>='.
__index Sets handler **h (t, k)** for non-existing field access.
__newindex Sets handler **h (t, k)** for new field assignment.
__call Sets handler **h (f, ...)** for function call (using the object as a function).
__tostring Sets handler **h (a)** to convert to string, e.g. for **print ()**.
__gc Sets finalizer **h (ud)** for userdata (can be set from the C side only).
__mode Table mode: **'k'** = weak keys, **'v'** = weak values, **'kv'** = both.
__metatable Sets a value to be returned by **getmetatable ().**

7 - Incompatibilities with the Previous Version

Here we list the incompatibilities that may be found when moving a program from Lua 5.0 to Lua 5.1. You can avoid most of the incompatibilities compiling Lua with appropriate options (see file `luaconf.h`). However, all these compatibility options will be removed in the next version of Lua.

7.1 - Changes in the Language

- The vararg system changed from the pseudo-argument `arg` with a table with the extra arguments to the vararg expression. (Option `LUA_COMPAT_VARARG` in `luaconf.h`.)
- There was a subtle change in the scope of the implicit variables of the **for** statement and for the **repeat** statement.
- The long string/long comment syntax (`[[`*string*`]]`) does not allow nesting. You can use the new syntax (`[=[`*string*`]=]`) in these cases. (Option `LUA_COMPAT_LSTR` in `luaconf.h`.)

7.2 - Changes in the Libraries

- Function `string.gfind` was renamed `string.gmatch`. (Option `LUA_COMPAT_GFIND`)
- When `string.gsub` is called with a function as its third argument, whenever this function returns **nil** or **false** the replacement string is the whole match, instead of the empty string.
- Function `table.setn` was deprecated. Function `table.getn` corresponds to the new length operator (`#`); use the operator instead of the function. (Option `LUA_COMPAT_GETN`)
- Function `loadlib` was renamed `package.loadlib`. (Option `LUA_COMPAT_LOADLIB`)
- Function `math.mod` was renamed `math.fmod`. (Option `LUA_COMPAT_MOD`)
- Functions `table.foreach` and `table.foreachi` are deprecated. You can use a for loop with `pairs` or `ipairs` instead.
- There were substantial changes in function `require` due to the new module system. However, the new behavior is mostly compatible with the old, but `require` gets the path from `package.path` instead of from `LUA_PATH`.
- Function `collectgarbage` has different arguments. Function `gcinfo` is deprecated; use `collectgarbage("count")` instead.

7.3 - Changes in the API

- The `luaopen_*` functions (to open libraries) cannot be called directly, like a regular C function. They must be called through Lua, like a Lua function.
- Function `lua_open` was replaced by `lua_newstate` to allow the user to set a memory-allocation function. You can use `luaL_newstate` from the standard library to create a state with a standard allocation function (based on `realloc`).
- Functions `luaL_getn` and `luaL_setn` (from the auxiliary library) are deprecated. Use `lua_objlen` instead of `luaL_getn` and nothing instead of `luaL_setn`.
- Function `luaL_openlib` was replaced by `luaL_register`.
- Function `luaL_checkudata` now throws an error when the given value is not a userdata of the expected type. (In Lua 5.0 it returned `NULL`.)

The basic library

Environment and global variables

getfenv ([f]) — If **f** is a function, returns its environment; if **f** is a number, returns the environment of function at level **f** (1 = current [default], 0 = global); if the environment has a field **__fenv**, returns that instead.

setfenv (f, t) — Sets environment for function **f** or function at level **f** (0 = current thread); if the original environment has a field **__fenv**, raises an error.

_G — Variable whose value is the global environment.

_VERSION — Variable containing the interpreter's version, e.g. **"Lua 5.0"**.

