Manual de Referencia de Lua 5.1

por Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes

(traducción de Julio Manuel Fernández-Díaz; véanse las notas sobre la misma al final del documento.)

contenido · índice · english · português · español · deutsch

1 – Introducción

Lua es un lenguage de programación extensible diseñado para una programación procedimental general con utilidades para la descripción de datos. También ofrece un buen soporte para la programación orientada a objetos, programación funcional y programación orientada a datos. Se pretende que Lua sea usado como un lenguaje de *script* potente y ligero para cualquier programa que lo necesite. Lua está implementado como una biblioteca escrita en C *limpio* (esto es, en el subconjunto común de ANSI C y C++).

Siendo un lenguaje de extensión, Lua no tiene noción de programa principal (*main*): sólo funciona *embebido* en un cliente anfitrión, denominado *programa contenedor* o simplemente anfitrión (*host*). Éste puede invocar funciones para ejecutar un trozo de código Lua, puede escribir y leer variables de Lua y puede registrar funciones C para que sean llamadas por el código Lua. A través del uso de funciones C, Lua puede ser aumentado para abarcar un amplio rango de diferentes dominios, creando entonces lenguajes de programación personalizados que comparten el mismo marco sintáctico. La distribución de Lua incluye un programa anfitrión de muestra denominado `lua`, que usa la biblioteca de Lua para ofrecer un intérprete de Lua completo e independiente.

Lua es software libre, y se proporciona, como es usual, sin garantías, como se establece en su licencia. La implementación descrita en este manual está disponible en el sitio web oficial de Lua, `www.lua.org`.

Como cualquier otro manual de referencia, este documento es parco en algunos lugares. Para una discusión de las decisiones detrás del diseño de Lua, véanse los artículos técnicos disponibles en el sitio web de Lua. Para una detallada introducción a la programación en Lua, véase el libro de Roberto, *Programming in Lua (Second Edition)*.

2 – El lenguaje

Esta sección describe el léxico, la sintaxis y la semántica de Lua. En otras palabras, esta sección describe qué elementos (*tokens*) son válidos, cómo deben combinarse y qué significa su combinación.

Las construcciones del lenguaje se explicarán usando la notación BNF extendida usual, en la que {*a*} significa 0 o más *a*es, y [*a*] significa una *a* opcional. Los símbolos no terminales se muestran en *itálica*, las palabras clave (*keywords*) se muestran en **negrita**, y los otros símbolos terminales se muestran en un tipo de letra de paso fijo (`typewriter`), encerrada entre comillas simples. La sintaxis completa de Lua se encuentra al final de este manual.

2.1 – Convecciones léxicas

Los *nombres* (también llamados *identificadores*) en Lua pueden ser cualquier tira de caracteres (*string*) sólo con letras, dígitos y caracteres de subrayado (*underscore*), no comenzando por un dígito. Esto coincide con la definición de los nombres en la mayoría de los lenguajes. (La definición de letra depende de la implementación local actual a través del sistema *locale*: cualquier carácter considerado alfabético en el sistema local puede ser usado en un identificador.) Los identificadores se usan para nombrar variables y campos de tablas.

Las siguientes *palabras clave* (*keywords*) están reservadas y no pueden usarse como nombres:

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

En Lua las letras mayúsculas y las minúsculas se consideran diferentes: `and` es una palabra reservada, pero `And` y `AND` son dos nombres diferentes válidos. Como convención, los nombres que comienzan por un subrayado seguido por letras en mayúsculas (como `_VERSION`) están reservados para uso como variables globales internas de Lua.

Los siguientes *strings* denotan otros elementos:

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

Los *strings literales* pueden ser delimitados por comillas simples (apóstrofes) o dobles, y pueden contener las siguientes secuencias de *escape* de C: '`\a`' (pitido, *bell*) '`\b`' (retroceso, *backspace*), '`\f`' (salto de página, *form feed*), '`\n`' (nueva línea, *newline*), '`\r`' (retorno de carro, *carriage return*), '`\t`' (tabulador horizontal, *horizontal tab*), '`\v`' (tabulador vertical, *vertical tab*), '`\\`' (barra inversa, *backslash*), '`\"`' (comilla doble, *quotation mark* o *double quote*) y '`\'`' (apóstrofe, *apostrophe* o *single quote*). Además, una '`\`*newline*' (esto es, una barra inversa seguida por un salto de línea real) produce un salto de línea en el *string*. Un carácter en un *string* puede también especificarse por su valor numérico usando la secuencia de escape '`\`*ddd*', donde *ddd* es una secuencia de tres dígitos decimales. (Tenga presente que si la secuencia numérica de escape está seguida de un dígito debe ser expresada usando exactamente tres dígitos.) Los *strings* en Lua pueden contener cualquier valor de 8 bits, incluyendo el carácter cero, el cual puede ser especificado mediante '`\0`'.

Para poner una comilla (simple) doble, una barra inversa, un retorno de carro o un carácter cero dentro de un *string* literal encerrado por comillas (simples) dobles se debe usar una secuencia de

escape. Cualquier otro carácter puede ser incluido en el literal. (Algunos caracteres de control pueden causar problemas con el sistema de ficheros, pero Lua no tiene problemas con ellos.)

Los *strings* literales pueden definirse usando un formato largo, encerrados en *corchetes largos*. Definimos un *corchete largo de abrir de nivel n* como un corchete de abrir seguido de *n* signos igual (=) seguidos de otro corchete de abrir. Así, un corchete largo de abrir de nivel 0 se escribe `[[`, un corchete largo de abrir de nivel 1 se escribe `[=[`, y así sucesivamente. Los *corchetes largos de cerrar* se define de manera similar; por ejemplo, un corchete largo de cerrar de nivel 4 se expresa `]====]`. Un *string* largo comienza en un corchete largo de abrir de cualquier nivel y termina en el primer corchete largo de cerrar del mismo nivel. Los *strings* literales delimitados de esta manera pueden extenderse por varias líneas, las secuencias de escape no son interpretadas y se ignoran los corchetes largos de cualquier otro nivel. Por tanto, pueden contener cualquier cosa excepto un corchete de cerrar del mismo nivel o caracteres cero.

Por conveniencia, cuando un corchete largo de abrir es seguido inmediatamente de un carácter de nueva línea, éste no es incluido en el *string*. Por ejemplo, usando el código de caracteres ASCII (en el cual 'a' se codifica como 97, el carácter de nueva línea se codifica como 10, y '1' se codifica como 49), los cinco literales siguientes denotan el mismo *string*:

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

Las *constantes numéricas* pueden contener una parte decimal opcional y también un exponente opcional. Lua también acepta constantes enteras hexadecimales, escritas anteponiendo el prefijo `0x`. Algunos ejemplos de constantes numéricas válidas son

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

Los *comentarios* comienzan con un doble guión (`--`) en cualquier lugar fuera de un *string*. Si el texto que sigue inmediatamente después de `--` no es un corchete largo de abrir el comentario es *corto* y llega hasta el final de línea. En otro caso tenemos un comentario *largo*, que alcanza hasta el correspondiente corchete largo de cerrar. Los comentarios largos se usan frecuentemente para deshabilitar temporalmente trozos de código.

2.2 – Valores y tipos

Lua es un *lenguaje dinámicamente tipado*. Esto significa que las variables no tienen tipos; sólo tienen tipo los valores. No existen definiciones de tipo en el lenguaje. Todos los valores almacenan su propio tipo.

Todos los valores en Lua son *valores de primera clase*. Esto significa que todos ellos pueden ser almacenados en variables, pueden ser pasados como argumentos de funciones, y también ser devueltos como resultados.

Existen ocho tipos básicos en Lua: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread* y *table*. *Nil* es el tipo del valor **nil**, cuya principal propiedad es ser diferente de cualquier otro valor; normalmente representa la ausencia de un valor útil. *Boolean* es el tipo de los valores **false** (falso)

y **true** (verdadero). Tanto **nil** como **false** hacen una condición falsa; cualquier otro valor la hace verdadera. *Number* representa números reales (en coma flotante y doble precision). (Es fácil construir intérpretes de Lua que usen otra representación interna para los números, ya sea en coma flotante con precisión simple o enteros largos. Véase el fichero `luaconf.h`.) *String* representa una tira de caracteres. Lua trabaja con 8 bits: los *strings* pueden contener cualquier carácter de 8 bits, incluyendo el carácter cero ('`\0`') (véase §2.1).

Lua puede llamar (y manejar) funciones escritas en Lua y funciones escritas en C (véase §2.5.8).

El tipo *userdata* se incluye para permitir guardar en variables de Lua datos arbitrarios en C. Este tipo corresponde a bloques de memoria y no tienen asociadas operaciones predefinidas en Lua, excepto la asignación y el test de identidad. Sin embargo, usando §*metatablas*, el programador puede definir operaciones asociadas a valores de tipo *userdata* (véase §2.8). Los valores de este tipo no pueden ser creados o modificados en Lua, sino sólo a través de la API de C. Esto garantiza la integridad de los datos propiedad del programa anfitrión.

El tipo *thread* representa procesos de ejecución y es usado para implementar co-rutinas (véase §2.11). No deben confundirse los procesos de Lua con los del sistema operativo. Lua soporta co-rutinas en todos los sistemas, incluso en aquéllos que no soporten procesos.

El tipo *table* (tabla) implementa *arrays* asociativos, esto es, *arrays* que pueden ser indexados no sólo con números, sino también con cualquier valor (excepto **nil**). Las tablas pueden ser *heterogéneas*, ya que pueden contener valores de todos los tipos (excepto **nil**). Las tablas son el único mecanismo de estructuración de datos en Lua; pueden ser usadas para representar *arrays* ordinarios, tablas de símbolos, conjuntos, registros, grafos, árboles, etc. Para representar registros Lua usa el nombre del campo como índice. El lenguaje soporta esta representación haciendo la notación `b.nombre` equivalente a `b["nombre"]`. Existen varias maneras convenientes de crear tablas en Lua (véase §2.5.7).

Como índices, también los valores de los campos de una tabla pueden ser de cualquier tipo (excepto **nil**). En particular, debido a que las funciones son valores de primera clase, los campos de las tablas pueden contener funciones. Entonces las tablas pueden contener también *métodos* (véase §2.5.9).

Los valores de las tablas, las funciones, los procesos y los *userdata* (completos) son *objetos*: las variables no *contienen* realmente esos valores, sino que sólo los *referencian*. La asignación, el paso de argumentos y el retorno de las funciones siempre manejan referencias a esos valores; esas operaciones no implican ningún tipo de copia.

La función de biblioteca `type` retorna un *string* que describe el tipo de un valor dado.

2.2.1 – Coerción

Lua puede convertir automáticamente entre valores *string* y valores numéricos en tiempo de ejecución. Cualquier operación aritmética aplicada a un *string* intenta convertir el mismo en un número, siguiendo las reglas normales de conversión. Y viceversa, cuando un número se usa donde se espera un *string* el número se convierte a *string*, con un formato razonable. Para un control completo en la conversión de números en *strings* debe usarse la función `format` de la biblioteca de manejo de *strings* (véase `string.format`).

2.3 – Variables

Las variables son lugares donde se almacenan valores. Existen tres tipos de variables en Lua: globales, locales y campos de tabla.

Un único nombre puede denotar una variable local o una global (o un argumento formal de una función, el cual es una forma particular de una variable local):

```
var ::= nombre
```

`nombre` denota identificadores, como se definen en §2.1.

Lua asume que las variables son globales, a no ser que sean declaradas explícitamente como locales (véase §2.4.7). Las variables locales tienen un ámbito (*scope*) definido *léxicamente*: pueden ser accedidas libremente desde dentro de las funciones definidas en su mismo ámbito (véase §2.6).

Antes de la primera asignación el valor de una variable es **nil**.

Los corchetes se usan para indexar una tabla:

```
var ::= prefixexp '[' exp ']'
```

La primera expresión (*prefixexp*) debe dar como resultado un valor tabla; la segunda expresión (*exp*) identifica una entrada específica en esta tabla. La expresión que denota la tabla que es indexada tienen una sintaxis restringida; véase §2.5 para más detalles.

La sintaxis `var.nombre` es otra manera de expresar `var["nombre"]` y se usa para denotar campos de tablas:

```
var ::= prefixexp '.' nombre
```

La manera en qué se accede a las variables globales y a los campos de las tablas puede ser cambiada mediante metatablas. Un acceso a la variable indexada `t[i]` equivale a una llamada a `gettable_event(t,i)` (véase §2.8 para una completa descripción de la función `gettable_event`. Esta función no está definida ni es invocable desde Lua. Se usa aquí sólo con propósitos ilustrativos).

Todas las variables globales se almacenan como campos de tablas ordinarias en Lua, denominadas *tablas de entorno* o simplemente *entornos* (véase §2.9). Cada función tiene su propia referencia a un entorno, así que todas las variables globales de esta función se refieren a esa tabla de entorno. Cuando se crea una función, ésta hereda el entorno de la función que la creó. Para obtener la tabla de entorno de una función en código Lua, se invoca a `getfenv`. Para reemplazarla se llama a `setfenv`. (Se pueden manejar los entornos de una función C, pero sólo a través de la biblioteca de depuración; véase §5.9.)

Un acceso a la variable global `x` equivale a `_env.x`, que a su vez equivale a

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

donde `_env` es el entorno de la función que se está ejecutando en ese momento (véase §2.8 para una completa descripción de la función `gettable_event`. Esta función no está definida ni es invocable desde Lua. Igualmente, la variable `_env` no está definida en Lua. Se usan aquí sólo con propósitos ilustrativos.)

2.4 – Sentencias

Lua soporta un conjunto casi convencional de sentencias, similar a los de Pascal o C. Este conjunto incluye la asignación, estructuras de control de flujo, llamadas a funciones, constructores de tablas y declaraciones de variables.

2.4.1 – *Chunks*

La unidad de ejecución en Lua se denomina *chunk*, el cual es simplemente un conjunto de sentencias que se ejecutan secuencialmente. Cada sentencia puede llevar opcionalmente al final un punto y coma:

```
chunk ::= {sentencia [';']}
```

No existen sentencias vacías en Lua y por tanto '`;;`' no es legal.

Lua maneja cada *chunk* como el cuerpo de una función anónima con un número variable de argumentos (véase §2.5.9). Los *chunks* pueden definir variables locales, recibir argumentos y retornar valores.

Un *chunk* puede ser almacenado en un fichero o en un *string* dentro de un programa anfitrión. Cuando se ejecuta un *chunk* primero se precompila, creándose instrucciones para una máquina virtual, y es entonces cuando el código compilado es ejecutado por un intérprete de la máquina virtual.

Los *chunks* pueden también estar precompilados en forma binaria; véase el programa `luac` para más detalles. Las formas fuente y compilada de los programas son intercambiables; Lua detecta automáticamente el tipo de fichero y actúa de manera acorde.

2.4.2 – Bloques

Un bloque es una lista de sentencias; sintácticamente un bloque es lo mismo que un *chunk*:

```
bloque ::= chunk
```

Un bloque puede ser delimitado explícitamente para producir una sentencia simple:

```
sentencia ::= do bloque end
```

Los bloques explícitos son útiles para controlar el ámbito de las declaraciones de variable. También se utilizan a veces para añadir sentencias **return** o **break** en medio de otro bloque (véase §2.4.4).

2.4.3 – La asignación

Lua permite asignaciones múltiples. Por tanto la sintaxis de una asignación define una lista de variables a la izquierda y una lista de expresiones a la derecha. Los elementos de ambas listas están separados por comas:

```
sentencia ::= varlist '=' explist
varlist ::= var {',' var}
explist ::= exp {',' exp}
```

Las expresiones se analizan en §2.5.

Antes de una asignación la lista de expresiones se *ajusta* a la longitud de la lista de variables. Si existen más valores de los necesarios el exceso se descarta. Si existen menos valores de los

necesarios la lista se extiende con tantos valores **nil** como se necesiten. Si la lista de expresiones finaliza con una llamada a una función entonces todos los valores devueltos en la llamada pueden entrar en la lista de valores antes del ajuste (excepto cuando se encierra entre paréntesis; véase §2.5).

La sentencia de asignación primero evalúa todas sus expresiones y sólo después se hace la asignación. Entonces, el código

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

asigna 20 a `b[3]`, sin afectar a `b[4]` debido a que `i` en `b[i]` se evalúa (a 3) antes de que se le asigne el valor 4. Similarmente, la línea

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

intercambia los valores de `x` e `y`.

El mecanismo de asignación a las variables globales y a los campos de tablas puede ser modificado mediante metatablas. Una asignación a una variable indexada `t[i] = val` equivale a `settable_event(t,i,val)`. (Véase §2.8 para una completa descripción de la función `settable_event`. Esta función no está definida ni es invocable desde Lua. Se usa sólo con propósitos ilustrativos.)

Una asignación a la variable global `x = val` equivale a la asignación `_env.x = val`, que a su vez equivalen a

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

donde `_env` es el entorno de la función que está ejecutándose en ese momento. (La variable `_env` no está definida en Lua. Se utiliza aquí sólo con propósitos ilustrativos.)

2.4.4 – Estructuras de control

Las estructuras de control **if**, **while** y **repeat** tienen el significado habitual y la sintaxis familiar:

```
sentencia ::= while exp do bloque end
sentencia ::= repeat bloque until exp
sentencia ::= if exp then bloque {elseif exp then bloque} [else bloque] end
```

Lua tiene también una sentencia **for**, en dos formatos (véase §2.4.5).

La condición de una expresión de una estructura de control puede retornar cualquier valor. Tanto **false** como **nil** se consideran falsos. Todos los valores diferentes de **nil** y **false** se consideran verdaderos (en particular, el número 0 y el *string* vacío son también verdaderos).

En el bucle **repeat**–**until** el bloque interno no acaba en la palabra clave **until** sino detrás de la condición. De esta manera la condición puede referirse a variables locales declaradas dentro del bloque del bucle.

La orden **return** se usa para devolver valores desde una función o un *chunk* (el cual es justamente una función). Las funciones y los *chunks* pueden retornar más de un valor, por lo que la sintaxis para **return** es

```
sentencia ::= return [explist]
```

La orden **break** se usa para terminar la ejecución de los bucles **while**, **repeat** y **for**, saltando a la sentencia que sigue después del bucle:

```
sentencia ::= break
```

Un **break** finaliza el bucle más interno que esté activo.

Las órdenes **return** y **break** pueden aparecer sólo como *última* sentencia dentro de un bloque. Si se necesita realmente un **return** o un **break** en medio de un bloque se debe usar un bloque más interno explícitamente, como en '`do return end`' y '`do break end`', debido a que así **return** y **break** son las últimas sentencias en su propio bloque.

2.4.5 – La sentencia for

La sentencia **for** tiene dos formas: una numérica y otra genérica.

La forma numérica del bucle **for** repite un bloque mientras una variable de control sigue una progresión aritmética. Tiene la sintaxis siguiente:

```
sentencia ::= for nombre '=' exp1 ',' exp2 [',' exp3] do bloque end
```

El *bloque* se repite para los valores de *nombre* comenzando en *exp1* hasta que sobrepasa *exp2* usando como paso *exp3*. Más precisamente una sentencia **for** como

```
for v = e1, e2, e3 do bloque end
```

equivale al código:

```
do
  local var, limit, step = tonumber(e1), tonumber(e2), tonumber(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
    bloque
    var = var + step
  end
end
```

Nótese lo siguiente:

- Todas las expresiones de control se evalúan sólo una vez, antes de que comience el bucle. Deben resultar todas en números.
- *var*, *limit* y *step* son variables invisibles. Los nombres aparecen aquí sólo con propósitos ilustrativos.
- Si la tercera expresión (el paso) está ausente se utiliza paso 1.
- Se puede utilizar **break** para salir del bucle **for**.
- La variable de control `v` es local dentro del bucle; no se puede utilizar su valor después de que finalice el bucle **for** o después de una salida del mismo con **break**. Si se necesita el valor de la variable `var` entonces debe asignarse a otra variable antes del **break** o de la salida del bucle.

La sentencia **for** genérica trabaja con funciones, denominadas *iteradores*. En cada iteración se invoca a la función iterador que produce un nuevo valor, parándose la iteración cuando el nuevo valor es **nil**. El bucle **for** genérico tiene la siguiente sintaxis:

```
sentencia ::= for lista_de_nombres in explist do bloque end
lista_de_nombres ::= nombre {',' nombre}
```

Una sentencia **for** como

```
for var_1, ..., var_n in explist do bloque end
```

equivale al código:

```
do
  local f, s, var = explist
  while true do
    local var_1, ... , var_n = f(s, var)
    var = var_1
    if var == nil then break end
    bloque
  end
end
```

Nótese lo siguiente:

- *explist* se evalúa sólo una vez. Sus resultados son una función *iterador*, un *estado* y un valor inicial para la primera *variable iteradora*.
- *f*, *s* y *var* son variables invisibles. Los nombres que aquí aparecen son sólo ilustrativos.
- Se puede usar una orden **break** para salir del bucle **for**.
- Las variables de control del bucle *var_i* son locales en el bucle; no se pueden usar sus valores después de que acabe el bucle **for**. Si se necesitan sus valores se deben asignar a otras variables antes de que se salga del bucle (normalmente o con un **break**).