Loading and executing

require (pkgname) — Loads a package, raises error if cannot load it, returns **true** if cached.

dofile ([filename]) — Loads and executes the contents of **filename** [default: standard input]; returns its returned values.

loadfile (filename) — Loads the contents of **filename**, do not execute it; returns compiled chunk as function, or **nil** and error message.

loadstring (s [, n]) — Loads the contents of string **s**, do not execute it, set chunk name = **n;** returns compiled chunk as function, or **nil** and error message.

loadlib (lib, func) — Links to the dynamic library named **lib** (e.g. .so or .dll); returns function named **func**, or **nil** and error message.

pcall (f [, *args*]) — Calls function **f** in protected mode; returns **true** and results if OK, else **false** and error message.

xpcall (f, h) — As **pcall ()** but passes error handler **h** instead of extra args; returns as **pcall ()** but with the result of **h ()** as error message, if any (use **debug.traceback ()** from the debug library for extended error info).

Simple output and error feedback

print (*args*) — Prints each of the passed *args* to stdout using **tostring** (see below).

error (msg [, n]) — Terminates the program or the last protected call (e.g. **pcall ()**) with error message **msg** quoting level **n** [default: 1, current function].

assert (v [, msg]) — Calls **error (msg)** if **v** is **nil** or **false** [default **msg**: **"assertion failed!"**].

Information and conversion

type (x) — Returns the type of **x** as a string (e.g. **"nil"**, **"string"**); see *Types*.

tostring (x) — Converts **x** to a string, using **t**'s metatable's **__tostring** if available.

tonumber (x [, b]) — Converts string **x** representing a number in base **b** [2..36, default: 10] to a number, or **nil** if invalid; for base 10 accepts full format (e.g. "1.5e6").

unpack (t) — Returns **t[1]**..**t[n]** (n = **getn(t)**, see *Table library*) as separate values.

Iterators

ipairs (t) — Returns an iterator getting **index, value** pairs of array **t** in num. order.

pairs (t) — Returns an iterator getting **key, value** pairs of table **t** in no order.

next (t [, inx]) — If **inx** is **nil** [default] returns first **index, value** pair of table **t**; if **inx** is the previous index returns next **index, value** pair (**nil** when finished).

Garbage collection

gcinfo () — Returns dynamic memory usage and garbage collector's threshold, in KB.

collectgarbage ([k]) — Sets garbage collector's threshold at **k** KB [default: 0], collects garbage if **k** is below the current dynamic memory usage (always if **k** is 0).

Coroutines

coroutine.create (f) — Creates a new coroutine with Lua function **f** as body, returns it.

coroutine.resume (co, *args*) — Starts or continues running coroutine **co**, passing *args* to it; returns **true** (and possibly values) if **co** calls **coroutine.yield ()** or terminates, returns **false** and a message in case of error.

coroutine.yield (*args*) — Suspends execution of the calling coroutine (not from within C functions, metamethods or iterators), any *args* become extra return values of **coroutine.resume ()**.

coroutine.status (co) — Returns the status of coroutine **co** as a string: either **"running"**, **"suspended"** or **"dead"**.

coroutine.wrap (f) — Creates a new coroutine with Lua function **f** as body and returns a function; this function will act as **coroutine.resume ()** without the first arg and the first return value, propagating any errors.

The table library

Tables as arrays (lists)

table.insert (t, [i,] v) — Inserts **v** at numerical index **i** [default: after the end] in table **t** , increments table size using **table.setn ()**.

table.remove (t [, i]) — Removes element at numerical index **i** [default: last element] from table **t**, decrements table size using **table.setn ()**, returns the removed element or no value on empty table.

table.getn (t) — Returns value of **t.n**, or value previously set by **table.setn ()**, or table size as last consecutive numerical index starting from 1.

table.setn (t, n) — Changes **t.n** if it exists, or sets table size to be returned by **table.getn ()**.

table.sort (t [, cf]) — Sorts (in-place) elements from **t[1]** to **t[table.getn ()]**, using compare function **cf (e1, e2)** [default: '<'].

table.concat (t [, s [, i [, j]]]) — Returns a single string made by concatenating table elements **t[i]** to **t[j]** [default: **i =1**, **j = table.getn ()**] separated by string **s**; returns empty string if no given elements or **i** > **j**.