2.4.6 – Sentencias de llamadas a función

Para evitar posibles efectos laterales, las llamadas a función pueden ser realizadas como sentencias:

```
sentencia ::= llamada_a_func
```

En ese caso todos los valores retornados se descartan. Las llamadas a función están explicadas en §2.5.8.

2.4.7 – Declaraciones locales

Las variables locales pueden ser declaradas en cualquier lugar dentro de un bloque. Esas declaraciones pueden incluir una asignación inicial:

```
sentencia ::= local lista_de_nombres ['=' explist]
```

Si está presente, una asignación inicial tiene la misma semántica que una asignación múltiple (véase §2.4.3). En otro caso todas las variables son inicializadas con **nil**.

Un *chunk* es también un bloque (véase §2.4.1), así que las variables locales pueden ser declaradas en un *chunk* fuera de cualquier bloque explícito. El ámbito de esas variables se extiende hasta el final del *chunk*.

Las reglas de visibilidad para las variables locales se exponen en §2.6.

2.5 – Expresiones

Las expresiones básicas en Lua son las siguientes:

```
exp ::= prefixexp
exp ::= nil  |  false  |  true
exp ::= Número
exp ::= String
exp ::= func
exp ::= constructor_de_tabla
exp ::= '...'
exp ::= exp operador_binario exp
exp ::= operador_unario exp
prefixexp ::= var  |  llamada_a_func  |  '(' exp ')'
```

Los números y los *string* literales se explican en §2.1; las variables se explican en §2.3; la definición de funciones se explica en §2.5.9; las llamadas a función se explican en §2.5.8; los constructores de tablas se explican en §2.5.7. Las expresiones *vararg* (que indican un número variable de argumentos en una función), denotadas mediante tres puntos ('`...`'), pueden ser usadas directamente sólo cuando están dentro de las funciones con *vararg*; se explican en §2.5.9.

Los operadores binarios comprenden los operadores aritméticos (véase §2.5.1), los operadores relacionales (véase §2.5.2) y los operadores lógicos (véase §2.5.3). Los operadores unarios compenden el menos unario (véase §2.5.1), el **not** unario (véase §2.5.3) y el *operador de longitud* unario (véase §2.5.5).

Tanto las llamadas a función como las expresiones *vararg* pueden resultar en valores múltiples. Si la expresión se usa como una sentencia (véase §2.4.6) (sólo posible con llamadas a función), entonces su lista de valores retornados se ajusta a cero elementos, descartando todos los valores retornados. Si la expresión se usa como el último (o único) elemento de una lista de expresiones entonces no se realiza ningún ajuste (a no ser que la llamada se encierre entre paréntesis). En todos los demás contextos Lua ajusta el resultado de la lista a un solo elemento, descartando todos los valores excepto el primero.

He aquí varios ejemplos:

```
f()                -- ajustado a 0 resultados
g(f(), x)          -- f() es ajustado a 1 resultado
g(x, f())          -- g toma x y todos los valores devueltos por f()
a,b,c = f(), x     -- f() se ajusta a 1 resultado (c toma el valor nil)
a,b = ...          -- a toma el primer argumento vararg, b toma
                   -- el segundo (a y b pueden ser nil si no existen los
                   -- correspondientes argumentos vararg)
a,b,c = x, f()     -- f() se ajusta a 2 resultados
a,b,c = f()        -- f() se ajusta a 3 resultados
return f()         -- retorna todos los valores devueltos por f()
```

```
return ...        -- retorna todos los argumentos vararg recibidos
return x,y,f()    -- retorna x, y, y todos los valores devueltos por f()
{f()}             -- crea una lista con todos los valores retornados por f()
{...}             -- crea una lista con todos los argumentos vararg
{f(), nil}        -- f() se ajusta a 1 resultado
```

Una expresión encerrada en paréntesis siempre resulta en un único valor. Entonces, `(f(x,y,z))` siempre es un valor único, incluso si `f` retorna varios valores. (El valor de `(f(x,y,z))` es el primer valor retornado por `f` o **nil** si `f` no retorna ningún valor).

2.5.1 – Operadores aritméticos

Lua tiene los operadores aritméticos comunes: los binarios + (adición), - (substracción), * (multiplicación), / (división), % (módulo) y ^ (exponenciación); y el unario - (negación). Si los operandos son números o *strings* que se convierten a números (véase §2.2.1), entonces todas las operaciones tienen el significado corriente. La exponenciación trabaja con cualquier exponente. Por ejemplo, `x^(-0.5)` calcula la inversa de la raiz cuadrada de `x`. El módulo se define como

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

Esto es, es el resto de la división que redondea el cociente hacia menos infinito.

2.5.2 – Operadores relacionales

Los operadores relacionales en Lua son

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

Devuelven siempre un resultado **false** o **true**.

La igualdad (==) primero compara el tipo de los operandos. Si son diferentes entonces el resultado es **false**. En otro caso se comparan los valores de los operandos. Los números y los *strings* se comparan de la manera usual. Los objetos (tablas, *userdata*, procesos y funciones) se comparan por *referencia*: dos objetos se consideran iguales sólo si son el *mismo* objeto. Cada vez que se crea un nuevo objeto (una tabla, *userdata*, proceso o función) este nuevo objeto es diferente de todos los demás objetos preexistentes.

Se puede cambiar la manera en que Lua compara tablas y *userdata* usando el metamétodo "eq" (véase §2.8).

Las reglas de conversión de §2.2.1 *no* se aplican en las comparaciones de igualdad. De este modo `"0"==0` es **false**, y `t[0]` y `t["0"]` denotan diferentes entradas en una tabla.

El operador ~= es exactamente la negación de la igualdad (==).

El orden de los operadores funciona de la siguiente manera. Si ambos argumentos son números entonces se comparan como tales. En otro caso, si ambos argumentos son *strings* sus valores se comparan de acuerdo al sistema local. En otro caso, Lua trata de usar los metamétodos "lt" o "le" (véase §2.8).

2.5.3 – Operadores lógicos

Los operadores lógicos en Lua son **and**, **or** y **not**. Como las estructuras de control (véase §2.4.4) todos los operadores lógicos consideran **false** y **nil** como falso y todo lo demás como verdadero.

El operador negación **not** siempre retorna **false** o **true**. El operador conjunción **and** retorna su primer operando si su valor es **false** o **nil**; en caso contrario **and** retorna su segundo operando. El operador disyunción **or** retorna su primer operando si su valor es diferente de **nil** y **false**; en caso contrario **or** retorna su segundo argumento. Tanto **and** como **or** usan evaluación de *cortocircuito*; esto es, su segundo operando se evalúa sólo si es necesario. He aquí varios ejemplos:

```
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
```

(En este manual '`-->`' indica el resultado de la expresión precedente.)

2.5.4 – Concatenación

El operador de concatenación de *strings* en Lua se denota mediante dos puntos seguidos ('`..`'). Si ambos operandos son *strings* o números entonces se convierten a *strings* mediante las reglas mencionadas en §2.2.1. En otro caso se invoca al metamétodo "concat" (véase §2.8).

2.5.5 – El operador longitud

El operador longitud se denota mediante `#`. La longitud de un *string* es su número de bytes (significado normal de la longitud de un *string* cuando cada carácter ocupa un byte).

La longitud de una tabla `t` se define como un índice entero `n` tal que `t[n]` no es **nil** y `t[n+1]` es **nil**; además, si `t[1]` es **nil** entonces `n` puede ser cero. Para un *array* regular, con valores no **nil** desde 1 hasta un `n` dado, la longitud es exactamente `n`, el índice es su último valor. Si el *array* tiene "agujeros" (esto es, valores **nil** entre otros valores que no lo son), entonces `#t` puede ser cualquiera de los índices que preceden a un valor **nil** (esto es, Lua puede considerar ese valor **nil** como el final del *array*).

2.5.6 – Precedencia de los operadores

La precedencia de los operadores en Lua sigue lo expuesto en la tabla siguiente de menor a mayor prioridad:

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

Como es usual, se pueden usar paréntesis para cambiar la precedencia en una expresión. Los operadores de concatenación ('..') y de exponenciación ('^') son asociativos por la derecha. Todos los demás operadores son asociativos por la izquierda.

2.5.7 – Constructores de tabla

Los constructores de tabla son expresiones que crean tablas. Cada vez que se evalúa un constructor se crea una nueva tabla. Los constructores pueden ser usados para crear tablas vacías o para crear tablas e inicializar alguno de sus campos. La sintaxis general para esos constructores es

```
constructor_de_tabla ::= '{' [lista_de_campos] '}'
lista_de_campos ::= campo {separador_de_campos campo} [separador_de_campos]
campo ::= '[' exp ']' '=' exp  |  nombre '=' exp  |  exp
separador_de_campos ::= ','  |  ';'
```

Cada campo de la forma `[exp1] = exp2` añade una entrada a la nueva tabla con la clave `exp1` y con el valor `exp2`. Un campo de la forma `nombre = exp` equivale a `["nombre"] = exp`. Finalmente, campos de la forma `exp` son equivalentes a `[i] = exp`, donde `i` son números enteros consecutivos, comenzando con 1. Los campos en el otro formato no afectan este contador. Por ejemplo,

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

equivale a

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

Si el último campo en la lista tiene la forma `exp` y la expresión es una llamada a función o una expresión *vararg*, entonces todos los valores retornados por esta expresión entran en la lista consecutivamente (véase §2.5.8). Para evitar esto debe encerrarse la llamada a la función (o la expresión *vararg*) entre paréntesis (véase §2.5).

La lista de campos puede tener un separador opcional al final, una conveniencia para código fuente generado de manera automática.

2.5.8 – Llamadas a función

Una llamada a una función tiene en Lua la siguiente sintaxis:

```
llamada_a_func ::= prefixexp argumentos
```

En una llamada a función, se evalúan primero *prefixexp* y los *argumentos*. Si el valor de *prefixexp* es del tipo *function*, entonces se invoca a esta función con los argumentos dados. En caso contrario se invoca el metamétodo "call", pasando como primer argumento el valor de *prefixexp* seguido por los argumentos originales de la llamada (véase §2.8).

La forma

```
llamada_a_func ::= prefixexp ':' nombre argumentos
```

puede ser usada para invocar "métodos". Una llamada `v:nombre(...)` es otra manera de expresar `v.nombre(v,...)`, excepto que `v` se evalúa sólo una vez.

Los argumentos tienen la siguiente sintaxis:

```
argumentos ::= '(' [explist] ')'
argumentos ::= constructor_de_tabla
argumentos ::= String
```

Todos los argumentos de la expresión se evalúan antes de la llamada. Un llamada de la forma `f{...}` es otra manera de expresar `f({...})`; esto es, la lista de argumentos es una nueva tabla simple. Una llamada de la forma `f'...'` (o `f"..."` o `f[[...]]`) es otra manera de expresar `f('...')`; esto es, la lista de argumentos es un *string* literal simple.

Como excepción a la sintaxis de formato libre de Lua, no se puede poner una rotura de línea antes de '(' en una llamada a función. Esta restricción evita algunas ambigüedades en el lenguaje. Si se escribe

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

Lua podría ententerlo como una sentencia simple, `a = f(g).x(a)`. Entonces, si se desean dos sentencias se debe añadir un punto y coma entre ellas. Si realmente se desea llamar a `f`, se debe eliminar la rotura de línea antes de `(g)`.

Una llamada de la forma `return` *llamada_a_func* se denomina una *llamada de cola*. Lua implementa *llamadas de cola correctas* (o *recursión de cola correcta*): en una llamada de cola la función invocada reutiliza la entrada en la pila de la función que la está llamando. Por tanto no existe límite en el número de llamadas de cola anidadas que un programa puede ejecutar. Sin embargo una llamada de cola borra cualquier información de depuración relativa a la función invocante. Nótese que una llamada de cola sólo ocurre con una sintaxis particular donde el **return** tiene una llamada simple a función como argumento; esta sintásis hace que la función invocante devuelva exactamente el retorno de la función invocada. Según esto ninguno de los siguientes ejemplos son llamadas de cola:

```
return (f(x))        -- resultados ajustados a 1
return 2 * f(x)
return x, f(x)       -- resultados adicionales
f(x); return         -- resultados descartados
return x or f(x)     -- resultados ajustados a 1
```

2.5.9 – Definición de funciones

La sintaxis para la definición de funciones es

```
func ::= function cuerpo_de_func
cuerpo_de_func ::= '(' [lista_de_argumentos] ')' bloque end
```

La siguiente forma simplifica la definición de funciones:

```
sentencia ::= function nombre_de_func cuerpo_de_func
sentencia ::= local function nombre cuerpo_de_func
nombre_de_func ::= nombre {'.' nombre} [':' nombre]
```

La sentencia

```
function f () cuerpo_de_función end
```

se traduce en

```
f = function () cuerpo_de_función end
```

La sentencia

```
function t.a.b.c.f () cuerpo_de_función end
```

se traduce en

```
t.a.b.c.f = function () cuerpo_de_función end
```

La sentencia

```
local function f () cuerpo_de_función end
```

se traduce en

```
local f; f = function () cuerpo_de_función end
```

no en:

```
local f = function () cuerpo_de_función end
```

(Esto sólo entraña diferencias cuando el cuerpo de la función contiene referencias a `f`.)

Una definición de función es una expresión ejecutable, cuyo valor tiene el tipo *function*. Cuando Lua precompila un *chunk* todos sus cuerpos de función son también precompilados. Entonces cuando Lua ejecuta la definición de función, la misma es *instanciada* (o *cerrada*). Esta instancia de función (o *closure*) es el valor final de la expresión. Diferentes instancias de la misma función pueden referirse a diferentes variables locales externas y pueden tener diferentes tablas de entorno.

Los argumentos formales de una función actúan como variables locales que son inicializadas con los valores actuales de los argumentos:

```
lista_de_argumentos ::= lista_de_nombres [',' '...']  |  '...'
```

Cuando se invoca una función, la lista de argumentos actuales se ajusta a la longitud de la lista de argumentos formales, a no ser que la función sea de tipo *vararg*, lo que se indica por tres puntos ('`...`') al final de la lista de argumentos formales. Una función *vararg* no ajusta su lista de argumentos; en su lugar recolecta todos los argumentos actuales extra y se los pasa a la función a través de una *expresión vararg*, lo que también se indica por medio de tres puntos. El valor de esta

expresión es una lista de todos los argumentos actuales extra, similar a una función con resultados múltiples. Si la expresión *vararg* se usa en el interior de otra expresión o en el medio de una lista de expresiones, entonces su retorno se ajusta a un sólo elemento. Si la expresión es usada como el último elemento de una lista de expresiones entonces no se hace ningún ajuste (a no ser que la llamada se realice entre paréntesis).

Como ejemplo, consideremos las siguientes definiciones:

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

Entonces tenemos la siguiente correspondencia de los argumentos actuales a los formales y a la expresión *vararg*:

```
LLAMADA           ARGUMENTOS

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, ... -->  (nada)
g(3, 4)           a=3, b=4,   ... -->  (nada)
g(3, 4, 5, 8)     a=3, b=4,   ... -->  5  8
g(5, r())         a=5, b=1,   ... -->  2  3
```

Los resultados se devuelven usando la sentencia **return** (véase §2.4.4). Si el flujo del programa alcanza el final de una función sin encontrar una sentencia **return** entonces la función retorna sin resultados.

La sintaxis *con dos puntos* (':') se usa para definir *métodos*, esto es, funciones que tienen un argumento extra denominado `self`. Entonces la sentencia

```
function t.a.b.c:f (params) cuerpo_de_función end
```

es otra manera de expresar

```
t.a.b.c.f = function (self, params) cuerpo_de_función end
```

2.6 – Reglas de visibilidad

Lua es un lenguaje con ámbito léxico. El ámbito de las variables comienza en la primera sentencia *después* de su declaración y termina al final del bloque más interno que incluya la declaración. Consideremos el siguiente ejemplo:

```
x = 10                -- variable global
do                    -- nuevo bloque
  local x = x         -- nueva 'x', con valor 10
  print(x)            --> 10
  x = x+1
  do                  -- otro bloque
```

```
    local x = x+1     -- otra 'x'
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10  (el valor de la variable global)
```

Tengase presente que en una declaración como `local x = x`, la nueva `x` que está siendo declarada no tiene ámbito todavía, y la segunda `x` se refiere a la variable externa.

Debido a las reglas de ámbito léxico, las variables locales pueden ser accedidas por funciones definidas en el interior de su propio ámbito. Una variable local usada en una función interna se denomina *upvalue* o *variable local externa* en el interior de la función.

Nótese que cada ejecución de una sentencia **local** define nuevas variables locales. Considérese el siguiente ejemplo:

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

El bucle crea diez *closures* (esto es, diez instancias de una función anónima). Cada uno de estas instancias usa una variable y diferente, mientras que todas ellas comparten la misma `x`.

2.7 – Manejo de errores

Debido a que Lua es un lenguaje de extensión embebido, todas las acciones de Lua comienzan con código C en el programa anfitrión llamando a una función de la biblioteca de Lua (véase `lua_pcall`). Cada vez que ocurra un error durante la compilación o ejecución de Lua, el control retorna a C, que puede tomar las medidas apropiadas (tales como imprimir un mensaje de error).

Se puede generar (o activar) explícitamente en Lua un error invocando la función `error`. Si se necesita capturar errores en Lua se puede usar la función `pcall`.

2.8 – Metatablas

Cada valor en Lua puede tener una *metatabla*. Ésta es una tabla ordinaria de Lua que define el comportamiento del valor original para ciertas operaciones especiales. Se pueden cambiar varios aspectos del comportamiento de las operaciones realizadas sobre un valor estableciendo campos específicos en su metatabla. Por ejemplo, cuando un valor no numérico es un operando de una adición Lua busca una función en el campo `"__add"` de su metatabla. Si se encuentra una, entonces se invoca esa función para realizar la adición.

Llamamos *eventos* a los campos de una metatabla y a los valores los denominamos *metamétodos*. En el ejemplo anterior el evento es `"add"` mientras que el metamétodo es la función que realiza la adición.

Se puede solicitar la metatabla de cualquier valor a través de la función `getmetatable`.

Se puede reemplazar la metatabla de una tabla a través de la función `setmetatable`. No se puede cambiar la metatabla de otros tipos de datos desde Lua (excepto usando la biblioteca de depuración); se debe usar la API de C para ello.

Las tablas y los *userdata* completos tienen metatablas individuales (aunque varias tablas y *userdata* pueden compartir sus metatablas); los valores de los otros tipos comparten una única metatabla por tipo. Por tanto, existe una única metatabla para todos los números, otra para todos los *strings*, etc.

Una metatabla puede controlar cómo se comporta un objeto en las operaciones aritméticas, en las comparaciones de orden, en la concatenación, en la operación longitud y en el indexado. Una metatabla puede también definir una función que será invocada cuando se libera memoria ocupada (*garbage collection*) por *userdata*. A cada una de esas operaciones Lua le asocia una clave específica denominada *evento*. Cuando Lua realiza una de esas operaciones con un valor, comprueba si éste tiene una metatabla con el correspondiente evento. Si es así, el valor asociado con esa clave (el *metamétodo*) controla cómo realiza Lua la operación.

Las metatablas controlan las operaciones listadas a continuación. Cada operación se identifica por su correspondiente nombre. La clave asociada a cada operación es un *string* con su nombre prefijado con dos subrayados, '__'; por ejemplo, la clave para la operación "add" es el *string* `"__add"`. La semántica de esas operaciones está mejor expuesta a través de una función Lua que describe cómo ejecuta el intérprete la operación.

El código Lua mostrado aquí es sólo ilustrativo; el comportamiento real está codificado internamente en el intérprete y es mucho más eficiente que esta simulación. Todas las funciones usadas en estas descripciones (`rawget`, `tonumber`, etc.) están descritas en §5.1. En particular, para recuperar el metamétodo de un objeto dado, usamos la expresión

```
metatable(objeto)[evento]
```

Esto puede también ser expresado mediante

```
rawget(getmetatable(objeto) or {}, evento)
```

Por tanto, el acceso a un metamétodo no invoca otros metamétodos, y el acceso a los objetos sin metatablas no falla (simplemente devuelve **nil**).

- **"add":** La operación +.

  La función `getbinhandler` que aparece más abajo define cómo escoge Lua un manejador de la operación binaria. Primero Lua prueba el primer operando. Si su tipo no define un manejador para la operación entonces Lua lo intenta con el segundo operando.

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

  Usando esta función el comportamiento del código `op1 + op2` es

  ```
  function add_event (op1, op2)
    local o1, o2 = tonumber(op1), tonumber(op2)
    if o1 and o2 then  -- ¿son numéricos ambos operandos?
      return o1 + o2   -- '+' aquí es la primitiva 'add'
    else  -- al menos uno de los operandos es no numérico
  ```

```
    local h = getbinhandler(op1, op2, "__add")
    if h then
      -- invoca el manejador de ambos operandos
      return (h(op1, op2))
    else  -- no existe un manejador disponible: comportamiento por defecto
      error(···)
    end
  end
end
```

- **"sub":** La operación `-`. El comportamiento es similar a la operación "add".
- **"mul":** La operación `*`. El comportamiento es similar a la operación "add".
- **"div":** La operación `/`. El comportamiento es similar a la operación "add".
- **"mod":** La operación `%`. El comportamiento es similar a la operación "add", usando `o1 - floor(o1/o2)*o2` como operación primitiva.
- **"pow":** La operación `^` (exponenciación). El comportamiento es similar a la operación "add", con la función `pow` (de la biblioteca matemática de C) como operación primitiva.
- **"unm":** La operación `-` unaria.

```
function unm_event (op)
  local o = tonumber(op)
  if o then  -- ¿es numérico el operando?
    return -o  -- '-' aquí es la función primitiva 'unm'
  else  -- el operando no es numérico.
    -- intentar obtener un manejador para el operando
    local h = metatable(op).__unm
    if h then
      -- invocar el manejador con el operando
      return (h(op))
    else  -- no hay manejador disponible: comportamiento por defecto
      error(···)
    end
  end
end
```

- **"concat":** La operación `..` (concatenación).

```
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  -- concatenación primitiva de strings
  else
    local h = getbinhandler(op1, op2, "__concat")
    if h then
      return (h(op1, op2))
    else
      error(···)
    end
```

```
  end
end
```

- **"len":** La operación #.

```
function len_event (op)
  if type(op) == "string" then
    return strlen(op)         -- longitud primitiva de string
  elseif type(op) == "table" then
    return #op                -- longitud primitiva de tabla
  else
    local h = metatable(op).__len
    if h then
      -- invocar el manejador con el operando
      return (h(op))
    else  -- no hay manejador disponible: comportamiento por defecto
      error(···)
    end
  end
end
```

  Véase §2.5.5 para una descripción de la longitud de una tabla.

- **"eq":** La operación ==. La función `getcomphandler` define cómo elige Lua un metamétodo para el operador de comparación. Se selecciona un metamétodo cuando ambos objetos que son comparados tienen el mismo tipo y el mismo metamétodo para la operación dada.

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

  El evento "eq" se define así:

```
function eq_event (op1, op2)
  if type(op1) ~= type(op2) then  -- ¿diferentes tipos?
    return false   -- diferentes objetos
  end
  if op1 == op2 then   -- ¿iguales primitivas?
    return true   -- los objetos son iguales
  end
  -- probar un metamétodo
  local h = getcomphandler(op1, op2, "__eq")
  if h then
    return (h(op1, op2))
  else
    return false
  end
end
```

  `a ~= b` equivale a `not (a == b)`.

- **"lt":** La operación <.

```
function lt_event (op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 < op2   -- comparación numérica
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 < op2   -- comparación lexicográfica
  else
    local h = getcomphandler(op1, op2, "__lt")
    if h then
      return (h(op1, op2))
    else
      error(···);
    end
  end
end
```

  `a > b` equivale a `b < a`.

- **"le":** La operación <=.

```
function le_event (op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 <= op2   -- comparación numérica
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 <= op2   -- comparación lexicográfica
  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` equivale a `b <= a`. Téngase presente que en ausencia de un metamétodo "le" Lua intenta usar el de "lt", asumiendo que `a <= b` equivale a `not (b < a)`.

- **"index":** El acceso indexado `tabla[clave]`.

```
function gettable_event (tabla, clave)
  local h
  if type(tabla) == "table" then
    local v = rawget(tabla, clave)
    if v ~= nil then return v end
    h = metatable(tabla).__index
    if h == nil then return nil end
```

```
  else
    h = metatable(tabla).__index
    if h == nil then
      error(···);
    end
  end
  if type(h) == "function" then
    return (h(tabla, clave))  -- invocar el manejador
  else return h[clave]        -- o repetir la operación con él
  end
end
```

- **"newindex":** La asignación indexada `tabla[clave] = valor`.

```
function settable_event (tabla, clave, valor)
  local h
  if type(tabla) == "table" then
    local v = rawget(tabla, clave)
    if v ~= nil then rawset(tabla, clave, valor); return end
    h = metatable(tabla).__newindex
    if h == nil then rawset(tabla, clave, valor); return end
  else
    h = metatable(tabla).__newindex
    if h == nil then
      error(···);
    end
  end
  if type(h) == "function" then
    h(tabla, clave, valor)    -- invoca el manejador
  else h[clave] = valor       -- o repite la operación con él
  end
end
```

- **"call":** invocada cuando Lua llama a un valor.

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

2.9 – Entornos

Además de metatablas, los objetos de tipo proceso, las funciones y los *userdata* tienen otra tabla asociada, denominada *entorno*. Como las metatablas los entornos son tablas normales y varios

objetos pueden compartir el mismo entorno.

Los entornos asociados con *userdata* no tienen significado en Lua. Es sólo una característica para los programadores asociar una tabla con *userdata*.

Los entornos asociados con procesos se denominan *entornos globales*. Son usados como entornos por defecto para los procesos y para las funciones no anidadas creadas por el proceso (a través de `loadfile`, `loadstring` o `load`) y pueden ser accedidas directamente por el código en C (véase §3.3).

Los entornos asociados con funciones C pueden ser accedidos directamente por el código en C (véase §3.3). Son usadas como entornos por defecto por otras funciones C creadas por la función dada.

Los entornos asociados con funciones en Lua son utilizados para resolver todos los accesos a las variables globales dentro de la función (véase §2.3). Son usados también como entornos por defecto por otras funciones en Lua creadas por la función.

Se puede cambiar el entorno de una función Lua o de un proceso en ejecución invocando `setfenv`. Se puede obtener el entorno de una función Lua o del proceso en ejecución invocando `getfenv`. Para manejar el entorno de otros objetos (*userdata*, funciones C, otros procesos) se debe usar la API de C.

2.10 – Liberación de memoria no utilizada

Lua realiza automáticamente la gestión de la memoria. Esto significa que no debemos preocuparnos ni de asignar (o reservar) memoria para nuevos objetos ni de liberarla cuando los objetos dejan de ser necesarios. Lua gestiona la memoria automáticamente ejecutando un *liberador de memoria* (*garbage collector*) de cuando en cuando para eliminar todos los *objetos muertos* (esos objetos que ya no son accesibles desde Lua). Todos los objetos en Lua son susceptibles de gestión automática: tablas, *userdata*, funciones, procesos y *strings*.

Lua implementa un liberador de memoria del tipo marcado-barrido incremental. Utiliza dos números para controlar sus ciclos de liberación de memoria: la *pausa del liberador de memoria* y el *multiplicador del paso del liberador de memoria*.

La pausa del liberador de memoria controla cuánto tiempo debe esperar el liberador de memoria antes de comenzar un nuevo ciclo. Valores grandes hacen al liberador menos agresivo. Valores menores que 1 significan que el liberador no esperará para comenzar un nuevo ciclo. Un valor de 2 significa que el liberador espera que la memoria total en uso se doble antes de comenzar un nuevo ciclo.

El multiplicador del paso controla la velocidad relativa del liberador en cuanto a asignación de memoria. Los valores más largos hacen el liberador más agresivo pero también aumentan el tamaño de cada paso incremental. Valores menores que 1 hacen el liberador demasiado lento y puede resultar en que el liberador nunca acabe un ciclo. El número por defecto, 2, significa que el liberador se ejecuta a una velocidad doble que la asignación de memoria.

Se pueden cambiar esos números invocando en C a `lua_gc` o en Lua a `collectgarbage`. Ambos tienen como argumentos un porcentaje (y entonces un argumento 100 significa un valor real de 1). Con esas funciones se puede también controlar el liberador directamente (por ejemplo, pararlo y reiniciarlo).

2.10.1 – Metamétodos de liberación de memoria

Usando la API de C se pueden establecer metamétodos liberadores de memoria para *userdata* (véase §2.8). Esos metamétodos se denominan también *finalizadores*. Éstos permiten coordinar el sistema de liberación de memoria de Lua con gestores externos de recursos (tales como cerrar ficheros, conexiones de red o de bases de datos, o liberar su propia memoria).

Los *userdata* que se van a liberar con un campo `__gc` en sus metatablas no son liberados inmediatamente por el liberador de memoria. En su lugar Lua los pone en una lista. Después de eso Lua hace el equivalente a la siguiente función para cada *userdata* en la lista:

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

Al final de cada ciclo de liberación de memoria, los finalizadores de *userdata* que aparecen en la lista que va a ser liberada son invocados en orden *inverso* al de su creación. Esto es, el primer finalizador en ser invocado es el que está asociado con el *userdata* creado en último lugar por el programa. El propio *userdata* puede ser liberado sólo en el próximo ciclo de liberación de memoria.

2.10.2 – Tablas débiles

Una *tabla débil* es una tabla cuyos elementos son *referencias débiles*. Una referencia débil es ignorada por el liberador de memoria. En otras palabras, si las únicas referencias a un objeto son referencias débiles entonces se libera la memoria asociada con este objeto.

Una tabla débil puede tener claves débiles, valores débiles o ambas cosas. Una tabla con claves débiles permite la liberación de sus claves, pero prohibe la liberación de sus valores. Una tabla con claves débiles y valores débiles permite la liberación tanto de claves como de valores. En cualquier caso, ya sea la clave o el valor el liberado, el par completo es eliminado de la tabla. La debilidad de una tabla está controlada por el campo `__mode` de su metatabla. Si el campo `__mode` es un *string* que contiene el carácter 'k', las claves en la tabla son débiles. Si `__mode` contiene 'v', los valores en la tabla son débiles.

Después de usar una tabla como metatabla no se debería cambiar el valor de su campo `__mode`. En caso contrario el comportamiento débil de las tablas controladas por esa metatabla es indefinido.

2.11 – co-rutinas

Lua tiene co-rutinas, también denominadas *multiprocesos colaborativos*. En Lua una co-rutina representa un proceso de ejecución independiente. A diferencia de los sistemas *multiproceso*, en Lua una co-rutina sólo suspende su ejecución invocando de manera explícita una función *yield* (cesión).

Se pueden crear co-rutinas con una llamada a `coroutine.create`. El único argumento de esta función es otra función que es la función principal de la co-rutina.

Cuando se llama por primera vez a `coroutine.resume`, pasándole como argumento el proceso retornado por `coroutine.create`, la co-rutina comienza a ejecutarse en la primera línea de su función principal. Los argumentos extra pasados a `coroutine.resume` se pasan a su vez a la función principal de la co-rutina. Después de que la co-rutina empieza a ejecutarse lo hace hasta que termina o se produce una *cesión* del control de flujo del programa.

Una co-rutina puede terminar su ejecución de dos maneras: normalmente, cuando su función principal retorna (explícita o implícitamente, después de su última instrucción); y anormalmente, si se produjo un error no protegido. En el primer caso, `coroutine.resume` devuelve **true**, más cualquier valor retornado por la función principal de la co-rutina. En caso de error `coroutine.resume` devuelve **false** más un mensaje de error.

Una co-rutina cede el control invocando a `coroutine.yield`. Cuando una co-rutina cede el control la correspondiente `coroutine.resume` retorna inmediatamente, incluso si la cesión ocurre dentro de una llamada a una función anidada (esto es, no en la función principal, sino en una función directa o indirectamente invocada desde la función principal). En el caso de una cesión, `coroutine.resume` también devuelve **true**, más cualesquiera valores pasados a `coroutine.yield`. La próxima vez que se resuma la misma co-rutina, continuará su ejecución desde el punto en que fue realizada la cesión, con la llamada a `coroutine.yield` devolviendo cualquier argumento extra pasado a `coroutine.resume`.

La función `coroutine.wrap` crea una co-rutina, justo igual que lo haría `coroutine.create`, pero en lugar de retornar la co-rutina misma, devuelve una función que, cuando es invocada resume la co-rutina. Cualesquiera argumentos pasados a esta función pasan como argumentos a `coroutine.resume`. `coroutine.wrap` devuelve todos los valores retornados por `coroutine.resume`, excepto el primero (el código booleano de error). A diferencia de `coroutine.resume`, `coroutine.wrap` no captura errores; cualquier error se propaga a la rutina invocante.

Como ejemplo, considérese el siguiente código:

```
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"))
```

Cuando se ejecuta se produce la siguiente salida:

```
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
```

3 – La interface del programa de aplicación (API)

Esta sección describe la API de C para Lua, esto es, el conjunto de funciones C disponibles para que el programa anfitrión se comunique con Lua. Todas las funciones de la API y sus tipos y constantes relacionados están declaradas en el fichero de cabecera `lua.h`.

Aunque se usa el término "function", algunas rutinas en la API pueden ser macros. Todas esas macros usan cada uno de sus argumentos exactamente una vez (excepto su primer argumento, que es siempre el estado de Lua), y por tanto no generan efectos laterales ocultos.

Como en la mayoría de las bibliotecas de C, la funciones API de Lua no verifican la validez ni la consistencia de sus argumentos. Sin embargo se puede cambiar este comportamiento compilando Lua con las definiciones adecuadas para la macro `luai_apicheck`, en el fichero `luaconf.h`.

3.1 – La pila (*stack*)

Lua usa una *pila virtual* para pasar valores a y desde C. Cada elemento en esta pila representa un valor de Lua (**nil**, número, *string*, etc.).

Siempre que Lua llame al C, la función llamada obtiene una nueva pila, que es independiente de las pilas anteriores y de las pilas de las funciones C que todavía están activas. Esta pila contiene inicialmente cualquier argumento de la función C y es donde ésta coloca los resultados que deben ser devueltos a la rutina invocadora (véase `lua_CFunction`).

Por conveniencia, la mayoría de las operaciones de petición de la API no siguen una disciplina estricta de pila. En su lugar pueden referirse a cualquier elemento en la pila usando un *índice*: un valor positivo representa una posición *absoluta* en la pila (comenzando en 1); un valor negativo representa un *desplazamiento* relativo a la parte superior de la pila. Más específicamente, si la pila tiene *n* elementos, entonces el índice 1 representa el primer elemento (esto es, el elemento que fue colocado primero en la pila) y un índice *n* representa el último elemento; un índice *-1* también representa el último elemento (esto es, el elemento en la parte superior) y un índice *-n* representa el primer elemento. Decimos que un índice es *válido* si tiene un valor comprendido entre 1 y la parte superior de la pila (esto es, si `1 ≤ abs(índice) ≤ top`).

3.2 – El tamaño de la pila

Cuando el programador interacciona con la API de Lua es responsable de asegurar la consistencia. En particular *es responsable de controlar el crecimiento correcto de la pila*. Se puede

usar la función `lua_checkstack` para hacer crecer el tamaño de la pila.

Siempre que Lua llama al C, se asegura de que al menos existen `LUA_MINSTACK` posiciones disponibles en la pila. `LUA_MINSTACK` está definida como 20, así que normalmente el programador no tiene que preocuparse del espacio en la pila, a no ser que su código tenga bucles que coloquen elementos en la pila.

La mayoría de las funciones de petición aceptan como índice cualquier valor dentro del espacio disponible en la pila, o sea índices hasta el máximo del tamaño de la pila establecido mediante `lua_checkstack`. Esos índices se denominan *índices aceptables*. Más formalmente, definimos un *índice aceptable* de la siguiente manera:

```
(índice < 0 && abs(índice) <= top) ||
(índice > 0 && índice <= stackspace)
```

Nótese que 0 no es nunca un índice aceptable.

3.3 – Pseudoíndices

Excepto en los casos en que se indique, cualquier función que acepta índices válidos también puede ser invocada con *pseudoíndices*, los cuales representan algunos valores en Lua que son accesibles desde el código en C pero que no están en la pila. Los pseudoíndices son usados para acceder al entorno del proceso, al entorno de la función, al registro y a los *upvalues* de una función C (véase §3.4).

El entorno del proceso (donde "viven" las variables globales) está siempre en el pseudoíndice `LUA_GLOBALSINDEX`. El entorno de una función C que está en ejecución está siempre en el pseudoíndice `LUA_ENVIRONINDEX`.

Para acceder y cambiar el valor de una variable global se pueden usar operaciones normales de tabla en la tabla de entorno. Por ejemplo, para acceder al valor de una variable global se hace

```
lua_getfield(L, LUA_GLOBALSINDEX, nombre_de_variable_global);
```

3.4 – Instancias en C

Cuando se crea una función C es posible asociarle algunos valores, creando una *instancia en C*; esos valores se denominan *upvalues* y son accesibles a la función en cualquier momento en que sea invocada (véase `lua_pushcclosure`).

Siempre que se invoque a una función C sus *upvalues* se localizan en pseudoíndices específicos. Éstos se producen mediante la macro `lua_upvalueindex`. El primer valor asociado con una función está en la posición `lua_upvalueindex(1)`, y así sucesivamente. Cualquier acceso a `lua_upvalueindex(`*n*`)`, donde *n* es mayor que el número de *upvalues* de la función actual produce un índice aceptable pero inválido.

3.5 – El registro

Lua proporciona un *registro*, una tabla predefinida que puede ser usada por cualquier código en C para almacenar cualquier valor que Lua necesite guardar. Esta tabla se localiza siempre en el pseudoíndice `LUA_REGISTRYINDEX`. Cualquier biblioteca de C puede almacenar datos en esta tabla, pero debería tener cuidado de elegir claves diferentes de aquéllas usadas por otras

bibliotecas, para evitar conflictos. Típicamente se podría usar como clave un *string* conteniendo el nombre de la biblioteca o *userdata* "ligeros" con la dirección de un objeto de C en el código.

Las claves de tipo entero en el registro son usadas como mecanismo para referenciar, implementado en la biblioteca auxiliar, y por tanto no deberían ser usados para otros propósitos diferentes.

3.6 – Manejo de errores en C

Internamente Lua usa la función de C `longjmp` para facilitar el manejo de errores. (Se puede también elegir usar directamente excepciones si se trabaja en C++; véase el fichero `luaconf.h`.) Cuando Lua se encuentra con cualquier error (tal como un error de asignación de memoria, un error de tipo, un error de sintaxis o un error de ejecución) entonces *activa* un error, esto es, realiza un salto largo en la memoria. Un *entorno protegido* usa `setjmp` para establecer un punto de recuperación; cualquier error provoca un salto al punto de recuperación activo más reciente.

Muchas funciones de la API pueden activar un error, por ejemplo debido a un problema de asignación de memoria. La documentación de cada función indica si puede activar un error.

Dentro de una función C se puede activar un error invocando `lua_error`.

3.7 – Funciones y tipos

He aquí la lista de todas las funciones y tipos de la API de C por orden alfabético. Cada función tiene un indicador como éste: [-o, +p, *x*]

El primer campo, `o`, indica cuántos elementos elimina la función en la pila. El segundo campo, `p`, es cuantos elementos coloca la función en la pila. (Toda función siempre coloca sus resultados después de eliminar sus argumentos.) Un campo de la forma `x|y` significa que la función puede colocar (o eliminar) `x` ó `y` elementos, dependiendo de la situación; un signo de interrogación '`?`' significa que no se puede conocer cuántos elementos coloca/elimina la función observando sólo sus argumentos (por ejemplo, puede depender de lo qué esté en la pila). El tercer, campo `x`, indica si la función puede activar errores: '`-`' significa que la función nunca activa errores; '`m`' indica que la función puede activar un error sólo debido a falta de memoria; '`e`' indica que la función puede activar otros tipos de errores; 'v' indica que la función puede activar un error a propósito.

lua_Alloc

```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

El tipo de la función que maneja la memoria usada por los estados de Lua. La función que maneja memoria debe proporcionar una funcionalidad similar a `realloc`, pero no exactamente la misma. Sus argumentos son: `ud`, un puntero opaco pasado a `lua_newstate`; `ptr`, un puntero al bloque que está siendo reservado/reasignado/liberado; `osize`, el tamaño original del bloque; `nsize`, el nuevo tamaño del bloque. `ptr` es `NULL` si y sólo si `osize` es cero. Cuando `nsize` es cero, el manejador debe retornar `NULL`; si `osize` no es cero debe ser liberado el bloque apuntado por `ptr`. Cuando `nsize` no es cero el manejador retorna `NULL` si y sólo si no puede ejecutar la petición. Cuando `nsize` no es cero y `osize` es cero el manejador debería comportarse como `malloc`.

Cuando `nsize` y `osize` no son cero, el manejador se comporta como `realloc`. Lua asume que el manejador nunca falla cuando `osize >= nsize`.