Iterating on table contents

table.foreach (t, f) — Calls function **f (k, v)** for every field of table **t** in no order, passing key **k** and value **v = t[k]**, stops if **f ()** returns non-**nil**; returns non-**nil** value returned from **f ()**, or no value.

table.foreachi (t, f) — Calls function **f (i, v)** for **i** = **1** to **table.getn ()**, passing index **i** and value **v = t[i]**, stops if **f ()** returns non-**nil**; returns non-**nil** value returned from **f ()**, or no value.

The math library

Basic operations

math.abs (x)	Returns the absolute value of **x**.
math.mod (x, y)	Returns the remainder of **x** / **y** as a rounded-down integer, for **y** ~= 0.
math.floor (x)	Returns **x** rounded down to the nearest integer.
math.ceil (x)	Returns **x** rounded up to the nearest integer.
math.min (*args*)	Returns the minimum value from the *args* received.
math.max (*args*)	Returns the maximum value from the *args* received.

Exponential and logarithmic

math.sqrt (x)	Returns the square root of **x**, for **x** >= 0.
math.pow (x, y)	Returns **x** raised to the power of **y**, i.e. **x^y**; if **x** < 0, **y** must be integer.
__pow (x, y)	Global function added by the math library to make operator '^' work.
math.exp (x)	Returns e (base of natural logs) raised to the power of **x**, i.e. e^**x**.
math.log (x)	Returns the natural logarithm of **x**, for **x** >= 0.
math.log10 (x)	Returns the base-10 logarithm of **x**, for **x** >= 0.

Trigonometrical

math.deg (a)	Converts angle **a** from radians to degrees.
math.rad (a)	Converts angle **a** from degrees to radians.
math.pi	Constant containing the value of PI.
math.sin (a)	Returns the sine of angle **a** (measured in radians).
math.cos (a)	Returns the cosine of angle **a** (measured in radians).
math.tan (a)	Returns the tangent of angle **a** (measured in radians).
math.asin (x)	Returns the arc sine of **x** in radians, for **x** in [-1, 1].
math.acos (x)	Returns the arc cosine of **x** in radians, for **x** in [-1, 1].
math.atan (x)	Returns the arc tangent of **x** in radians.
math.atan2 (y, x)	Similar to **math.atan (y / x)** but with quadrant and allowing x = 0.

Splitting on powers of 2

math.frexp (x)	Splits **x** into normalized fraction and exponent of 2, returns both.
math.ldexp (x, y)	Returns **x** * (2 ^ **y**) with **x** = normalized fraction, **y** = exponent of **2**.

Pseudo-random numbers

math.random ([n [, m])	Returns a pseudo-random number in range [0, 1) if no arguments, in range [1, **n**] if **n** is given, in range [**n**, **m**] if both args are passed.
math.randomseed (n)	Sets a seed **n** for random sequence (same seed = same sequence).

The string library

Basic operations

string.len (s)	Returns the length of string **s**, including embedded zeros.
string.sub (s, i [, j])	Returns the substring of **s** from position **i** to **j** [default: -1] inclusive.
string.rep (s, n)	Returns a string made of **n** concatenated copies of string **s**.
string.upper (s)	Returns a copy of **s** converted to uppercase according to locale.
string.lower (s)	Returns a copy of **s** converted to lowercase according to locale.

Character codes

string.byte (s [, i])	Returns the platform-dependent numerical code (e.g. ASCII) of character at position **i** [default: 1] in string **s**, or **nil** if invalid **i**.
string.char (*args*)	Returns a string made of the characters whose platform-dependent numerical codes are passed as *args*.

Formatting

string.format (s [, *args*])	Returns a copy of **s** where formatting directives beginning with '**%**' are replaced by the value of arguments *args*, in the same order. (see *Formatting directives* below)

Finding, replacing, iterating

string.find (s, p [, i [, d]])	Returns first and last position of pattern **p** in string **s**, or **nil** if not found, starting search at position **i** [default: 1]; returns parenthesized 'captures' as extra results. If **d** is true, treat pattern as plain string. (see *Patterns* below)
string.gfind (s, p)	Returns an iterator getting next occurrence of pattern **p** (or its captures) in string **s** as substring(s) matching the pattern. (see *Patterns* below)
string.gsub (s, p, r [, n])	Returns a copy of **s** with up to **n** [default: 1] occurrences of pattern **p** (or its captures) replaced by **r** if **r** is a string (**r** can include references to captures in the form **%***n*), or by calling **r ()** if it is a function: **r ()** will receive captured substrings and should return the replacement string; returns as second result the number of substitutions made. (see *Patterns* below)

Function storage

string.dump (f)	Returns a binary representation of function **f**, for later use with **loadstring ()**. **f** must be a Lua function with no upvalues.

Note — String indexes go from 1 to string.**len (s)**, from end of string if negative (index -1 refers to the last character).

```
function string.match(s1,s2) -- Lua 5.1
  local _,_,s = string.find(s1,s2) -- Lua 5.0
  return s
end
```

Formatting directives for string.format

% [*flags*] [*field_width*] [**.***precision*] *type*

Formatting field types

%d	Decimal integer.
%o	Octal integer.
%x	Hexadecimal integer, uppercase if **%X**.
%f	Floating-point in the form [-]nnnn.nnnn.
%e	Floating-point in exp. form [-]n.nnnn e [+|-]nnn, uppercase if **%E**.
%g	Floating-point as **%e** if exp. < -4 or >= precision, else as **%f**; uppercase if **%G**.
%c	Character having the (system-dependent) code passed as integer.
%s	String with no embedded zeros.
%q	String between double quotes, with all special characters escaped.
%%	The '**%**' character.

Formatting flags

-	Left-justifies in **field_width** [default: right-justify].
+	Prepends sign (applies to numbers).
(**space**)	Prepends sign if negative, else blank space.
#	Adds "0x" before **%x**, force decimal pt. for **%e**, **%f**, leaves trailing zeros for **%g**.

Formatting field width

n	Puts at least **n** characters, pad with blanks.
0n	Puts at least **n** characters, left-pad with zeros

Formatting precision

.n	Puts at least **n** digits for integers; rounsd to **n** decimals for floating-point; puts no more than **n** characters for strings.

Formatting examples

string.format ("results: %d, %d", 13, 27)	**results: 13, 27**
string.format ("<%5d>", 13)	**< 13>**
string.format ("<%-5d>", 13)	**<13 >**
string.format ("<%05d>", 13)	**<00013>**
string.format ("<%06.3d>", 13)	**< 013>**
string.format ("<%f>", math.pi)	**<3.141593>**
string.format ("<%e>", math.pi)	**<3.141593e+00>**
string.format ("<%.4f>", math.pi)	**<3.1416>**
string.format ("<%9.4f>", math.pi)	**< 3.1416>**
string.format ("<%c>", 64)	**<@>**
string.format ("<%s.4>", "goodbye")	**<good>**
string.format("%q", [[she said "hi"]])	**"she said \"hi\""**

Patterns and pattern items

General pattern format: *pattern_item* [*pattern_items*]

cc	Matches a single character in the class *cc* (see *Pattern character classes* below).
*cc******	Matches zero or more characters in the class *cc*; matchest longest sequence.
*cc***-**	Matches zero or more characters in the class *cc*; matchest shortest sequence.
*cc***+**	Matches one or more characters in the class *cc*; matchest longest sequence.
*cc***?**	Matches zero or one character in the class *cc*.
%*n*	(*n* = 1..9) Matches the *n*-th captured string (see *Pattern captures*).
%b*xy*	Matches the balanced string from character *x* to character *y* (e.g. nested parenthesis).
^	Anchor spattern to start of string, must be the first item in the pattern.
$	Anchor spattern to end of string, must be the last item in the pattern.