He aquí una implementación simple para la función manejadora de memoria. Es usada en la biblioteca auxiliar por `lua_newstate`.

```
static void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {
  (void) ud;  (void) osize;  /* no usadas */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

Este código asume que `free(NULL)` no tiene efecto y que `realloc(NULL, size)` es equivalente a `malloc(size)`. ANSI C asegura ambos comportamientos.

lua_atpanic

`lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);` [-0, +0, -]

Establece una nueva función de pánico y devuelve la vieja.

Si ocurre un error fuera de cualquier entorno protegido Lua llama a la *función pánico* y luego invoca `exit(EXIT_FAILURE)`, saliendo por tanto de la aplicación anfitriona. Si usa otra función de pánico diferente, ésta puede evitar esta salida sin retorno (por ejemplo, haciendo un salto largo).

La función de pánico puede acceder al mensaje de error en la parte superior de la pila.

lua_call

`void lua_call (lua_State *L, int nargs, int nresults);` [-(nargs + 1), +nresults, *e*]

Llama a una función.

Para llamar a una función se debe usar el siguiente protocolo: primero, la función a ser invocada se coloca en la parte superior de la pila; entonces, se colocan también en la pila los argumentos de la función en orden directo; esto es, el primer argumento se coloca primero. Finalmente, se llama a `lua_call`; `nargs` es el número de argumentos que se han colocado en la pila. Todos los argumentos y el valor de la función se eliminan de la pila cuando la función es invocada. Los resultados de la función se colocan en la parte superior de la pila cuando retorna la función. El número de resultados se ajusta a `nresults`, a no ser que `nresults` sea `LUA_MULTRET`. En este caso se colocan *todos* los resultados de la función. Lua tiene cuidado de que los valores retornados se ajusten en el espacio de pila. Los resultados de la función son colocados en la pila en orden directo (el primero es colocado antes), por lo que después de la llamada el último resultado aparece en la parte superior de la pila.

Cualquier error dentro de la función llamada se propaga hacia atrás (con un `longjmp`).

El siguiente ejemplo muestra cómo puede el programa anfitrión hacer algo equivalente a este código en Lua:

```
a = f("how", t.x, 14)
```

Aquí está en C:

```
lua_getfield(L, LUA_GLOBALSINDEX, "f");          /* función que es llamada */
lua_pushstring(L, "how");                              /* primer argumento */
lua_getfield(L, LUA_GLOBALSINDEX, "t");           /* tabla que es indexada */
lua_getfield(L, -1, "x");          /* coloca en la pila t.x (2º argumento) */
lua_remove(L, -2);                               /* elimina 't' de la pila */
lua_pushinteger(L, 14);                                    /* 3º argumento */
lua_call(L, 3, 1);    /* llama a la función con 3 argumentos y 1 resultado */
lua_setfield(L, LUA_GLOBALSINDEX, "a"); /* modifica la variable global 'a' */
```

Téngase presente que el código de arriba está "equilibrado" al final, pues la pila ha vuelto a su configuración inicial. Esto está considerado como una buena práctica de programación.

lua_CFunction

```
typedef int (*lua_CFunction) (lua_State *L);
```

Tipo para las funciones C.

Con objeto de comunicar adecuadamente con Lua, una función C debe usar el siguiente protocolo, el cual define la manera en que son pasados los argumentos y los resultados: una función C recibe sus argumentos desde Lua en su pila en orden directo (el primer argumento se coloca primero). Por tanto, cuando comienza una función, `lua_gettop(L)` devuelve el número de argumentos recibidos por la función. Su primer argumento (si existe) está en el índice 1 y su último argumento está en el índice `lua_gettop(L)`. Para retornar valores a Lua una función C sólo los coloca en la pila, en orden directo (el primer resultado va primero), y retorna el número de resultados. Cualquier otro valor en la pila por debajo de los resultados debe ser adecuadamente descartado por Lua. Como una función Lua, una función C llamada desde Lua puede retornar varios resultados.

Como ejemplo, la siguiente función recibe un número variable de argumentos numéricos y retorna su media y su suma:

```
static int foo (lua_State *L) {
   int n = lua_gettop(L);   /* número de argumentos */
   lua_Number sum = 0;
   int i;
   for (i = 1; i <= n; i++) {
     if (!lua_isnumber(L, i)) {
       lua_pushstring(L, "argumento incorrecto en la función 'media'");
       lua_error(L);
     }
     sum += lua_tonumber(L, i);
   }
   lua_pushnumber(L, sum/n);    /* primer resultado */
   lua_pushnumber(L, sum);     /* segundo resultado */
   return 2;                /* número de resultados */
}
```

lua_checkstack

```
int lua_checkstack (lua_State *L, int extra);
```
[-0, +0, *m*]

Se asegura de que hay al menos `extra` posiciones libres en la pila. Devuelve `false` si no puede hacer crecer la pila hasta ese tamaño. Esta función nunca disminuye la pila; si la pila es ya más grande que el nuevo tamaño la deja sin modificar.

lua_close

```
void lua_close (lua_State *L);
```
[-0, +0, -]

Destruye todos los objetos en el estado dado de Lua (llamando al correspondiente metamétodo de liberación de memoria, si existe) y libera toda la memoria dinámica usada por este estado. En algunas plataformas puede no ser necesario llamar a esta función, debido a que todos los recursos se liberan de manera natural cuando finaliza el programa anfitrión. Por otro lado, programas de ejecución larga, como puede ser el *demonio* de un servidor web, pueden necesitar la liberación de estados tan pronto como éstos no se necesiten para evitar un crecimiento desmesurado.

lua_concat

```
void lua_concat (lua_State *L, int n);
```
[-n, +1, *e*]

Concatena los `n` valores de la parte superior de la pila, los elimina y deja el resultado en la parte superior de la pila. Si `n` es 1 el resultado es el valor simple en la pila (esto es, la función no hace nada); si `n` es 0 el resultado es un *string* vacío. La concatenación se realiza siguiendo la semántica normal de Lua (véase §2.5.4).

lua_cpcall

```
int lua_cpcall (lua_State *L, lua_CFunction func, void *ud);
```
[-0, +(0|1), -]

Invoca la función C `func` en modo protegido. `func` comienza con un solo elemento en su pila, un *userdata* ligero conteniendo `ud`. En caso de errores `lua_cpcall` devuelve el mismo código de error que `lua_pcall`, además del objeto error en la parte superior de la pila; en caso contrario retorna cero y no cambia la pila. Todos los valores retornados por `func` se descartan.

lua_createtable

```
void lua_createtable (lua_State *L, int narr, int nrec);
```
[-0, +1, *m*]

Crea una nueva tabla vacía y la coloca en la pila. La nueva tabla tiene espacio reservado para `narr` elementos *array* y `nrec` elementos *no array*. Esta reserva es útil cuando no se sabe cuántos elementos va a contener la tabla. En otro caso se puede usar la función `lua_newtable`.

lua_dump

```
int lua_dump (lua_State *L, lua_Writer writer, void *data);
```
[-0, +0, *m*]

Vuelca una función en forma de *chunk* binario. Recibe una función de Lua en la parte superior de la pila y produce un *chunk* binario que si se carga de nuevo resulta en una función equivalente a la volcada previamente. Según va produciendo partes del *chunk*, `lua_dump` invoca a la función `writer` (véase `lua_Writer`) con los datos `data` para escribirlos.

El valor retornado es el código de error devuelto por la última llamada a `Writer`; 0 significa no error.

Esta función no elimina de la pila la función de Lua.

lua_equal

```
int lua_equal (lua_State *L, int index1, int index2);
```
[-0, +0, *e*]

Retorna 1 si los dos valores en los índices aceptables `index1` e `index2` son iguales, siguiendo la semántica del operador == de Lua (esto es, puede invocar metamétodos). En otro caso retorna 0. También devuelve 0 si alguno de los índices no es válido.

lua_error

```
int lua_error (lua_State *L);
```
[-1, +0, *v*]

Genera un error de Lua. El mensaje de error (que puede ser realmente un valor de Lua de cualquier tipo) debe de estar en la parte superior de la pila. Esta función realiza un salto largo, y por tanto nunca retorna. (véase `luaL_error`).

lua_gc

```
int lua_gc (lua_State *L, int what, int data);
```
[-0, +0, *e*]

Controla el liberador de memoria.

Esta función realiza varias tareas, de acuerdo con el valor del argumento `what`:

- `LUA_GCSTOP` --- detiene el liberador de memoria.
- `LUA_GCRESTART` --- reinicia el liberador de memoria.
- `LUA_GCCOLLECT` --- realiza un ciclo completo de liberación de memoria.
- `LUA_GCCOUNT` --- retorna la cantidad actual de memoria (en Kbytes) en uso por Lua.
- `LUA_GCCOUNTB` --- retorna el resto de dividir por 1024 la cantidad actual de memoria en bytes en uso por Lua.
- `LUA_GCSTEP` --- realiza un paso incremental de liberación de memoria. El "tamaño" del paso está controlado por `data` (un valor mayor significa más pasos) de una manera no especificada. Si se desea controlar el tamaño del paso se debe afinar experimentalmente el valor de `data`. La función retorna 1 si el paso acabó con un ciclo de liberación de memoria.
- `LUA_GCSETPAUSE` --- establece `data`/100 como el nuevo valor de la *pausa* del liberador de memoria (véase §2.10). La función retorna el valor previo de la pausa.
- `LUA_GCSETSTEPMUL` --- establece `data`/100 como el nuevo valor del *multiplicador del paso* del liberador de memoria (véase §2.10). La función retorna el valor previo del multiplicador.

lua_getallocf

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```
[-0, +0, -]

Retorna la función manejadora de memoria de un estado dado. Si `ud` no es `NULL` Lua guarda en `*ud` el puntero opaco pasado a `lua_newstate`.

lua_getfenv

```
void lua_getfenv (lua_State *L, int index);
```
[-0, +1, -]

Coloca en la pila la tabla de entorno de un valor en el índice dado.

lua_getfield

```
void lua_getfield (lua_State *L, int index, const char *k);
```
[-0, +1, *e*]

Coloca en la pila el valor `t[k]`, donde `t` es el valor dado por el índice válido. Como en Lua esta función puede activar un metamétodo para el evento "index" (véase §2.8).

lua_getglobal

```
void lua_getglobal (lua_State *L, const char *name);
```
[-0, +1, *e*]

Coloca en la pila el valor del `nombre` global. Está definida como macro:

```
#define lua_getglobal(L,s)  lua_getfield(L, LUA_GLOBALSINDEX, s)
```

lua_getmetatable

```
int lua_getmetatable (lua_State *L, int index);
```
[-0, +(0|1), -]

Coloca en la pila la metatabla del valor situado en el índice aceptable dado. Si el índice no es válido o si el valor no tiene metatabla, la función retorna 0 y no coloca nada en la pila.

lua_gettable

```
void lua_gettable (lua_State *L, int index);
```
[-1, +1, *e*]

Coloca en la pila el valor `t[k]`, donde `t` es el valor en el índice válido y `k` es el valor situado en la parte superior de la pila.

Esta función quita la clave de la parte superior de la pila (colocando a su vez el valor resultante en su lugar). Como en Lua esta función puede activar un metamétodo para el evento "index" (véase §2.8).

lua_gettop

```
int lua_gettop (lua_State *L);
```
[-0, +0, -]

Retorna el índice del elemento situado en la parte superior de la pila. Debido a que los índices comienzan en 1 este resultado es igual al número de elementos en la pila (y así, 0 significa una pila vacía).

lua_insert

```
void lua_insert (lua_State *L, int index);
```
[-1, +1, -]

Mueve el elemento situado en la parte superior de la pila hacia el índice válido dado, desplazando hacia arriba los elementos por encima de este índice para abrir hueco. No puede ser invocada con un pseudoíndice debido a que éste no es una posición real en la pila.

lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

El tipo usado por la API de Lua para representar valores enteros.

Por defecto es `ptrdiff_t`, que es normalmente el tipo entero con signo más grande que la máquina maneja "confortablemente".

lua_isboolean

```
int lua_isboolean (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable tiene tipo booleano y 0 en caso contrario.

lua_iscfunction

```
int lua_iscfunction (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es una función C y 0 en caso contrario.

lua_isfunction

```
int lua_isfunction (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es una función (en C o en Lua) y 0 en caso contrario.

lua_islightuserdata

```
int lua_islightuserdata (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es un *userdata* ligero y 0 en caso contrario.

lua_isnil

```
int lua_isnil (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es **nil** y 0 en caso contrario.

lua_isnone

```
int lua_isnone (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es no válido (esto es, si se refiere a un elemento fuera de la pila actual) y 0 en caso contrario.

lua_isnoneornil

```
int lua_isnoneornil (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es no válido (esto es, si se refiere a un elemento fuera de la pila actual) o si el valor en este índice es `nil`, y 0 en caso contrario.

lua_isnumber

```
int lua_isnumber (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es un número o un *string* convertible a número y 0 en caso contrario.

lua_isstring

```
int lua_isstring (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es un *string* o un número (que es siempre convertible a un *string*) y 0 en caso contrario.

lua_istable

```
int lua_istable (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es una tabla y 0 en caso contrario.

lua_isthread

```
int lua_isthread (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es un proceso y 0 en caso contrario.

lua_isuserdata

```
int lua_isuserdata (lua_State *L, int index);
```
[-0, +0, -]

Retorna 1 si el valor en la situación del índice aceptable es un *userdata* (ligero o completo) y 0 en caso contrario.

lua_lessthan

```
int lua_lessthan (lua_State *L, int index1, int index2);
```
[-0, +0, *e*]

Retorna 1 si el valor situado en la posición del índice aceptable `index1` es menor que el situado en la posición del índice aceptable `index2`, siguiendo la semántica del operador < de Lua (esto es, puede invocar metamétodos). En caso contrario retorna 0. También retorna 0 si alguno de los índices es inválido.

lua_load

```
int lua_load (lua_State *L,
              lua_Reader reader,
              void *data,
              const char *chunkname);
```
[-0, +1, -]

Carga un *chunk* de Lua. Si no hay errores, `lua_load` coloca el *chunk* compilado en la parte superior de la pila. En caso contrario coloca ahí un mensaje de error. Los valores de retorno de `lua_load` son:

- 0 --- sin errores.
- `LUA_ERRSYNTAX` --- error de sintaxis durante la precompilación.
- `LUA_ERRMEM` --- error de reserva de memoria.

`lua_load` detecta automáticametne si el *chunk* está en binario o en forma de texto, y lo carga de acuerdo con esto (véase el programa `luac`).

`lua_load` usa una función `lectora` suplida por el usuario para leer el *chunk* (véase `lua_Reader`). El argumento `data` es un valor opaco pasado a la función lectora.

El argumento `chunkname` da un nombre al *chunk*. el cual es usado en los mensajes de error y en la información de depuración (véase §3.8).

lua_newstate

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```
[-0, +0, -]

Crea un nuevo estado independiente. Retorna `NULL` si no puede crear el estado (debido a falta de memoria). El argumento `f` es la función de reserva de memoria; Lua hace toda la reserva de memoria para este estado a través de esta función. El segundo argumento, `ud`, es un puntero opaco que Lua simplemente pasa al reservador de memoria en cada llamada.

lua_newtable

```
void lua_newtable (lua_State *L);
```
[-0, +1, *m*]

Crea una nueva tabla vacía y la coloca en la pila. Equivale a `lua_createtable(L, 0, 0)`.

lua_newthread

```
lua_State *lua_newthread (lua_State *L);
```
[-0, +1, *m*]

Crea un nuevo proceso, lo coloca en la pila y retorna un puntero a un `lua_State` que representa este nuevo proceso. El nuevo estado retornado por esta función comparte con el original todos los objetos globales (como las tablas), pero tiene una pila de ejecución independiente.

No existe una función explícita para cerrar o destruir un proceso. Los procesos están sujetos a liberación de memoria, como cualquier otro objeto de Lua.

lua_newuserdata

```
void *lua_newuserdata (lua_State *L, size_t size);
```
[-0, +1, *m*]

Esta función reserva un nuevo bloque de memoria con el tamaño dado, coloca en la pila un nuevo *userdata* completo con la dirección del bloque de memoria y retorna esta dirección.

Los *userdata* representan valores de C en Lua. Un *userdata completo* representa un bloque de memoria. Es un objeto (como una tabla): se puede crear, puede tener su propia metatabla y se

puede detectar cuándo está siendo eliminado de memoria. Un *userdata* completo es sólo igual a sí mismo (en un test de igualdad directa).

Cuando Lua libera un *userdata* completo con un metamétodo `gc`, llama al metamétodo y marca el *userdata* como finalizado. Cuando este *userdata* es liberado de nuevo entonces es cuando Lua libera definitivamente la memoria correspondiente.

lua_next

```
int lua_next (lua_State *L, int index);
```
[-1, +(2|0), *e*]

Elimina una clave de la pila y coloca un par clave-valor de una tabla en el índice dado (la "siguiente" pareja después de la clave dada). Si no hay más elementos en la tabla entonces `lua_next` retorna 0 (y no coloca nada en la pila).

Una típica iteración de recorrido de tabla sería:

```
/* la tabla está en la pila en el índice 't' */
lua_pushnil(L);  /* primera clave */
while (lua_next(L, t) != 0) {
  /* 'clave' está en el índice -2 y 'valor' en el índice -1 */
  printf("%s - %s\n",
         lua_typename(L, lua_type(L, -2)),
         lua_typename(L, lua_type(L, -1)));
  /* elimina 'valor'; mantiene 'clave' para la siguiente iteración */
  lua_pop(L, 1);
}
```

Mientras se recorre una tabla no debe llamarse a `lua_tolstring` directamente en una clave a no ser que se conozca que la clave es realmente un *string*. Recuerde que `lua_tolstring` *cambia* el valor en el índice dado; esto confunde a la siguiente llamada a `lua_next`.

lua_Number

```
typedef double lua_Number;
```

El tipo de los números en Lua. Por defecto es un `double`, pero puede ser cambiado en `luaconf.h`.

A través del fichero de configuración se puede cambiar Lua para que opere con otro tipo de números (por ejemplo, `float` o `long`).

lua_objlen

```
size_t lua_objlen (lua_State *L, int index);
```
[-0, +0, -]

Retorna la "longitud" de un valor situado en el índice aceptable: para un *string*, es la longitud del mismo; para una tabla, es el resultado del operador longitud ('#'); para un *userdata*, es el tamaño del bloque de memoria reservado para el mismo; para otros valores es 0.

lua_pcall

```
int lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);
```
[-(nargs + 1), +(nresults|1), -]

Invoca una función en modo protegido.

Tanto `nargs` como `nresults` tienen el mismo significado que en `lua_call`. Si no hay errores durante la llamada, `lua_pcall` se comporta exactamente igual que `lua_call`. Sin embargo en caso de error `lua_pcall` lo captura, colocando un único valor en la pila (el mensaje de error) y retorna un código de error. Como `lua_call`, `lua_pcall` siempre elimina la función y sus argumentos de la pila.

Si `errfunc` es 0 entonces el mensaje de error retornado en la pila es exactamente el mensaje original. En otro caso, `errfunc` es el índice en la pila de una *función manejadora de error*. (En la implementación actual, este índice no puede ser un pseudoíndice.) En caso de errores de ejecución esta función será llamada con el mensaje de error y el valor devuelto será el mensaje retornado en la pila por `lua_pcall`.

Típicamente la función manejadora de error se usa para añadir más información de depuración al mensaje de error, tal como un "trazado inverso" de la pila. Esa información no puede ser recolectada después del retorno de `lua_pcall`, puesto que por entonces la pila ya no tiene esa información.

La función `lua_pcall` retorna 0 en caso de éxito o uno de los siguientes códigos de error (definidos en `lua.h`):

- `LUA_ERRRUN` --- un error de ejecución.
- `LUA_ERRMEM` --- un error de reserva de memoria. Para este error Lua no llama a la función manejadora de error.
- `LUA_ERRERR` --- error mientras se está ejecutando la función manejadora de error.

lua_pop

```
void lua_pop (lua_State *L, int n);
```
[-n, +0, -]

Elimina `n` elementos de la pila.

lua_pushboolean

```
void lua_pushboolean (lua_State *L, int b);
```
[-0, +1, -]

Coloca el valor booleano `b` en la pila.

lua_pushcclosure

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
```
[-n, +1, *m*]

Coloca en la pila una nueva instancia en C.

Cuando se crea una función C es posible asociarle algunos valores, creando entonces una *instancia en C* (véase §3.4); estos valores son entonces accesibles a la función en cualquier momento en que sea invocada. Para asociar valores a una función C, primero éstos deberían colocarse en la pila (cuando hay varios, el primero se coloca antes). Entonces se invoca `lua_pushcclosure` para crear y colocar la función C en la pila, con el argumento `n` indicando cuantos valores están asociados con la misma. `lua_pushcclosure` también elimina esos valores de la pila.

lua_pushcfunction

`void lua_pushcfunction (lua_State *L, lua_CFunction f);` [-0, +1, *m*]

Coloca una función C en la pila. Esta función recibe un puntero a una función C y coloca en la pila un valor de Lua de tipo `function` que, cuando se llama, invoca la correspondiente función C.

Cualquier función que sea registrada en Lua debe seguir el protocolo correcto para recibir sus argumentos y devolver sus resultados (véase `lua_CFunction`).

`lua_pushcfunction(L, f)` está definida como una macro:

```
#define lua_pushcfunction(L, f)  lua_pushcclosure(L, f, 0)
```

lua_pushfstring

`const char *lua_pushfstring (lua_State *L, const char *fmt, ...);` [-0, +1, *m*]

Coloca en la pila un *string* formateado y retorna un puntero a este *string*. Es similar a la función `sprintf` de C, pero tiene con ella algunas importantes diferencias:

- No tiene que reservar memoria para el resultado, puesto que éste es un *string* de Lua, y Lua se encarga de realizar la reserva de memoria (y también la liberación).
- Los especificadores de conversión son bastante restrictivos. No hay indicadores, anchuras o precisiones. Los especificadores de conversión pueden ser sólo: '%%' (inserta un '%' en el *string*), '%s' (inserta un *string* terminado en cero sin restricciones de tamaño), '%f' (inserta un `lua_Number`), '%p' (inserta un puntero como número hexadecimal), '%d' (inserta un `int`), y '%c' (inserta un `int` como carácter).

lua_pushinteger

`void lua_pushinteger (lua_State *L, lua_Integer n);` [-0, +1, -]

Coloca un número entero de valor `n` en la pila.

lua_pushlightuserdata

`void lua_pushlightuserdata (lua_State *L, void *p);` [-0, +1, -]

Coloca un *userdata* ligero en la pila.

Los *userdata* representan valores de C en Lua. Un *userdata ligero* representa un puntero. Es un valor (como un número): no se crea ni tiene metatablas y no sufre liberación de memoria (puesto que nunca fue reservada). En una comparación de igualdad, un *userdata ligero* es igual que cualquier otro *userdata ligero* con la misma dirección en C.

lua_pushliteral

`void lua_pushliteral (lua_State *L, const char *s);` [-0, +1, *m*]

Esta macro es equivalente a `lua_pushlstring`, pero puede ser usada solamente cuando `s` es un *string* literal. En esos casos, proporciona automáticamente la longitud del *string*.

lua_pushlstring

```
void lua_pushlstring (lua_State *L, const char *s, size_t len);
```
[-0, +1, *m*]

Coloca el *string* apuntado por `s` con tamaño `len` en la pila. Lua realiza (o reutiliza) una copia interna del *string* dado, así que la memoria en `s` puede ser liberada o reutilizada inmediamente después de que la función retorne. El *string* puede contener ceros.

lua_pushnil

```
void lua_pushnil (lua_State *L);
```
[-0, +1, -]

Coloca un valor nil en la pila.

lua_pushnumber

```
void lua_pushnumber (lua_State *L, lua_Number n);
```
[-0, +1, -]

Coloca un número con valor `n` en la pila.

lua_pushstring

```
void lua_pushstring (lua_State *L, const char *s);
```
[-0, +1, *m*]

Coloca el *string* terminado en cero al que apunta `s` en la pila. Lua realiza (o reutiliza) una copia interna del *string* dado, así que la memoria en `s` puede ser liberada o reutilizada inmediamente después de que la función retorne. El *string* no puede contener caracteres cero; se asume que el final del mismo es el primer carácter cero que aparezca.

lua_pushthread

```
int lua_pushthread (lua_State *L);
```
[-0, +1, -]

Coloca un proceso representado por `L` en la pila. Retorna 1 si este proceso es el proceso principal de su estado.

lua_pushvalue

```
void lua_pushvalue (lua_State *L, int index);
```
[-0, +1, -]

Coloca una copia del elemento situado en el índice válido dado en la pila.

lua_pushvfstring

```
const char *lua_pushvfstring (lua_State *L,
                              const char *fmt,
                              va_list argp);
```
[-0, +1, *m*]

Equivalente a `lua_pushfstring`, excepto que recibe un argumento de tipo `va_list` en lugar de un número variable de argumentos.

lua_rawequal

```
int lua_rawequal (lua_State *L, int index1, int index2);
```
[-0, +0, -]

Retorna 1 si los dos valores situados en los índices aceptables `index1` e `index2` son iguales de manera primitiva (esto es, sin invocar metamétodos). En caso contrario retorna 0. También retorna 0 si alguno de los índices no es válido.

lua_rawget

```
void lua_rawget (lua_State *L, int index);
```
[-1, +1, -]

Similar a `lua_gettable`, pero realiza un acceso directo (sin metamétodos).

lua_rawgeti

```
void lua_rawgeti (lua_State *L, int index, int n);
```
[-0, +1, -]

Coloca en la pila el valor `t[n]`, donde `t` es el valor en el índice válido. El acceso es directo, esto es, sin invocar metamétodos.

lua_rawset

```
void lua_rawset (lua_State *L, int index);
```
[-2, +0, *m*]

Similar a `lua_settable`, pero realizando una asignación directa (sin invocar metamétodos).

lua_rawseti

```
void lua_rawseti (lua_State *L, int index, int n);
```
[-1, +0, *m*]

Realiza el equivalente a `t[n] = v`, donde `t` es el valor en el índice válido y `v` es el valor en la parte superior de la pila.

Esta función elimina el valor de la parte superior de la pila. La asignación es directa, sin invocar metamétodos.

lua_Reader

```
typedef const char * (*lua_Reader) (lua_State *L,
                                    void *data,
                                    size_t *size);
```

La función de lectura usada por `lua_load`. Cada vez que necesita otro fragmento de *chunk*, `lua_load` llama al "lector", pasándole su argumento `data`. El lector debe retornar un puntero a un bloque de memoria con un nuevo fragmento de *chunk* y establece `size` como el tamaño del bloque. El bloque debe existir hasta que la función lectora se invoque de nuevo. Para señalar el final del *chunk* el lector debe retornar `NULL`. La función lectora puede retornar fragmentos de cualquier tamaño mayor que cero.

lua_register

```
void lua_register (lua_State *L,
                   const char *name,
                   lua_CFunction f);
```
[-0, +0, *e*]

Establece la función C `f` como el nuevo valor del `nombre` global. Está definida en la macro:

```
#define lua_register(L,n,f) \
       (lua_pushcfunction(L, f), lua_setglobal(L, n))
```

lua_remove

```
void lua_remove (lua_State *L, int index);
```
[-1, +0, -]

Elimina el elemento en la posición del índice válido dado, desplazando hacia abajo los elementos que estaban por encima de este índice para llenar el hueco. No puede ser llamada con un pseudoíndice, debido a que éste no es una posición real en la pila.

lua_replace

```
void lua_replace (lua_State *L, int index);
```
[-1, +0, -]

Mueve el elemento que está en la parte superior de la pila a la posición dada (y lo elimina de la parte superior de la pila), sin desplazar ningún elemento de la misma (por tanto reemplazando el valor en la posición dada).

lua_resume

```
int lua_resume (lua_State *L, int narg);
```
[-?, +?, -]

Comienza y resume una co-rutina en un proceso dado.

Para comenzar una co-rutina se debe crear un nuevo proceso (véase `lua_newthread`); entonces se coloca en su propia pila la función principal más cualquier posible argumento; posteriormente se invoca `lua_resume`, con `narg` siendo el número de argumentos. Esta llamada retorna cuando la co-rutina suspende o finaliza su ejecución. Cuando retorna, la pila contiene todos los valores pasados a `lua_yield`, o todos los valores retornados por el cuerpo de la función. `lua_resume` retorna `LUA_YIELD` si la co-rutina cedió el control, 0 si la co-rutina acabó su ejecución sin errores, o un código de error en caso de errores (véase `lua_pcall`). En caso de error, se deja información en la pila, así que se puede usar la API de depuración con ella. El mensaje de error está en la parte superior de la pila. Para reiniciar una co-rutina se ponen en la pila sólo los valores que son pasados como resultado de `yield`, y entonces se invoca `lua_resume`.

lua_setallocf

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```
[-0, +0, -]

Utiliza `f` con el *userdata* `ud` como función de reserva de memoria de un estado dado .

lua_setfenv

```
int lua_setfenv (lua_State *L, int index);
```
[-1, +0, -]

Elimina una tabla de la parte superior de la pila y la toma como nuevo entorno para el valor situado en la posición del índice. Si el valor dado no es ni una función ni un proceso ni un *userdata* entonces `lua_setfenv` retorna 0. En caso contrario retorna 1.

lua_setfield

```
void lua_setfield (lua_State *L, int index, const char *k);
```
[-1, +0, *e*]

Realiza el equivalente a `t[k] = v`, donde `t` es el valor en la posición del índice válido y `v` es el valor en la parte superior de la pila.

Esta función elimina el valor de la pila. Como en Lua, esta función puede activar un metamétodo para el evento "newindex" (véase §2.8).

lua_setglobal

```
void lua_setglobal (lua_State *L, const char *name);
```
[-1, +0, *e*]

Elimina un valor de la pila y lo toma como nuevo valor del `nombre` global. Está definida en una macro:

```
#define lua_setglobal(L,s)   lua_setfield(L, LUA_GLOBALSINDEX, s)
```

lua_setmetatable

```
int lua_setmetatable (lua_State *L, int index);
```
[-1, +0, -]

Elimina una tabla de la pila y la toma como nueva metatabla para el valor en la situación del índice aceptable.

lua_settable

```
void lua_settable (lua_State *L, int index);
```
[-2, +0, *e*]

Hace el equivalente a `t[k] = v`, donde `t` es el valor en la posición del índice válido, `v` es el valor en la parte superior de la pila y `k` es el valor justamente debajo.

Esta función elimina de la pila tanto la clave como el valor. Como en Lua, esta función puede activar un metamétodo para el evento "newindex" (véase §2.8).

lua_settop

```
void lua_settop (lua_State *L, int index);
```
[-?, +?, -]

Acepta cualquier índice aceptable ó 0 y establece la parte superior de la pila en este índice. Si ese valor es mayor que el antiguo entonces los nuevos elementos se rellenan con **nil**. Si `index` es 0 entonces todos los elementos de la pila se eliminan.

lua_State

```
typedef struct lua_State lua_State;
```

Estructura opaca que almacena todo el estado de un intérprete de Lua. La biblioteca de Lua es totalmente re-entrante: no tiene variables globales. Toda la información acerca de un estado se guarda en esta estructura.

Un puntero a este estado debe ser pasado como primer argumento a cualquier función de la biblioteca, excepto a `lua_newstate`, la cual crea un nuevo estado de Lua desde cero.

lua_status

```
int lua_status (lua_State *L);
```
[-0, +0, -]

Retorna el estatus del proceso `L`.

El estatus puede ser 0 para un proceso normal, un código de error si el proceso finaliza su ejecución con un error, o `LUA_YIELD` si el proceso está suspendido.

lua_toboolean

```
int lua_toboolean (lua_State *L, int index);
```
[-0, +0, -]

Convierte el valor de Lua situado en la posición del índice aceptable en un booleano de C (0 ó 1). Como todos los test en Lua, `lua_toboolean` retorna 1 para cada valor de Lua diferente de **false** y **nil**; en caso contrario retorna 0. También retorna 0 cuando se invoca sin un índice válido. (Si se desea aceptar sólo los valores booleanos reales, úsese `lua_isboolean` para verificar el tipo del valor.)

lua_tocfunction

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```
[-0, +0, -]

Convierte en una función C el valor situado en el índice aceptable. Este valor debe ser una función C; en caso contrario retorna `NULL`.

lua_tointeger

```
lua_Integer lua_tointeger (lua_State *L, int index);
```
[-0, +0, -]

Convierte el valor de Lua situado en el índice aceptable en un entero sin signo del tipo `lua_Integer`. El valor de Lua debe ser un número o un *string* convertible a un número (véase §2.2.1); en otro caso `lua_tointeger` retorna 0.

Si el número no es entero se trunca de una manera no especificada.

lua_tolstring

```
const char *lua_tolstring (lua_State *L, int index, size_t *len);
```
[-0, +0, *m*]

Convierte el valor de Lua situado en la posición del índice aceptable en un *string* (`const char*`). Si `len` no es `NULL`, también establece `*len` como longitud del mismo. El valor Lua puede ser un *string* o un número; en caso contrario la función retorna `NULL`. Si el valor es un número entonces `lua_tolstring` también *cambia el valor actual en la pila a un string*. (Este cambio confunde a `lua_next` cuando `lua_tolstring` se aplica a claves durante el recorrido de una tabla.)

`lua_tolstring` retorna un puntero totalmente alineado a un *string* dentro de un estado de Lua. Este *string* siempre tiene un cero ('`\0`') después de su último carácter (como en C), pero puede contener otros ceros en su cuerpo. Debido a que Lua tiene liberación de memoria, no existen garantías de que el puntero retornado por `lua_tolstring` siga siendo válido después de que el valor correspondiente sea eliminado de la pila.

lua_tonumber

```
lua_Number lua_tonumber (lua_State *L, int index);
```
[-0, +0, -]

Convierte el valor Lua dado en la posicion del índice aceptable en un número (véase `lua_Number`). El valor Lua debe ser un número o un *string* convertible a número (véase §2.2.1); en caso contrario `lua_tonumber` retorna 0.

lua_topointer

```
const void *lua_topointer (lua_State *L, int index);
```
[-0, +0, -]

Convierte el valor situado en el índice aceptable en un puntero genérico de C (`void*`). El valor puede ser un *userdata*, una tabla, un proceso o una función; en caso contrario `lua_topointer` retorna `NULL`. Lua se asegura de que diferentes objetos retornen diferentes punteros. No hay una manera directa de convertir un puntero de nuevo a su valor original.

Típicamente esta función sólo es usada para información de depuración.

lua_tostring

```
const char *lua_tostring (lua_State *L, int index);
```
[-0, +0, *m*]

Equivalente a `lua_tolstring` con `len` igual a `NULL`.

lua_tothread

```
lua_State *lua_tothread (lua_State *L, int index);
```
[-0, +0, -]

Convierte el valor en la posición del índice aceptable en un proceso de Lua (representado como `lua_State*`). Este valor debe ser un proceso; en caso contrario la función retorna `NULL`.

lua_touserdata

```
void *lua_touserdata (lua_State *L, int index);
```
[-0, +0, -]

Si el valor en la posición del índice aceptable es un *userdata* completo retorna la dirección de su bloque de memoria. Si el valor es un *userdata* ligero retorna su puntero. En otro caso retorna `NULL`.

lua_type

```
int lua_type (lua_State *L, int index);
```
[-0, +0, -]

Retorna el tipo del valor situado en el índice aceptable o `LUA_TNONE` si la dirección es inválida (esto es, un índice a una posición "vacía" en la pila). Los tipos retornados por `lua_type` están codificados por las siguientes constantes, definidas en `lua.h`: `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD` y `LUA_TLIGHTUSERDATA`.

lua_typename

```
const char *lua_typename  (lua_State *L, int tp);
```
[-0, +0, -]

Retorna el nombre del tipo codificado por el valor `tp`, el cual debe ser uno de los valores retornados `lua_type`.

lua_Writer

```
typedef int (*lua_Writer) (lua_State *L,
                           const void* p,
                           size_t sz,
                           void* ud);
```

La función escritora usada por `lua_dump`. Cada vez que produce otro fragmento de *chunk*, `lua_dump` llama al escritor, pasándole el *buffer* para ser escrito (`p`), su tamaño (`sz`) y el argumento `data` proporcionado a `lua_dump`.

El escritor retorna un código de error: 0 significa no errores y cualquier otro valor significa un error y evita que `lua_dump` llame de nuevo al escritor.

lua_xmove

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```
[-?, +?, -]

Intercambia valores entre diferentes procesos del *mismo* estado global.

Esta función elimina `n` valores de la pila indicada por `from` y los coloca en la pila indicada por `to`.

lua_yield

```
int lua_yield  (lua_State *L, int nresults);
```
[-?, +?, -]

Produce la cesión de una co-rutina.

Esta función debería ser llamada solamente como expresión de retorno de una función C, como sigue:

```
   return lua_yield (L, nresults);
```

Cuando una función C llama a `lua_yield` de esta manera, la co-rutina que está ejecutándose suspende su ejecución, y la llamada a `lua_resume` que comenzó esta co-rutina retorna. El argumento `nresults` es el número de valores de la pila que son pasados como resultados a `lua_resume`.

3.8 – El interface de depuración

Lua no tiene utilidades de depuración internas. En su lugar ofrece una interface especial por medio de funciones y *hooks*. Esta interface permite la construcción de diferentes tipos de depuradores, analizadores de código y otras herramientas que necesitan "información interna" del intérprete.

lua_Debug

```
typedef struct lua_Debug {
  int event;
  const char *name;           /* (n) */
  const char *namewhat;       /* (n) */
  const char *what;           /* (S) */
  const char *source;         /* (S) */
```

```
  int currentline;            /* (l) */
  int nups;                   /* (u) number of upvalues */
  int linedefined;            /* (S) */
  int lastlinedefined;        /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* private part */
  other fields
} lua_Debug;
```

Una estructura usada para contener diferentes fragmentos de información acerca de la función activa. `lua_getstack` rellena sólo la parte privada de esta estructura, para su uso posterior. Para rellenar otros campos de `lua_debug` con información útil, llámese a `lua_getinfo`.

Los campos de `lua_debug` tienen el siguiente significado:

- **`source`:** Si la función fue definida en un *string* entonces `source` es ese *string*. Si la función fue definida en un fichero entonces `source` comienza con un carácter '`@`' seguido del nombre del fichero.
- **`short_src`:** una versión "imprimible" de `source`, que será usada en los mensajes de error.
- **`linedefined`:** el número de línea donde comienza la definición de la función.
- **`lastlinedefined`:** el número de línea donde acaba la definición de función.
- **`what`:** el *string* `"Lua"` si la función es una función Lua, `"C"` si la función es una función C, `"main"` si es la parte principal de un *chunk*, y `"tail"` si es una función que realiza una llamada de cola. Es el último caso Lua no tiene más información acerca de la función.
- **`currentline`:** la línea actual donde la función dada se está ejecutando. Cuando esta información no está disponible, `currentline` toma el valor *-1*.
- **`name`:** un nombre razonable para la función dada. Debido a que las funciones en Lua son valores de primera clase, no tienen un nombre fijo: algunas funciones pueden ser el valor de variables globales, mientras que otras pueden ser almacenadas sólo en un campo de una tabla. La función `lua_getinfo` analiza cómo fue invocada la función para encontrarle un nombre adecuado. Si no puede encontrarlo entonces `nombre` se hace `NULL`.
- **`namewhat`:** explica el campo `nombre`. El valor de `namewhat` puede ser `"global"`, `"local"`, `"method"`, `"field"`, `"upvalue"` o `""` (un *string* vacío), de acuerdo a cómo fue invocada la función. (Lua usa un *string* vacío cuando otras opciones no son idóneas.)
- **`nups`:** el numero de *upvalues* de la función.

lua_gethook

```
lua_Hook lua_gethook (lua_State *L);
```

[-0, +0, -]

Retorna la función *hook* actual.

lua_gethookcount

```
int lua_gethookcount (lua_State *L);
```

[-0, +0, -]

Retorna el contador de *hook* actual.

lua_gethookmask

```
int lua_gethookmask (lua_State *L);
```

[-0, +0, -]

Retorna la máscara del *hook* actual.

lua_getinfo

`int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);` [-(0|1), +(0|1|2), *m*]

Devuelve información acerca de una función específica o de una invocación de función.

Para obtener información acerca de una invocación de función, el parámetro `ar` debe ser un registro de activación válido, que haya sido llenado con una llamada previa a `lua_getstack` o dada como argumento a un *hook* (véase `lua_Hook`).

Para obtener información de una función se coloca la misma en la parte superior de la pila y se comienza el *string* `what` con el carácter '>'. (En ese caso, `lua_getinfo` elimina la función de la parte superior de la pila.) Por ejemplo, para conocer en qué línea fue definida una función `f` se puede utilizar el siguiente código:

```
lua_Debug ar;
lua_getfield(L, LUA_GLOBALSINDEX, "f");  /* obtiene la 'f' global */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

Cada carácter en el *string* `what` selecciona los campos de la estructura `ar` que serán rellenados o un valor que será colocado en la parte superior de la pila:

- **'n':** rellena los campos `name` y `namewhat`;
- **'S':** rellena los campos `source`, `short_src`, `linedefined`, `lastlinedefined` y `what`;
- **'l':** rellena el campo `currentline`;
- **'u':** rellena el campo `nups`;
- **'f':** coloca en la pila la función que está ejecutándose al nivel dado;
- **'L':** coloca en la pila una tabla cuyos índices son los números de las líneas que son válidas en la función. (Una *línea válida* es una línea con algún código asociado, esto es, una línea donde se puede poner un punto de rotura. Las líneas no válidas incluyen líneas vacías y comentarios.

Esta función devuelve 0 en caso de error (por ejemplo, una opción inválida en `what`).

lua_getlocal

`const char *lua_getlocal (lua_State *L, lua_Debug *ar, int n);` [-0, +(0|1), -]

Obtiene información acerca de una variable local de un registro de activación dado. El argumento `ar` debe ser un registro de activación válido que fue rellenado en una llamada previa a `lua_getstack` o dado como argumento a un *hook* (véase `lua_Hook`). El índice `n` selecciona qué variable local inspeccionar (1 es el primer argumento o la primera variable local activa, y así sucesivamente, hasta la última variable local activa). `lua_getlocal` coloca el valor de la variable en la pila y retorna su nombre.

Los nombres de variable que comienzan con '(' (paréntesis de abrir) representan variables internas (variables de control de bucle, variables temporales y variables locales de funciones C).

Retorna `NULL` (y no coloca nada en la pila) cuando el índice es mayor que el número de variables locales activas.

lua_getstack

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```
[-0, +0, -]

Obtiene información acerca de la pila en ejecución del intérprete.