Pattern captures

(*sub_pattern*)	Stores substring matching *sub_pattern* as capture **%1**..**%9**, in order.
()	Stores current string position as capture **%1**..**%9**, in order.

Pattern character classes

.	Any character.		
%a	Any letter.	**%A**	Any non-letter.
%c	Any control character.	**%C**	Any non-control character.
%d	Any digit.	**%D**	Any non-digit.
%l	Any lowercase letter.	**%L**	Any non-(lowercase letter).
%p	Any punctuation character	**%P**	Any non-punctuation character
%s	Any whitespace character.	**%S**	Any non-whitespace character.
%u	Any uppercase letter.	**%U**	Any non-(uppercase letter).
%w	Any alphanumeric character.	**%W**	Any non-alphanumeric character.
%x	Any hexadecimal digit.	**%X**	Any non-(hexadecimal digit).
%z	The zero character.	**%Z**	Any non-zero character.
%*x*	(*x* = symbol) The symbol itself.		
x	If *x* not in ^$()%.[]*+-? the character itself.		
[*set* **]**	Any character in any of the given classes, can also be a range **[***c1***-***c2***]**.	**[^***set* **]**	Any character not in *set*.

Pattern examples

string.find("Lua is great!", "is")	**5**	**6**
string.find("Lua is great!", "%s")	**4**	**4**
string.gsub("Lua is great!", "%s", "-")	**Lua-is-great!**	**2**
string.gsub("Lua is great!", "[%s%l]", "*")	**L**********!**	**11**
string.gsub("Lua is great!", "%a+", "*")	*** * *!**	**3**
string.gsub("Lua is great!", "(.)", "%1%1")	**LLuuaa iiss ggrreeaatt!!**	**13**
string.gsub("Lua is great!", "%but", "")	**L!**	**1**
string.gsub("Lua is great!", "^.-a", "LUA")	**LUA is great!**	**1**
string.gsub("Lua is great!", "^.-a", function (s) return string.upper(s) end)	**LUA is great!**	**1**

The I/O library

Complete I/O

io.open (fn [, m])	Opens file with name **fn** in mode **m**: "r" = read [default], "w" = write", "a" = append, "r+" = update-preserve, "w+" = update-erase, "a+" = update-append (add trailing "b" for binary mode on some systems), returns a file object (an userdata with a C handle) usable with **':'** syntax.
file**:close** ()	Closes **file**.
file**:read** (*formats*)	Returns a value from **file** for each of the passed *formats*: "*n" = reads a number, "*a" = reads the whole **file** as a string from current position ("" at end of file), "*l" = reads a line (**nil** at end of file) [default], *n* = reads a string of up to *n* characters (**nil** at end of file).
file**:lines** ()	Returns an iterator function reading line-by-line from **file**; the iterator does not close the file when finished.
file**:write** (*values*)	Write each of the *values* (strings or numbers) to **file**, with no added separators. Numbers are written as text, strings can contain binary data (in this case, **file** may need to be opened in binary mode on some systems).
file**:seek** ([p] [, of])	Sets the current position in **file** relative to **p** ("set" = start of file [default], "cur" = current, "end" = end of file) adding offset **of** [default: zero]; returns the new current position in **file**.
file**:flush** ()	Writes to **file** any data still held in memory buffers.

Simple I/O

io.input ([file])	Sets **file** as default input file; **file** can be either an open file object or a file name; in the latter case the file is opened for reading in text mode; returns a file object, the current one if no **file** given; raises error on failure.
io.output ([file])	Sets **file** as default output file (the current output file is not closed); **file** can be either an open file object or a file name; in the latter case the file is opened for writing in text mode; returns a file object, the current one if no **file** given; raises error on failure.
io.close ([file])	Closes **file** (a file object) [default: closes the default output file].
io.read (*formats*)	Reads from the default input file, same usage as **file:read ()** above.
io.lines ([fn])	Opens the file with name **fn** for reading and returns an iterator function reading from it line-by-line; the iterator closes the file when finished; if no **fn** given, returns an iterator reading lines from the default input file.
io.write (*values*)	Writes to the default output file, same usage as **file:write ()** above.
io.flush ()	Writes to the default output file any data still held in memory buffers.