Esta función rellena partes de una estructura `lua_debug` con una identificación del *registro de activación* de la función que se está ejecutando al nivel dado. Nivel 0 es la función actualmente en ejecución, mientras que el nivel *n+1* es la función que ha invocado a la del nivel *n*. Cuando no hay errores, `lua_getstack` retorna 1; cuando se llama con un nivel mayor que el tamaño de la pila retorna 0.

lua_getupvalue

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
```
[-0, +(0|1), -]

Obtiene información acerca de un *upavalue* de una instancia. (Para las funciones Lua los *upvalues* son variables locales externas a la función que las usa, y que, por consiguiente, están incluidas en su instancia.) `lua_getupvalue` obtiene el índice `n` de un *upvalue*, coloca su valor en la pila y retorna su nombre. `funcindex` apunta hacia la instancia en la pila. (Los *upvalues* no siguen un orden particular, puesto que están activos a lo largo de toda la función. Por tanto, están numerados siguiendo un orden arbitrario.)

Retorna `NULL` (y no coloca nada en la pila) cuando el índice es mayor que el número de *upvalues*. Para funciones C esta función usa el *string* vacío `""` como nombre para todos los *upvalues*.

lua_Hook

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

Tipo para funciones *hook* de depuración.

Cada vez que se invoca un *hook* su argumento `ar` tiene en su campo `event` el evento que ha activado el *hook*. Lua identifica estos eventos con las siguientes constantes: `LUA_HOOKCALL`, `LUA_HOOKRET`, `LUA_HOOKTAILRET`, `LUA_HOOKLINE` y `LUA_HOOKCOUNT`. Además, para eventos de línea, también se establece el campo `currentline`. Para obtener el valor de algún otro campo en `ar`, el *hook* debe invocar a `lua_getinfo`. Para eventos de retorno, `event` puede ser `LUA_HOOKRET`, el valor normal, o `LUA_HOOKTAILRET`. En el último caso, Lua está simulando un retorno de una función que ha hecho una llamada de cola; en este caso, es inútil llamar a `lua_getinfo`.

Mientras Lua está ejecutando un *hook*, deshabilita otras llamadas a *hooks*. Por tanto, si un *hook* llama de nuevo a Lua para ejecutar una función o un *chunk* entonces esa ejecución ocurre sin ninguna llamada a *hooks*.

lua_sethook

```
int lua_sethook (lua_State *L, lua_Hook f, int mask, int count);
```
[-0, +0, -]

Establece la función *hook* de depuración.

`func` es la función *hook*. `mask` especifica en qué eventos debe ser llamado el *hook*: se forma mediante la operación "or" aplicada a los bits de las constantes `LUA_MASKCALL`, `LUA_MASKRET`, `LUA_MASKLINE`, y `LUA_MASKCOUNT`. El argumento `count` sólo tiene sentido cuando la máscara incluye `LUA_MASKCOUNT`. Para cada evento, el *hook* es invocado como se explica a continuación:

- **El hook tipo "call"** es invocado cuando el intérprete llama a una función. El *hook* es invocado justo después de que Lua entre en la nueva función, antes de que la función tome sus argumentos.
- **El hook de tipo "return"** es invocado cuando el intérprete retorna desde una función. El *hook* es invocado justo antes de que Lua deje la función. No se tiene acceso a los valores retornados por la función.
- **El hook tipo "line"** es invocado cuando el intérprete va a comenzar la ejecución de una nueva línea de código, o cuando salta hacia atrás en el código (incluso en la misma línea). (Este evento sólo ocurre cuando Lua está ejecutando una función Lua.)
- **El hook tipo "count"** es invocado después de que el intérprete ejecute un número de instrucciones igual a `count`. (Este evento sólo ocurre cuando Lua está ejecutando una función Lua.)

Un *hook* se deshabilita estableciendo `mask` a cero.

lua_setlocal

```
const char *lua_setlocal (lua_State *L, lua_Debug *ar, int n);
```
[-(0|1), +0, -]

Establece el valor de una variable local de un registro de activación dado. Los argumentos `ar` y `n` son como los de `lua_getlocal`. `lua_setlocal` asigna el valor en la parte superior de la pila a la variable y retorna su nombre. También elimina de la pila su valor.

Retorna `NULL` (y no hace nada con la pila) cuando el índice es mayor que el número de variables locales activas.

lua_setupvalue

```
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```
[-(0|1), +0, -]

Establece el valor de un *upvalue* de una instancia. Los argumentos `funcindex` y `n` son como los de `lua_getupvalue`. Asigna el valor que está en la parte superior de la pila al *upvalue* y retorna su nombre. También elimina de la pila el valor.

Retorna `NULL` (y no hace nada con la pila) cuando el índice es mayor que el número de *upavalues*.

4 – La biblioteca auxiliar

La *biblioteca auxiliar* proporciona varias funciones convenientes para realizar la interface de C con Lua. Mientras que la API básica proporciona las funciones primitivas para todas las interaciones entre C y Lua, la biblioteca auxiliar proporciona funciones de alto nivel para algunas tareas comunes.

Todas las funciones de la biblioteca auxiliar están definidas en el fichero de cabecera `lauxlib.h` y llevan el prefijo `luaL_`.

Todas ellas están construidas encima de la API básica así que realmente no proporcionan nada nuevo que no pueda ser realizado con la API.

Algunas funciones en la biblioteca auxiliar son usadas para verificar argumentos de funciones C. Sus nombres son siempre `luaL_check*` o `luaL_opt*`. Todas estas funciones activan un error si la

verificación no se satisface. Debido a que el mensaje de error se formatea para los argumentos (por ejemplo, `"bad argument #1"`), no se deberían usar estas funciones para otros valores de la pila.

4.1 – Funciones y tipos

Aquí tenemos la lista de todas las funciones y tipos de la biblioteca auxiliar por orden alfabético.

luaL_addchar

```
void luaL_addchar (luaL_Buffer *B, char c);
```

[-0, +0, *m*]

Añade el carácter `c` al *buffer* `B` (véase `luaL_Buffer`).

luaL_addlstring

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```

[-0, +0, *m*]

Añade el *string* al que apunta `s` con longitud `l` al *buffer* `B` (véase `luaL_Buffer`). El *string* puede contener ceros.

luaL_addsize

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```

[-0, +0, *m*]

Añade un *string* de longitud `n` previamente copiado en el área del *buffer* (véase `luaL_prepbuffer`) al *buffer* `B` (véase `luaL_Buffer`).

luaL_addstring

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```

[-0, +0, *m*]

Añade un *string* terminado en cero al que apunta `s` al *buffer* `B` (véase `luaL_Buffer`). El *string* no puede contener ceros.

luaL_addvalue

```
void luaL_addvalue (luaL_Buffer *B);
```

[-1, +0, *m*]

Añade el valor situado en la parte superior de la pila al *buffer* `B` (véase `luaL_Buffer`), eliminándolo de la pila.

Ésta es la única función asociada a los *buffers* de *string* que puede (y debe) ser invocada con un elemento extra en la pila, que es el valor que debe ser añadido al *buffer*.

luaL_argcheck

```
void luaL_argcheck (lua_State *L,
                    int cond,
                    int narg,
                    const char *extramsg);
```

[-0, +0, *v*]

Verifica si `cond` es verdadero. Si no es así activa un error con el mensaje

```
"bad argument #<numarg> to <func> (<extramsg>)"
```

donde `func` es recuperado de la pila de llamada.

luaL_argerror

```
int luaL_argerror (lua_State *L, int narg, const char *extramsg);
```
[-0, +0, *v*]

Activa un error con el mensaje

```
"bad argument #<numarg> to <func> (<extramsg>)"
```

donde `func` es recuperado de la pila de llamada.

Esta función nunca retorna, pero es corriente usarla en funciones C en la forma `return luaL_argerror(args)`.

luaL_Buffer

```
typedef struct luaL_Buffer luaL_Buffer;
```

Tipo para un *buffer* de *string*.

Un *buffer* de *string* permite al código en C construir a trozos *strings* de Lua. Su metodología de uso es como sigue:

- Primero se declara una variable `b` de tipo `luaL_Buffer`.
- Luego se inicializa la misma con una llamada a `luaL_buffinit(L, &b)`.
- Entonces se añaden las piezas del *string* al *buffer*, invocando alguna de las funciones `luaL_add*`.
- Se finaliza llamando a `luaL_pushresult(&b)`. Esta llamada deja el *string* final en la parte superior de la pila.

Durante su operación normal, un *buffer* de *strings* usa un número variable de posiciones en la pila. Así, mientras se está usando el *buffer*, no se puede asumir que se conoce la posición de la parte superior de la pila. Se puede usar la pila entre llamadas sucesivas a las operaciones de *buffer* siempre que su uso esté *equilibrado*; esto es, cuando se invoca una operación con el *buffer*, la pila está al mismo nivel en el que estaba inmediatamente antes de la operación previa con el *buffer*. (La única excepción a esta regla es `luaL_addvalue`.) Después de llamar a `luaL_pushresult` la pila está de nuevo en el mismo nivel que tenía cuando el *buffer* fue inicializado, más el *string* final en su parte superior.

luaL_buffinit

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
```
[-0, +0, *e*]

Inicializa un *buffer* `B`. Esta función no reserva ningún espacio nuevo de memoria; el *buffer* debe ser declarado como variable (véase `luaL_Buffer`).

luaL_callmeta

[-0, +(0|1), *e*]

```
int luaL_callmeta (lua_State *L, int obj, const char *e);
```

Invoca un metamétodo.

Si el objeto con índice `obj` tiene una metatabla y ésta tiene un campo `e`, esta función llama a este campo y le pasa el objeto como único argumento. En este caso la función retorna 1 y coloca en la pila el valor devuelto por la llamada. Si no hay metatabla o no hay metamétodo la función retorna 0 (sin colocar ningún valor en la pila).

luaL_checkany

```
void luaL_checkany (lua_State *L, int narg);
```
[-0, +0, *v*]

Verifica si la función tiene un argumento de algún tipo (incluyendo **nil**) en la posición `narg`.

luaL_checkint

```
int luaL_checkint (lua_State *L, int narg);
```
[-0, +0, *v*]

Verifica si el argumento `narg` de la función es un número y retorna este número como `int` (realizando un *cast* en C).

luaL_checkinteger

```
lua_Integer luaL_checkinteger (lua_State *L, int narg);
```
[-0, +0, *v*]

Verifica si el argumento `narg` de la función es un número y lo retorna como tipo `lua_Integer`.

luaL_checklong

```
long luaL_checklong (lua_State *L, int narg);
```
[-0, +0, *v*]

Verifica si el argumento `narg` de la función es un número y lo retorna como `long` (realizando un *cast* en C).

luaL_checklstring

```
const char *luaL_checklstring (lua_State *L, int narg, size_t *l);
```
[-0, +0, *v*]

Verifica si el argumento `narg` de la función es un *string* y retorna el mismo; si `l` no es `NULL` coloca la longitud del *string* en `*l`.

Esta función usa `lua_tolstring` para obtener su resultado, por lo que todas las conversiones y precauciones asociados a esa función se aplican aquí.

luaL_checknumber

```
lua_Number luaL_checknumber (lua_State *L, int narg);
```
[-0, +0, *v*]

Verifica si el argumento `narg` de la función es un número y retorna el mismo.

luaL_checkoption

```
int luaL_checkoption (lua_State *L,
                      int narg,
                      const char *def,
                      const char *const lst[]);
```

[-0, +0, *v*]

Verifica si el argumento `narg` de la función es un *string* y busca éste en el *array* `lst` (que debe estar terminado con NULL). Retorna el índice en el *array* donde se encontró el *string*. Activa un error si el argumento no es un *string* o si no pudo ser encontrado el *string*.

Si `def` no es `NULL`, se usa `def` como valor por defecto cuando la función no tiene un argumento `narg` o si este argumento es **nil**.

Ésta es una función útil para hacer corresponder *strings* con enumeraciones de C. La convención normal en las bibliotecas de Lua es usar *strings* en lugar de números para seleccionar opciones.

luaL_checkstack

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

[-0, +0, *v*]

Incrementa el tamaño de la pila a `top + sz` elementos, activando un error si la pila no puede crecer hasta ese tamaño. `msg` es un texto adicional que iría en el mensaje de error.

luaL_checkstring

```
const char *luaL_checkstring (lua_State *L, int narg);
```

[-0, +0, *v*]

Verifica si el argumento `narg` de la función es un *string* y retorna éste.

Esta función usa `lua_tolstring` para obtener su resultado, por lo que todas las conversiones y precauciones asociados a esa función se aplican aquí.

luaL_checktype

```
void luaL_checktype (lua_State *L, int narg, int t);
```

[-0, +0, *v*]

Verifica si el argumento `narg` de la función tiene tipo `t`.

luaL_checkudata

```
void *luaL_checkudata (lua_State *L, int narg, const char *tname);
```

[-0, +0, *v*]

Verifica si el argumento `narg` de la función es un *userdata* del tipo `tname` (véase `luaL_newmetatable`).

luaL_dofile

```
int luaL_dofile (lua_State *L, const char *filename);
```

[-0, +?, *m*]

Carga y ejecuta el fichero dado. Está definida en una macro:

```
(luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

Devuelve 0 si no hay errores ó 1 en caso de error.

luaL_dostring

```
int luaL_dostring (lua_State *L, const char *str);
```
[-0, +?, *m*]

Carga y ejecuta el *string* dado. Está definida en una macro:

```
(luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

Devuelve 0 si no hay errores ó 1 en caso de error.

luaL_error

```
int luaL_error (lua_State *L, const char *fmt, ...);
```
[-0, +0, *v*]

Activa un error. El formato del mensaje está dado por `fmt` más cualesquiera argumentos extra, siguiendo las mismas reglas de `lua_pushfstring`. También añade al principio del mensaje el nombre del fichero y el número de línea donde ocurrió el error, si esta información está disponible.

Esta función nunca retorna, pero es corriente usarla en la forma `return luaL_error(args)` en funciones C.

luaL_getmetafield

```
int luaL_getmetafield (lua_State *L, int obj, const char *e);
```
[-0, +(0|1), *m*]

Coloca en la parte superior de la pila el campo `e` de la metatabla del objeto situado en la posición del índice `obj`. Si el objeto no tiene metatabla o si el objeto no tiene este campo retorna 0 y deja la pila intacta.

luaL_getmetatable

```
void luaL_getmetatable (lua_State *L, const char *tname);
```
[-0, +1, -]

Coloca en la parte superior de la pila la metatabla asociada con el nombre `tname` en el registro (véase `luaL_newmetatable`).

luaL_gsub

```
const char *luaL_gsub (lua_State *L,
                       const char *s,
                       const char *p,
                       const char *r);
```
[-0, +1, *m*]

Crea una copia del *string* `s` reemplazando cualquier aparición del *string* `p` por el *string* `r`. Coloca el *string* resultante en la parte superior de la pila y devuelve su valor.

luaL_loadbuffer

```
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
                     size_t sz,
                     const char *name);
```
[-0, +1, *m*]

Carga un *buffer* como *chunk* de Lua. Esta función usa `lua_load` para cargar el *chunk* en el *buffer* apuntado por `buff` con tamaño `sz`.

Esta función retorna el mismo resultado que `lua_load`. `name` es el nombre del *chunk*, usado para información de depuración y en los mensajes de error.

luaL_loadfile

```
int luaL_loadfile (lua_State *L, const char *filename);
```
[-0, +1, *m*]

Carga un fichero como *chunk* de Lua. Esta función usa `lua_load` para cargar el *chunk* que está en el fichero `filename`. Si `filename` es `NULL` entonces se carga desde la entrada estándar. La primera línea en el fichero se ignora si comienza por `#`.

Esta función retorna el mismo resultado que `lua_load`, pero tiene un código extra de error `LUA_ERRFILE` si no puede leer o abrir el fichero.

Como `lua_load` esta función sólo carga el *chunk* y no lo ejecuta.

luaL_loadstring

```
int luaL_loadstring (lua_State *L, const char *s);
```
[-0, +1, *m*]

Carga un *string* como *chunk* de Lua. Esta función usa `lua_load` para cargar el *chunk* que está en el *string* `s` terminado en un carácter cero.

Esta función retorna el mismo resultado que `lua_load`.

Como `lua_load` esta función sólo carga el *chunk* y no lo ejecuta.

luaL_newmetatable

```
int luaL_newmetatable (lua_State *L, const char *tname);
```
[-0, +1, *m*]

Si el registro tiene ya una clave `tname` retorna 0. En caso contrario crea una nueva tabla que será usada como metatabla del *userdata*, añadiendo la clave `tname` al registro, y retornando 1.

En ambos caso coloca en la parte superior de la pila el valor final asociado con `tname` en el registro.

luaL_newstate

```
lua_State *luaL_newstate (void);
```
[-0, +0, -]

Crea un nuevo estado de Lua, invocando `lua_newstate` con una función de reserva de memoria basada en la función C estándar `realloc` y estableciendo una función de "pánico" (véase `lua_atpanic`) que imprime un mensaje en la salida estándar de error en caso de error fatal.

Retorna el nuevo estado o `NULL` si surgió un error de reserva de memoria.

luaL_openlibs

```
void luaL_openlibs (lua_State *L);
```
[-0, +0, *m*]

Abre todas las bibliotecas estándar de Lua en el estado dado.

luaL_optint

```
int luaL_optint (lua_State *L, int narg, int d);
```
[-0, +0, *v*]

Si el argumento `narg` de la función es un número retorna éste como un `int`. Si este argumento está ausente o es **nil** retorna `d`. En otro caso activa un error.

luaL_optinteger

```
lua_Integer luaL_optinteger (lua_State *L,
                             int narg,
                             lua_Integer d);
```
[-0, +0, *v*]

Si el argumento `narg` de la función es un número retorna el mismo como `lua_Integer`. Si este argumento está ausente o es **nil** retorna `d`. En caso contrario activa un error.

luaL_optlong

```
long luaL_optlong (lua_State *L, int narg, long d);
```
[-0, +0, *v*]

Si el argumento `narg` de la función es un número retorna el mismo como `long`. Si este argumento está ausente o es **nil** retorna `d`. En caso contrario activa un error.

luaL_optlstring

```
const char *luaL_optlstring (lua_State *L,
                             int narg,
                             const char *d,
                             size_t *l);
```
[-0, +0, *v*]

Si el argumento `narg` de la función es un *string* retorna éste. Si este argumento está ausente o es **nil** retorna `d`. En caso contrario activa un error.

Si `l` no es `NULL` coloca la longitud del resultado en `*l`.

luaL_optnumber

```
lua_Number luaL_optnumber (lua_State *L, int narg, lua_Number d);
```
[-0, +0, *v*]

Si el argumento `narg` de la función es un número retorna el mismo. Si este argumento está ausente o es **nil** retorna `d`. En caso contrario activa un error.

luaL_optstring

```
const char *luaL_optstring (lua_State *L,
                            int narg,
                            const char *d);
```
[-0, +0, *v*]

Si el argumento `narg` de la función es un *string* retorna éste. Si este argumento está ausente o es **nil** retorna `d`. En caso contrario activa un error.

luaL_prepbuffer

```
char *luaL_prepbuffer (luaL_Buffer *B);
```
[-0, +0, -]

Retorna una dirección que apunta a un espacio de tamaño `LUAL_BUFFERSIZE` donde se puede copiar un *string* para ser añadido al *buffer* `B` (véase `luaL_Buffer`). Después de copiar el *string* en este espacio se debe invocar `luaL_addsize` con el tamaño del *string* para añadirlo realmente en el *buffer*.

luaL_pushresult

```
void luaL_pushresult (luaL_Buffer *B);
```
[-?, +1, *m*]

Finaliza el uso del *buffer* `B` dejando el *string* en la parte superior de la pila.

luaL_ref

```
int luaL_ref (lua_State *L, int t);
```
[-1, +0, *m*]

Crea y retorna una *referencia* en la tabla en la posición del índice `t` para el objeto en la parte superior de la pila (y elimina el mismo de la pila).

Una referencia es una clave entera única. Mientras que no se añadan manualmente claves enteras a la tabla `t`, `luaL_ref` asegura la unicidad de la clave que retorna. Se puede recuperar un objeto apuntado por la referencia `r` invocando `lua_rawgeti(L, t, r)`. La función `luaL_unref` elimina una referencia y su objeto asociado.

Si el objeto en la parte superior de la pila es **nil**, `luaL_ref` retorna la constante `LUA_REFNIL`. Está garantizado que la constante `LUA_NOREF` es diferente de cualquier referencia retornada por `luaL_ref`.

luaL_Reg

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

Tipo para *arrays* de funciones para ser registradas por `luaL_register`. `name` es el nombre de la función y `func` es un puntero a la misma. Cualquier *array* de `luaL_Reg` debe finalizar con una entrada "centinela" en la que tanto `name` como `func` son `NULL`.

luaL_register

```
void luaL_register (lua_State *L,
                    const char *libname,
                    const luaL_Reg *l);
```
[-(0|1), +1, *m*]

Abre una biblioteca.

Cuando se llama con `libname` igual a `NULL` simplemente registra todas las funciones de la lista `l` (véase `luaL_Reg`) en la tabla que está en la parte superior de la pila.

Cuando se llama con un valor `libname` no nulo crea una nueva tabla `t`, establece la misma como valor de la variable global `libname`, establece la misma como valor de `package.loaded[libname]` y registra en ella todas las funciones de la lista `l`. Si existe una tabla en `package.loaded[libname]` o en la variable `libname` reutiliza esta tabla en lugar de crear una nueva.

En cualquier caso la función deja la tabla en la parte superior de la pila.

luaL_typename

```
const char *luaL_typename (lua_State *L, int index);
```
[-0, +0, -]

Retorna el nombre del tipo del valor situado en el índice dado.

luaL_typerror

```
int luaL_typerror (lua_State *L, int narg, const char *tname);
```
[-0, +0, *v*]

Genera un error con un mensaje de la forma:

```
location: bad argument narg to function (tname expected, got rt)
```

donde *location* está producida por `luaL_where`, *function* es el nombre de la función actual y *rt* es el nombre del tipo del argumento actual.

luaL_unref

```
void luaL_unref (lua_State *L, int t, int ref);
```
[-0, +0, -]

Libera la referencia `ref` de la tabla en el índice `t` (véase `luaL_ref`). La entrada es eliminada de la tabla, por lo que la memoria reservada para el objeto referido en la misma puede ser liberada. La referencia `ref` también es liberada para poder ser reutilizada.

Si `ref` es `LUA_NOREF` o `LUA_REFNIL`, `luaL_unref` no hace nada.

luaL_where

```
void luaL_where (lua_State *L, int lvl);
```
[-0, +1, *m*]

Coloca en la parte superior de la pila un *string* identificando la posición actual del control en el nivel `lvl` en la pila de llamada. Típicamente este *string* tiene el formato:

```
chunkname:currentline:
```

Nivel 0 es la función actualmente ejecutándose, nivel 1 es la función que llamó a la función actual, etc.

Esta función se usa para construir un prefijo para los mensajes de error.

5 – Bibliotecas estándar

Las bibliotecas estándar de Lua proporcionan funciones útiles que están implementadas directamente a través de la API de C. Algunas de estas funciones proveen servicios esenciales al lenguaje (por ejemplo, `type` y `getmetatable`); otras proporcionan acceso a servicios "externos" (por ejemplo, I/O); y otras podrían ser implementadas en Lua mismo pero son muy útiles o tienen requerimientos críticos de tiempo de ejecución y merecen una implementación en C (por ejemplo, `sort`).

Todas las bibliotecas están implementadas a través de la API oficial de C y se proporcionan como módulos separados en C. En estos momentos Lua tiene las siguientes bibliotecas estándar:

- biblioteca básica;
- biblioteca de empaquetado;
- manejo de *strings*;
- manejo de tablas;
- funciones matemáticas (sin, log, etc.);
- entrada y salida (I/O);
- interacción con el sistema operativo;
- utilidades de depuración.

Excepto para las bibliotecas básica y de empaquetado, cada biblioteca proporciona todas sus funciones como campos de tablas globales o como métodos de sus objetos.

Para tener acceso a estas bibliotecas el programa anfitrión en C debe invocar a `luaL_openlibs`, la cual abre todas las bibliotecas estándar. De manera alternativa se pueden abrir individualmente invocando a `luaopen_base` (la biblioteca básica), `luaopen_package` (la biblioteca de empaquetado), `luaopen_string` (la biblioteca de *strings*), `luaopen_table` (la biblioteca de tablas), `luaopen_math` (la biblioteca matemática), `luaopen_io` (la biblioteca de entrada/salida), `luaopen_os` (la biblioteca del Sistema Operativo) y `luaopen_debug` (la biblioteca de depuración). Estas funciones están declaradas en `lualib.h` y no deberían ser invocadas directamente: se deben llamar como a otra función C cualquiera de Lua, por ejemplo, usando `lua_call`.

5.1 – Funciones básicas

La biblioteca básica proporciona algunas funciones del núcleo de Lua. Si no se desea incluir esta biblioteca en una aplicación se debe analizar cuidadosamente si se necesitan proporcionar implementaciones de algunas de sus utilidades.

assert (v [, mensaje])

Activa un error cuando el valor de su argumento `v` es falso (por ejemplo, **nil** o **false**); en otro caso retorna todos sus argumentos. `mensaje` es un mensaje de error; cuando está ausente se utiliza por defecto `"assertion failed!"`.

collectgarbage (opt [, arg])

Esta función es una interface genérica al liberador de memoria. Realiza diversas funciones de acuerdo a su primer argumento, `opt`:

- **"stop":** detiene el liberador de memoria.
- **"restart":** reinicia el liberador de memoria.
- **"collect":** realiza un ciclo completo de liberación de memoria.
- **"count":** devuelve la memoria total en uso por Lua (en Kbytes).

- **"step":** realiza un paso de liberación de memoria. El "tamaño" del paso se controla por `arg` (valores grandes significan más pasos) de una manera no especificada. Si se desea controlar el tamaño del paso se debe afinar experimentalmente el valor de `arg`. Devuelve **true** si el paso acaba un ciclo de liberación.
- **"steppause":** establece `arg`/100 como el nuevo valor para la *pausa* del liberador (véase §2.10).
- **"setstepmul":** establece `arg`/100 como el nuevo valor para el *multiplicador del paso* del liberador (véase §2.10).

dofile (nombre_de_fichero)

Abre el fichero con el nombre dado y ejecuta su contenido como un *chunk* de Lua. Cuando se invoca sin argumentos, `dofile` ejecuta el contenido de la entrada estándar (`stdin`). Devuelve todos los valores retornados por el *chunk*. En caso de error, `dofile` propaga el error a su invocador (esto es, `dofile` no se ejecuta en modo protegido).

error (mensaje [, nivel])

Termina la última función protegida llamada, estableciendo `mensaje` como mensaje de error. La función `error` nunca retorna.

Normalmente `error` añade, al comienzo del mensaje, cierta información acerca de la posición del error. El argumento `nivel` especifica cómo obtener la posición del error. Con nivel 1 (por defecto) la posición del error es donde fue invocada la función `error`. Nivel 2 apunta el error hacia el lugar en que fue invocada la función que llamó a `error`; y así sucesivamente. Pasar un valor 0 como nivel evita la adición de la información de la posición al mensaje.

_G

Una variable global (no una función) que almacena el entorno global (o sea, `_G._G = _G`). Lua mismo no usa esta variable; cambiar su valor no afecta ningún entorno, ni viceversa. (Úsese `setfenv` para cambiar entornos.)

getfenv ([f])

Retorna el entorno actualmente en uso por la función. `f` puede ser una función Lua o un número que especifica la función a ese nivel de la pila: nivel 1 es la función que invoca a `getfenv`. Si la función dada no es una función Lua o si `f` es 0, `getfenv` retorna el entorno global. El valor por defecto de `f` es 1.

getmetatable (objeto)

Si `objeto` no tiene una metatabla devuelve **nil**. En otro caso, si la metatabla del objeto tiene un campo `"__metatable"` retorna el valor asociado, o si no es así retorna la metatabla del objeto dado.

ipairs (t)

Retorna tres valores: una función iteradora, la tabla `t`, y 0, de tal modo que la construcción

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

iterará sobre los pares (`1,t[1]`), (`2,t[2]`), ···, hasta la primera clave entera con un valor nil en la tabla.

load (func [, nombre_de_chunk])

Carga un *chunk* usando la función `func` para obtener sus partes. Cada llamada a `func` debe retornar un *string* que se concatena con los resultados previos. Un retorno de **nil** (o no valor) señala el final del *chunk*.

Si no hay errores retorna el *chunk* compilado como una función; en otro caso retorna **nil** más un mensaje de error. El entorno de la función retornada es el global.

`nombre_de_chunk` se utiliza para identificar el *chunk* en los mensajes de error y para información de depuración.

loadfile ([nombre_de_fichero])

Similar a `load`, pero obtiene el *chunk* del fichero `nombre_de_fichero` o de la entrada estándar si no se proporciona un nombre.

loadstring (string [, nombre_de_chunk])

Similar a `load`, pero obtiene el *chunk* del *string* proporcionado.

Para cargar y ejecutar un *string* dado úsese

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

Cuando está ausente, `nombre_de_chunk` toma por defecto el *string* dado.

next (tabla [, índice])

Permite al programa recorrer todos los campos de una tabla. Su primer argumento es una tabla y su segundo argumento es un índice en esta tabla. `next` retorna el siguiente índice de la tabla y su valor asociado. Cuando se invoca con **nil** como segundo argumento `next` retorna un índice inicial y su valor asociado. Cuando se invoca con el último índice o con **nil** en una tabla vacía `next` retorna **nil**. Si el segundo argumento está ausente entonces se interpreta como **nil**. En particular se puede usar `next(t)` para comprobar si una tabla está vacía.

El orden en que se enumeran los índices no está especificado, *incluso para índices numéricos*. (Para recorrer una tabla en orden numérico úsese el **for** numérico o la función `ipairs`.)

El comportamiento de `next` es *indefinido* si durante el recorrido se asigna un valor a un campo no existente previamente en la tabla. No obstante se pueden modificar campos existentes. En particular se pueden borrar campos existentes.

pairs (t)

Retorna tres valores: la función `next`, la tabla `t`, y **nil**, por lo que la construcción

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

iterará sobre todas las parejas clave-valor de la tabla `t`.

Véase `next` para las precauciones a tomar cuando se modifica la tabla durante las iteraciones.

pcall (f, arg1, ···)

Invoca la función `f` con los argumentos dados en modo protegido. Esto significa que ningún error dentro de `f` se propaga; en su lugar `pcall` captura el error y retorna un código de estatus. Su primer resultado es el código de estatus (booleano), el cual es verdadero si la llamada tiene éxito sin errores. En ese caso `pcall` también devuelve todos los resultados de la llamada después del primer resultado. En caso de error `pcall` retorna **false** más un mensaje de error.

print (···)

Recibe cualquier número de argumentos e imprime sus valores en el fichero estándar de salida (`stdout`), usando `tostring` como función para convertir los argumentos a *strings*. `print` no está diseñada para salida formateada sino sólo como una manera rápida de mostrar valores, típicamente para la depuración del código. Para salida formateada úsese `string.format`.

rawequal (v1, v2)

Verifica si `v1` es igual a `v2`, sin invocar ningún metamétodo. Devuelve un booleano.

rawget (tabla, índice)

Obtiene el valor real de `tabla[índice]` sin invocar ningún metamétodo. `tabla` debe ser una tabla e `índice` cualquier valor diferente de **nil**.

rawset (tabla, índice, valor)

Asigna `valor` a `tabla[índice]` sin invocar ningún metamétodo. `tabla` debe ser una tabla, `índice` cualquier valor diferente de **nil** y `valor` un valor cualquiera de Lua.

select (índice, ···)

Si `índice` es un número retorna todos los argumentos después del número `índice`. En otro caso `índice` debe ser el *string* `"#"`, y `select` retorna el número total de argumentos extra que recibe.

setfenv (f, tabla)

Establece el entorno que va a ser usado por una función. `f` puede ser una función Lua o un número que especifica la función al nivel de pila: nivel 1 es la función que invoca a `setfenv`. `setfenv` retorna la función dada.

Como caso especial, cuando `f` es 0 `setfenv` cambia el entorno del proceso que está en ejecución. En este caso `setfenv` no retorna valores.

setmetatable (tabla, metatabla)

Establece la metatabla de una tabla dada. (No se puede cambiar la metatabla de otros tipos desde Lua, sino sólo desde C.) Si `metatabla` es **nil** entonces se elimina la metatabla de la tabla dada. Si la metatabla original tiene un campo `"__metatable"` se activa un error.

Esta función retorna `tabla`.

tonumber (e [, base])

Intenta convertir su argumento en un número. Si el argumento es ya un número o un *string* convertible a un número entonces `tonumber` retorna este número; en otro caso devuelve **nil**.

Un argumento opcional especifica la base para interpretar el número. La base puede ser cualquier entero entre 2 y 36, ambos inclusive. En bases por encima de 10 la letra '`A`' (en mayúscula o minúscula) representa 10, '`B`' representa 11, y así sucesivamente, con '`Z`' representando 35. En base 10 (por defecto), el número puede tener parte decimal, así como un exponente opcional (véase §2.1). En otras bases sólo se aceptan enteros sin signo.

tostring (e)

Recibe un argumento de cualquier tipo y lo convierte en un *string* con un formato razonable. Para un control completo de cómo se convierten los números, úsese `string.format`.

Si la metatabla de `e` tiene un campo "`__tostring`" entonces `tostring` invoca al correspondiente valor con `e` como argumento y usa el resultado de la llamada como su propio resultado.

type (v)

Retorna el tipo de su único argumento, codificado como *string*. Los posibles resultados de esta función son "`nil`" (un string, no el valor **nil**), "`number`", "`string`", "`boolean`, "`table`", "`function`", "`thread`" y "`userdata`".

unpack (lista [, i [, j]])

Retorna los elementos de una tabla dada. Esta función equivale a

```
return lista[i], lista[i+1], ···, lista[j]
```

excepto que este código puede ser escrito sólo para un número fijo de elementos. Por defecto `i` es 1 y `j` es la longitud de la lista, como se define a través del operador longitud (véase §2.5.5).

_VERSION

Una variable global (no una función) que almacena un *string* que contiene la versión actual del intérprete. En esta versión de Lua el contenido actual de esta variable es "`Lua 5.1`".

xpcall (f, err)

Esta función es similar a `pcall`, excepto que se puede establecer un manejador de error.

`xpcall` invoca a la función `f` en modo protegido, usando `err` como manejador de error. Ningún error dentro de `f` se propaga; en su lugar `xpcall` captura el error, llamando a la función `err` con el objeto de error original, y retorna un código de estatus. Su primer resultado es el código de estatus (un booleano), que es verdadero si la llamada tiene éxito sin errores. En ese caso `xpcall` también devuelve todos los resultados de la llamada después del primer resultado. En caso de error `xpcall` retorna **false** más el resultado de `err`.

5.2 – Manejo de co-rutinas

Las operaciones relacionadas con co-rutinas comprenden una sub-biblioteca de la biblioteca básica y se sitúa en la tabla `coroutine`. Véase §2.11 para una descripción general de las co-rutinas.

coroutine.create (f)

Crea una nueva co-rutina con cuerpo `f`. `f` debe ser una función Lua. Retorna una nueva co-rutina, un objeto de tipo `"thread"`.

coroutine.resume (co [, val1, ···])

Comienza o continúa la ejecución de la co-rutina `co`. La primera vez que se llama a esta función la co-rutina comienza ejecutando su cuerpo. Los valores `val1`, ··· se pasan como argumentos al cuerpo de la función. Si la co-rutina ha cedido el control del flujo, `resume` la reinicia; los valores `val1`, ··· son pasados como resultados de la cesión.

Si la co-rutina se ejecuta sin error `resume` retorna **true** más los valores pasados a `yield` (si la co-rutina realiza la cesión) o los valores retornados por el cuerpo de la función (si la co-rutina acaba). Si existe cualquier error `resume` retorna **false** más un mensaje de error.

coroutine.running ()

Retorna la co-rutina en ejecución o **nil** cuando se invoca desde el proceso principal.

coroutine.status (co)

Retorna el estatus de la co-rutina `co` como un *string*: `"running"`, si la co-rutina está en ejecución (esto es, invocó a `status`); `"suspended"`, si la co-rutina está suspendida en una llamada a `yield`, o si todavía no ha comenzado a ejecutarse; `"normal"` si la co-rutina está activa pero no ejecutándose (esto es, si ha resumido otra co-rutina); y `"dead"` si la co-rutina ha finalizado su función o si se ha detenido con un error.

coroutine.wrap (f)

Crea una nueva co-rutina con cuerpo `f`. `f` debe ser una función Lua. Retorna una función que resume la co-rutina cada vez que es invocada. Cualquier argumento pasado a la función se comporta como un argumento extra para `resume`. Retorna los mismos valores devueltos por `resume`, excepto el primer booleano. En caso de error, éste se propaga.

coroutine.yield (···)

Suspende la ejecución de la co-rutina invocante. La co-rutina no puede estar ejecutando una función C, un metamétodo o un iterador. Cualquier argumento de `yield` es pasado como resultado extra a `resume`.

5.3 – Módulos

La biblioteca de empaquetado proporciona utilidades básicas para cargar y construir módulos en Lua. Exporta dos de sus funciones directamente al entorno global: `module` y `require`. Las demás se exportan en la tabla `package`.

module (nombre [, ···])

Crea un módulo. Si existe una tabla en `package.loaded[nombre]` ésta es el módulo. En otro caso si existe una tabla global `t` con el nombre dado ésta es el módulo. Sino, finalmente, crea una nueva tabla `t` y le da el nombre global de `nombre` y el valor de `package.loaded[nombre]`. Esta función también inicializa `t._NAME` con el nombre dado, `t._M` con el módulo (`t` mismo), y `t._PACKAGE` con el nombre del paquete (el módulo `nombre` completo menos su último componente; véase más abajo). Para acabar, `module` establece `t` como nuevo entorno de la función actual y el nuevo valor de `package.loaded[nombre]`, de tal manera que `require` retorna `t`.

Si `nombre` es un nombre compuesto (esto es, uno con componentes separados por puntos) `module` crea (o reutiliza, si ya existen) tablas para cada componente. Por ejemplo, si `nombre` es `a.b.c`, entonces `module` almacena la tabla módulo en el campo `c` del campo `b` de la tabla global `a`.

Esta función puede recibir argumentos *opcionales* después del nombre del módulo, donde cada opción es una función que será aplicada sobre el módulo.

require (nombre)

Carga el módulo dado. La función comienza buscando en la tabla `package.loaded` para determinar si `nombre` está ya cargado. Si es así entonces `require` devuelve el valor almacenado en `package.loaded[nombre]`. En otro caso intenta encontrar un *cargador* para el módulo.

Para encontrar un cargador, primero `require` se guía por `package.preload[nombre]`. Cambiando este *array*, se cambia la manera que en `require` busca un módulo. La siguiente explicación está basada en la configuración por defecto de `package.loaders`.

Primero `require` mira en `package.prelodad[modname]`. Si tiene un valor, éste (que debe ser una función) es el cargador. En otro caso `require` busca un cargador en Lua usando el camino de búsqueda guardado en `package.path`. Si también esto falla, busca un cargador en C usando el camino almacenado en `package.cpath`. Si también finalmente esto falla intenta un cargador *todo en uno* (véase `package.loaders`).

Una vez que se encontró el cargador, `require` lo invoca con un único argumento, `nombre`. Si el cargador retorna un valor, `require` lo asigna a `package.loaded[nombre]`. Si el cargador no retorna un valor y no está asignado un valor a `package.loaded[nombre]`, entonces `require` asigna **true** a esta entrada. En cualquier caso, `require` retorna el valor final de `package.loaded[nombre]`.

Si existen errores durante la carga o ejecución del módulo en proceso o si no se pudo encontrar un cargador para el módulo, entonces `require` activa un error.

package.cpath

El camino de búsqueda usado por `require` para buscar un cargador en C.

Lua inicializa este camino `package.cpath` de la misma manera en que inicializa el camino de Lua `package.path`, usando la variable de entorno `LUA_CPATH` (además de otro camino por defecto definido en `luaconf.h`).

package.loaded

Una tabla usada por `require` para controlar qué módulos están ya cargados. Cuando se solicita un módulo `nombre` y `package.loaded[nombre]` no es falso, `require` simplemente retorna el valor almacenado.

package.loaders

Una tabla usada por `require` que controla cómo se cargan los módulos

Cada entrada en esta tabla es una *función buscadora*. Cuando busca un módulo, `require` llama a cada uno de esas buscadoras en orden ascendente, con el nombre del módulo (el argumento pasado a `require`) com único argumento. La función puede retornar otra función (el módulo *cargador* o un *string* que explica que no encontró ese módulo (o **nil** si no tuvo nada que decir). Lua inicializa esta tabla con cuatro funciones.

La primera buscadora simplemente busca un cargador en la tabla `package.preload`.

La segunda buscadora busca un cargador como biblioteca de Lua, usando el camino de búsqueda guardado en `package.path`. Un camino es una secuencia de *plantillas* separadas por puntos y comas (`;`). En cada plantilla, el buscador cambia cada signo de interrogación que aparezca por `nombre_de_fichero`, que es el nombre del módulo con cada punto reemplazado por un "separador de directorios" (como "/" en Unix); entonces intentará abrir el fichero con el nombre resultante. Así, por ejemplo, si el el camino de Lua es el *string*:

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

la búsqueda de un fichero fuente de Lua para el módulo `foo` intentará abrir los ficheros `./foo.lua`., `./foo.lc` y `/usr/local/foo/init.lua`, en ese orden

La tercera buscadora busca un cargador como biblioteca de C, usando el camino dado en la variable `package.cpath`. Por ejemplo, si el camino de C es el *string*:

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

la buscadora, para el módulo `foo` intentará abrir los ficheros `./foo.so`., `./foo.dll` y `/usr/local/foo/init.so`, en ese orden. Una vez que encuentre una biblioteca en C, el buscador usa la utilidad de enlace dinámico para enlazar la aplicación con la biblioteca. Entones intenta encontrar la función C dentro de la biblioteca para ser usada como cargador. El nombre de esta función es el *string* `"luaopen_"` concatenado con una copia del nombre del módulo donde cada punto es reemplazado por un carácter de subrayado (_). Además, si el nombre del módulo tiene un guión, su prefijo hasta el primer guión incluido se elimina. Por ejemplo, si el nombre del módulo es `a.v1-b.c` el nombre de función será `luaopen_b_c`.

La cuarta buscadora intenta un *cargador todo-en-uno*. Busca en el camino de C una biblioteca con el nombre raiz del módulo dado. Por ejemplo, cuando se pide `a.b.c` buscará `a` en una bibliteca C. SI la encuentra busca dentro de ella una función para abrir el submódulo; en nuestro ejemplo, sería `luaopen_a_b_c`. Con esta utilidad, un paquete puede guardar varios submódulos C en una única biblioteca, con cada submódulo manteniendo su función original de apertura.

package.loadlib (nombre_de_biblioteca, nombre_de_func)

Enlaza dinámicamente el programa anfitrión con la biblioteca en C `nombre_de_biblio`. Dentro de esta biblioteca busca una función `nombre_de_func` y la retorna como una función C. (Por tanto, `nombre_de_func` debe seguir el protocolo; véase `lua_CFunction`).

Ésta es una función de bajo nivel. Se salta completamente el sistema de paquetes y de módulos. A diferencia de `require`, no realiza ninguna búsqueda en el camino y no añade automáticamente extensiones. `nombre_de_biblio` debe ser un nombre completo de fichero de la biblioteca en C, incluyendo si es necesario el camino completo y la extensión. `nombre_de_func` debe ser el nombre exacto exportado por la biblioteca en C (el cual puede depender del compilador de C y del cargador del sistema operativo usados).

Esta función no está soportada por el C ANSI. Por tanto sólo está disponible en algunas plataformas (Windows, Linux, Mac OS X, Solaris, BSD, además de otros sistemas Unix que soportan el estándar `dlfcn`).

package.path

El camino de búsqueda usado por `require` para buscar un cargador de Lua.

Al comienzo Lua inicializa esta variable con el valor de la variable de entorno `LUA_PATH` o con un camino por defecto definido en `luaconf.h`, si la variable de entorno no está definida. Si aparece `";;"` en el valor de la variable de entorno se reemplaza por el camino por defecto.

package.preload

Una tabla que almacena cargadores para módulos específicos (véase `require`).

package.seeall (módulo)

Establece una metatabla para `módulo` con su campo `__index` refiriéndose al entorno global, de tal manera que este módulo hereda los valores del entorno global. Se usa como una opción para la función `module`.

5.4 – Manejo de *strings*

Esta biblioteca proporciona funciones genéricas de manejo de *strings*, tales como encontrar y extraer *substrings* y detectar patrones. Cuando se indexa un *string* en Lua el primer carácter está en la posición 1 (no en 0 como en C). Se permite el uso de índices negativos que se interpretan como indexado hacia atrás, desde el final del *string*. Por tanto el último carácter del *string* está en la posición -1, y así sucesivamente.

La biblioteca de *strings* proporciona todas sus funciones en la tabla `string`. También establece una metatabla para *string* donde el campo `__index` apunta a la misma metatabla. Por tanto, se pueden usar las funciones de manejo de *string* en un estilo orientado a objetos. Por ejemplo, `string.byte(s, i)` puede ponerse `s:byte(i)`.

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

Devuelve los códigos numéricos internos de los caracteres `s[i]`, `s[i+1]`, ···, `s[j]`. El valor por defecto de `i` es 1; el valor por defecto de `j` es `i`.

Téngase en cuenta que los códigos numéricos no son necesariamente portables de unas plataformas a otras.

string.char (···)

Recibe cero o más enteros. Devuelve un *string* con igual longitud que el número de argumentos, en el que cada carácter tiene un código numérico interno igual a su correspondiente argumento.

Téngase en cuenta que los códigos numéricos no son necesariamente portables de unas plataformas a otras.

string.dump (function)

Devuelve un *string* que contiene la representación binaria de la función dada, de tal manera que una llamada posterior a `loadstring` con este *string* devuelve una copia de la función. `func` debe ser una función Lua sin *upvalues*.

string.find (s, patrón [, inicio [, básica]])

Busca la primera aparición de `patrón` en el *string* `s`. Si la encuentra, `find` devuelve los índices de `s` donde comienza y acaba la aparación; en caso contrario retorna **nil**. Un tercer argumento numérico opcional `inicio` especifica dónde comenzar la búsqueda; su valor por defecto es 1 y puede ser negativo. Un valor **true** como cuarto argumento opcional `básica` desactiva las utilidades de detección de patrones, realizando entonces la función una operación de "búsqueda básica de *substring*", sin caracteres "mágicos" en el `patrón`. Téngase en cuenta que si se proporciona el argumento `básica` también debe proporcionarse el argumento `inicio`.

Si el patrón tiene *capturas* entonces en una detección con éxito se devuelven los valores capturados, después de los dos índices.

string.format (formato, ···)

Devuelve una versión formateada de sus argumentos (en número variable) siguiendo la descripción dada en su primer argumento (`formato`, que debe ser un *string*). El *string* de formato sigue las mismas reglas que la familia de funciones C estándar `printf`. Las únicas diferencias son que las opciones/modificadores `*`, `l`, `L`, `n`, `p`, y `h` no están soportadas, y que existe una opción extra `q`. Esta última opción da formato a un *string* en una forma adecuada para ser leída de manera segura de nuevo por el intérprete de Lua: el *string* es escrito entre dobles comillas, y todas las dobles comillas, nuevas líneas, ceros y barras inversas del *string* se sustituyen por las secuencias de escape adecuadas en la escritura. Por ejemplo, la llamada