Standard files and utility functions

io.stdin	Predefined input file object.
io.stdout	Predefined output file object.
io.stderr	Predefined error output file object.
io.type (x)	Returns the string "file" if **x** is an open file, "closed file" if **x** is a closed file, **nil** if **x** is not a file object.
io.tmpfile ()	Returns a file object for a temporary file (deleted when program ends).

Note: the I/O functions return **nil** and an error message on failure, unless otherwise stated; passing a closed file object raises an error instead.

The operating sytem library

Date/time

os.clock ()	Returns the approximated CPU time from the start of the program, in seconds (measurement criteria may vary between systems).
os.time ([tt])	Returns a system-dependent number representing date/time described by table **tt** [default: current]. **tt** must have fields **year**, **month**, **day**; can have fields **hour**, **min**, **sec**, **isdst** (daylight saving, boolean). On many systems the returned value is a number of seconds from a fixed date/time.
os.date ([fmt [, t]])	Returns a table or a string describing date/time **t**, that should be a value returned by **os.time ()** [default: current date/time], according to the format string **fmt** [default: default: date/time according to locale settings]; if **fmt** is "*t" or "!*t", returns a table with fields **year** (nnnn), **month** (1..12), **day** (1..31), **hour** (0..23), **min** (0..59), **sec** (0..61), **wday** (1..7, Sunday = 1), **yday** (1..366), **isdst** (true = daylight saving), else returns the **fmt** string with time formatting directives beginning with '**%**' replaced according to *Time formatting directives* (see below); in either case a leading "!" requests UTC (Coordinated Universal Time).
os.difftime (t2, t1)	Returns the difference between two values returned by **os.time ()**.

System interaction

os.execute (cmd)	Calls a system shell to execute the string **cmd** as a command; returns a system-dependent status code.
os.exit ([code])	Terminates the program returning **code** [default: success].
os.getenv (var)	Returns a string with the value of the environment variable named **var**, or **nil** if no such variable exists.
os.setlocale (s [, c])	Sets the locale described by string **s** for category **c**: "all", "collate", "ctype", "monetary", "numeric" or "time" [default: "all"]; returns the name of the new locale, or **nil** if it cannot be set.
os.remove (fn)	Deletes the file named **fn**; in case of error returns **nil** and error description.
os.rename (of, nf)	Renames file **of** to **nf** ; in case of error returns **nil** and error description.
os.tmpname ()	Returns a string usable as name for a temporary file; subject to possible name conflicts, use **io.tmpfile ()** instead.

Time formatting directives (most used, portable features):

%c	Date/time (locale).		
%x	Date only (locale).	**%X**	Time only (locale).
%y	Year (nn).	**%Y**	Year(nnnn).
%j	Day of year (001..366).		
%m	Month (01..12).		
%b	Abbreviated month name (locale).	**%B**	Full month name (locale).
%d	Day of month (01..31).		
%U	Week number (01..53), Sunday-based.	**%W**	Week number (01..53) Monday-based.
%w	Weekday (0..6), 0 is Sunday.		
%a	Abbreviated weekday name (locale).	**%A**	Full weekday name (locale).
%H	Hour (00..23).	**%I**	Hour (01..12).
%p	Either AM or PM.		
%M	Minute (00..59).		
%S	Second (00..61).		
%Z	Time zone name, if any.		

The debug library

Basic functions

debug.debug ()	Enters interactive debugging shell (type **cont** to exit);local variables cannot be accessed directly.
debug.getinfo (f [, w])	Returns a table with information for function **f** or for function at level **f** [1 = caller], or **nil** if invalid level (see *Result fields for getinfo* below); characters in string **w** select one or more groups of fields [default: all] (see *Options for getinfo* below).
debug.getlocal (n, i)	Returns name and value of local variable at index **i** (from 1, in order of appearance) of the function at stack level **n** (1= caller); returns **nil** if **i** is out of range, raises error if **n** is out of range.
debug.getupvalue (f, i)	Returns name and value of upvalue at index **i** (from 1, in order of appearance) of function **f**; returns **nil** if **i** is out of range.
debug.traceback ([msg])	Returns a string with traceback of call stack, prepended by **msg**.

Changing hidden values

debug.setlocal (n, i, v)	Assigns value **v** to the local variable at index **i** (from 1, in order of appearance) of the function at stack level **n** (1= caller); returns **nil** if **i** is out of range, raises error if **n** is out of range.
debug.setupvalue (f, i, v)	Assigns value **v** to the upvalue at index **i** (from 1, in order of appearance) of function **f**; returns **nil** if **i** is out of range.

Hooks

debug.sethook ([h, m [, n]])	Sets function **h** as hook, called for events given in string (mask) **m**: "c" = function call, "r" = function return, "l" = new code line; also, a number **n** will call **h ()** every **n** instructions; **h ()** will receive the event type as first argument: "call", "return", "tail return", "line" (line number as second argument) or "count"; use **debug.getinfo (2)** inside **h ()** for info (not for "tail_return").
debug.gethook ()	Returns current hook function, mask and count set with **debug.sethook ()**.

Note: the debug library functions are inefficient and should not be used in normal operation.

Result fields for debug.getinfo

source	Name of file (prefixed by '@') or string where the function was defined.
short_src	Short version of **source**, up to 60 characters.
linedefined	Line of source where the function was defined.
what	"Lua" = Lua function, "C" = C function, "main" = part of main chunk.
name	Name of function, if available, or a reasonable guess if possible.
namewhat	Meaning of **name**: "global", "local", "method", "field" or "".
nups	Number of upvalues of the function.
func	The function itself.

Options for debug.getinfo (characters for argument w)

n	Returns fields name and **namewhat**.
f	Returns field **func**.
S	Returns fields **source**, **short_src**, **what** and **linedefined**.
l	Returns field **currentline**.
u	Returns field **nup**.

The stand-alone interpreter

Command line syntax

lua [*options*] [*script* [*arguments*]]

Options

-	Loads and executes **script** from standard input (no args allowed).
-e *stats*	Executes the Lua statements contained in the literal string *stats*, can be used multiple times on the same line.
-l *filename*	Requires *filename* (loads and executes it if not already done).
-i	Enters interactive mode after loading and executiong *script*.
-v	Prints version information.
--	Stops parsing options.

Recognized environment variables

LUA_INIT	If it contains a string in the form *@filename* loads and executes *filename*, else executes the string itself.
_PROMPT	Sets the prompt for interactive mode.

Special Lua variables

arg	**nil** if no arguments on the command line, else a table containing command line *arguments* starting from **arg[1]** while **arg.n** is the number of *arguments*; **arg [0]** holds the script name as given on the command line; **arg[-1]** and lower indexes contain the fields of the command line preceding the script name.
_PROMPT	Contains the prompt for interactive mode; can be changed by assigning to it.

The compiler

Command line syntax

luac [*options*] [*scripts*]

Options

-	Compiles from standard input.
-l	Produces a listing of the compiled bytecode.
-o *filename*	Sends output to **filename** [default: **luac.out**].
-p	Performs syntax and integrity checking only, does not output bytecode.
-s	Strips debug information; line numbers and local names are lost.
-v	Prints version information.
--	Stops parsing options.

Note: compiled chunks are portable on machines having the same word size.

Acknowledgments

I am grateful to all people that contributed with notes and suggestions, including John Belmonte, Albert-Jan Brouwer, Tiago Dionizio, Marius Gheorghe, Asko Kauppi, Philippe Lhoste, Virgil Smith, Ando Sonenblick, Nick Trout and of course Roberto Ierusalimschy, whose "Lua 5.0 Reference Manual" and " Programming in Lua" have been my main sources of Lua lore.