```
string.format('%q', 'un string con "comillas" y \n nueva línea')
```

producirá el *string*:

```
"un string con \"comillas\" y \
 nueva línea"
```

Las opciones `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X` y `x` esperan un número como argumento, mientras que `q` y `s` esperan un *string*.

Esta función no acepta valores de *string* que contengan caracteres cero, excepto como argumentos de la opción `q`.

string.gmatch (s, patrón)

Devuelve una función iteradora que, cada vez que se invoca, retorna las siguientes *capturas* del `patrón` en el *string* `s`.

Si el `patrón` no produce capturas entonces la coincidencia completa se devuelve en cada llamada.

Como ejemplo, el siguiente bucle

```
s = "hola mundo desde Lua"
for w in string.gmatch(s, "%a+") do
  print(w)
end
```

iterará sobre todas las palabras del *string* `s`, imprimiendo una por línea. El siguiente ejemplo devuelve en forma de tabla todos los pares `clave=valor` del *string* dado:

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

Para esta función, un `'^'` al principio de un patrón no funciona como un ancla, sino que previene la iteración.

string.gsub (s, patrón, reemplazamiento [, n])

Devuelve una copia de `s` en la que todas (o las `n` primeras, si se especifica el argumento opcional) las apariciones del `patrón` han sido reemplazadas por el `reemplazamiento` especificado, que puede ser un *string*, una tabla o una función. `gsub` también devuelve, como segundo valor, el número total de coincidencias detectadas.

Si `reemplazamiento` es un *string* entonces su valor se usa en la sustitución. El carácter `%` funciona como un carácter de escape: cualquier secuencia en `reemplazamiento` de la forma *%n*, con *n* entre 1 y 9, significa el valor de la *captura* número *n* en el *substring* (véase más abajo). La secuencia *%0* significa toda la coincidencia. La secuencia *%%* significa un carácter porcentaje `%`.

Si `reemplazamiento` es una tabla entonces en cada captura se devuelve el elemento de la tabla que tiene por clave la primera captura; si el patrón no proporciona ninguna captura entonce toda la coincidencia se utiliza como clave.

Si `reemplazamiento` es una función entonces la misma es invocada cada vez que exista una captura con todos los *substrings capturados* pasados como argumentos en el mismo orden; si no existen capturas entonces toda la coincidencia se pasa como un único argumento.

Si el valor devuelto por la tabla o por la llamada a la función es un *string* o un número, entonces se usa como *string* de reemplazamiento; en caso contrario si es **false** o **nil**, entonces no se realiza ninguna sustitución (esto es, la coincidencia original se mantiene en el *string*).

He aquí algunos ejemplos:

```
x = string.gsub("hola mundo", "(%w+)", "%1 %1")
--> x="hola hola mundo mundo"

x = string.gsub("hola mundo", "%w+", "%0 %0", 1)
--> x="hola hola mundo"

x = string.gsub("hola mundo desde Lua", "(%w+)%s*(%w+)", "%2 %1")
```

```
--> x="mundo hola Lua desde"

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

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

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

string.len (s)

Recibe un *string* y devuelve su longitud. El *string* vacío `""` tiene longitud 0. Los caracteres cero dentro del *string* también se cuentan, por lo que `"a\000bc\000"` tiene longitud 5.

string.lower (s)

Recibe un *string* y devuelve una copia del mismo con todas las letras mayúsculas cambiadas a minúsculas. El resto de los caracteres permanece sin cambios. La definición de letra mayúscula depende del sistema local.

string.match (s, patrón [, inicio])

Busca la primera *aparición* del `patrón` en el *string* `s`. Si encuentra una, entonces `match` retorna la *captura* del patrón; en caso contrario devuelve **nil**. Si el `patrón` no produce ninguna captura entonces se devuelve la coincidencia completa. Un tercer y opcional argumento numérico `inicio` especifica dónde comenzar la búsqueda; su valor por defecto es 1 y puede ser negativo.

string.rep (s, n)

Devuelve un *string* que es la concatenación de `n` copias del *string* `s`.

string.reverse (s)

Devuelve un *string* que es el original `s` invertido.

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

Retorna el *substring* de `s` que comienza en `i` y continúa hasta `j`; `i` y `j` pueden ser negativos. Si `j` está ausente entonces se asume que vale *-1* (equivalente a la longitud del *string*). En particular, la llamada `string.sub(s,1,j)` retorna un prefijo de `s` con longitud `j`, y `string.sub(s, -i)` retorna un sufijo de `s` con longitud `i`.

string.upper (s)

Recibe un *string* y devuelve una copia del mismo con todas las letras minúsculas cambiadas a mayúsculas. El resto de los caracteres permanece sin cambios. La definición de letra minúscula

depende del sistema local.

5.4.1 – Patrones

Clases de caracteres:

Se usan *clases de caracteres* para representar conjuntos de caracteres. Están permitidas las siguientes combinaciones para describir una clase de caracteres:

- ***x*:** (donde *x* no es uno de los *caracteres mágicos* `^$()%.[]*+-?`) representa el propio caracter *x*.
- **`.`:** (un punto) representa cualquier carácter.
- **`%a`:** representa cualquier letra.
- **`%c`:** representa cualquier carácter de control.
- **`%d`:** representa cualquier dígito.
- **`%l`:** representa cualquier letra minúscula.
- **`%p`:** representa cualquier carácter de puntuación.
- **`%s`:** representa cualquier carácter de espacio.
- **`%u`:** representa cualquier letra mayúscula.
- **`%w`:** representa cualquier carácter alfanumérico.
- **`%x`:** representa cualquier dígito hexadecimal.
- **`%z`:** representa el carácter con valor interno 0 (cero).
- **`%`*x*:** (donde *x* es cualquier carácter no alfanumérico) representa el carácter *x*. Ésta es la manera estándar de "escapar" los caracteres mágicos. Cualquier caracter de puntuación (incluso los no mágicos) pueden ser precedidos por un signo de porcentaje '`%`' cuando se quieran representarse a sí mismos en el patrón.

- **`[`*conjunto*`]`:** representa la clase que es la unión de todos los caracteres en el *conjunto*. Un rango de caracteres puede ser especificado separando el carácter del principio y del final mediante un guión '`-`'. Todas las clases del tipo `%`*x* descritas más arriba pueden ser también utilizadas como componentes del *conjunto*. Todos los otros caracteres en el *conjunto* se representan a sí mismos. Por ejemplo, `[%w_]` (o `[_%w]`) representa cualquier carácter alfanumérico o el subrayado, `[0-7]` representa un dígito octal, y `[0-7%l%-]` representa un dígito octal, una letra minúscula o el carácter '`-`'.

  La interacción entre los rangos y las clases no está definida. Por tanto, patrones como `[%a-z]` o `[a-%%]` carecen de significado.

- **`[^`*conjunto*`]`:** representa el complemento de *conjunto*, donde *conjunto* se interpreta como se ha indicado más arriba.

Para todas las clases representadas por letras simples (`%a`, `%c`, etc.) las correspondientes letras mayúsculas representan la clase complementaria. Por ejemplo, `%S` representa cualquier carácter no espacio.

Las definiciones de letra, espacio y otros grupos de caracteres dependen del sistema local. En particular, la clase `[a-z]` puede no ser equivalente a `%l`.

Elementos de un patrón

Cada *elemento de un patrón* puede ser

- una clase de carácter simple, que equivale a cualquier carácter simple de la clase;
- una clase de carácter simple seguida por '*', que equivale a 0 ó más repeticiones de los caracteres de la clase. Estos elementos de repetición siempre equivaldrán a la secuencia de caracteres más larga posible;
- un clase de carácter simple seguida por '+', que equivale a 1 ó más repeticiones de los caracteres de la clase. Estos elementos de repetición siempre equivaldrán a la secuencia de caracteres más larga posible;
- un clase de carácter simple seguida por '-', que también equivale a 0 ó más repeticiones de los caracteres de la clase. Al contrario que '*', Estos elementos de repetición siempre equivaldrán a la secuencia de caracteres más *corta* posible;
- una clase de carácter simple seguida por '?', que equivale a 0 ó 1 apariciones de un carácter de la clase;
- `%`*n*, para *n* entre 1 y 9; este elemento equivale a un *substring* igual a la captura número *n*;
- `%b`*xy*, donde *x* e *y* son dos caracteres diferentes; este elemento equivale a *strings* que comienzan con *x*, finalizan con *y*, estando *equilibrados* *x* e *y*. Esto significa que, iniciando un contador a 0, si se lee el *string* de izquierda a derecha, sumando *+1* por cada *x* que aparezca y *-1* por cada *y*, el *y* final es el primero donde el contador alcanza 0. Por ejemplo, el elemento `%b()` equivale a una expresión con paréntesis emparejados.

Patrón

Un *patrón* es una secuencia de elementos de patrón. Un '^' al comienzo de un patrón ancla la búsqueda del patrón al comienzo del *string* en el que se produce la búsqueda. Un '$' al final de un patrón ancla la búsqueda del patrón al final del *string* en el que se produce la búsqueda. En otras posiciones '^' y '$' no poseen un significado especial y se representan a sí mismos.

Capturas

Un patrón puede contener subpatrones encerrados entre paréntesis que describen *capturas*. Cuando sucede una coincidencia entre un patrón y un *string* dado, los *substrings* que concuerdan con lo indicado entre paréntesis en el patrón, son almacenados (*capturados*) para uso futuro. Las capturas son numeradas de acuerdo a sus paréntesis izquierdos. Por ejemplo, en el patrón `"(a*(.)%w(%s*))"`, la parte del *string* que concuerda con `"a*(.)%w(%s*)"` se guarda en la primera captura (y por tanto tiene número 1); el carácter que concuerda con `"."` se captura con el número 2, y la parte que concuerda con `"%s*"` tiene el número 3.

Como caso especial, la captura vacía `()` retorna la posición actual en el *string* (un número). Por ejemplo, si se aplica el patrón `"()aa()"` al *string* `"flaaap"`, dará dos capturas: 3 y 5.

Un patrón no puede contener caracteres cero. Úsese `%z` en su lugar.

5.5 – Manejo de tablas

Esta biblioteca proporciona funciones genéricas para manejo de tablas. Todas estas funciones están definidas dentro de la tabla `table`.

La mayoría de las funciones en la biblioteca de tablas asume que las mismas representan *arrays* o listas (o sea, están indexadas numéricamente). Para estas funciones, cuando hablamos de la "longitud" de una tabla queremos decir el resultado del operador longitud (#).

table.concat (tabla [, separador [, i [, j]]])

Dado una table donde todos sus elementos son *strings* o números devuelve `tabla[i]..separador..tabla[i+1] ··· separador..tabla[j]`. El valor por defecto de `separador` es el *string* vacío, el valor por defecto de `i` es 1 y el valor por defecto de `j` es la longitud de la tabla. Si `i` es mayor que `j`, la función devuelve un *string* vacío.

table.insert (tabla, [posición,] valor)

Inserta el elemento `valor` en la `posición` dada en la `tabla`, desplazando hacia adelante otros elementos para abrir hueco, si es necesario. El valor por defecto de `posición` es `n+1`, donde `n = #tabla` es la longitud de la tabla (véase §2.5.5), de tal manera que `table.insert(t,x)` inserta `x` al final de la tabla `t`.

table.maxn (tabla)

Devuelve el mayor índice numérico positivo de una tabla dada o cero si la tabla no tiene índices numéricos positivos. (Para hacer su trabajo esta función realiza un barrido lineal de la tabla completa.)

table.remove (tabla [, posición])

Elimina de `tabla` el elemento situado en la `posición` dada, desplazando hacia atrás otros elementos para cerrar espacio, si es necesario. Devuelve el valor del elemento eliminado. El valor por defecto de `posición` es `n`, donde `n` es la longitud de la tabla, por lo que la llamada `table.remove(t)` elimina el último elemento de la tabla `t`.

table.sort (tabla [, comparador])

Ordena los elementos de la tabla en un orden dado *modificando la propia tabla*, desde `table[1]` hasta `table[n]`, donde `n` es la longitud de la tabla. Si se proporciona el argumento `comparador` éste debe ser una función que recibe dos elementos de la tabla y devuelve verdadero cuando el primero es menor que el segundo (por lo que `not comparador(a[i+1],a[i])` será verdadero después de la ordenación). Si no se proporciona una función `comparador` entonces se usa el operador estándar < de Lua.

El algoritmo de ordenación no es estable; esto es, los elementos considerados iguales por la ordenación dada pueden sufrir cambios de orden relativos después de la ordenación.

5.6 – Funciones matemáticas

Esta biblioteca es una interface a la biblioteca matemática estándar de C. Proporciona todas sus funciones dentro de la tabla `math`.

math.abs (x)

Devuelve el valor absoluto de `x`.

math.acos (x)

Devuelve el arco coseno de `x` (en radianes).

math.asin (x)

Devuelve el arco seno de x (en radianes).

math.atan (x)

Devuelve el arco tangente de x (en radianes).

math.atan2 (y, x)

Devuelve el arco tangente de y/x (en radianes), pero usa los signos de ambos argumentos para determinar el cuadrante del resultado. (También maneja correctamente el caso en que x es cero.)

math.ceil (x)

Devuelve el menor entero mayor o igual que x.

math.cos (x)

Devuelve el coseno de x (se asume que está en radianes).

math.cosh (x)

Devuelve el coseno hiperbólico de x.

math.deg (x)

Devuelve en grados sexagesimales el valor de x (dado en radianes).

math.exp (x)

Devuelve el valor de e^x.

math.floor (x)

Devuelve el mayor entero menor o igual que x.

math.fmod (x, y)

Devuelve el resto de la división de x por y.

math.frexp (x)

Devuelve m y e tales que $x = m\ 2^e$, e es un entero y el valor absoluto de m está en el intervalo *[0.5, 1)* (o cero cuando x es cero).

math.huge

El valor HUGE_VAL, un valor más grande o igual que otro valor numérico cualquiera.

math.ldexp (m, e)

Devuelve $m\ 2^e$ (e debe ser un entero).

math.log (x)

Devuelve el logaritmo natural de `x`.

math.log10 (x)

Devuelve el logaritmo decimal (base 10) de `x`.

math.max (x, ···)

Devuelve el mayor valor de entre sus argumentos.

math.min (x, ···)

Devuelve el menor valor de entre sus argumentos.

math.modf (x)

Devuelve dos números, las partes entera y fraccional de `x` .

math.pi

El valor de *pi*.

math.pow (x, y)

Devuelve x^y. (Se puede también usar la expresión `x^y` para calcular este valor.)

math.rad (x)

Devuelve en radianes el valor del ángulo `x` (dado en grados sexagesimales).

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

Esta función es un interface a `rand`, generador simple de números pseudo-aleatorios proporcionado por el ANSI C. (Sin garantías de sus propiedades estadísticas.)

Cuando se invoca sin argumentos devuelve un número pseudoaleatorio real uniforme en el rango `[0,1)`. Cuando se invoca con un número entero `m`, `math.random` devuelve un número pseudoaleatorio entero uniforme en el rango `[1, m]`. Cuando se invoca con dos argumentos `m` y `n` enteros, `math.random` devuelve un número pseudoaleatorio entero uniforme en el rango `[m, n]`.

math.randomseed (x)

Establece `x` como "semilla" para el generador de números pseudoaleatorios: iguales semillas producen iguales secuencias de números.

math.sin (x)

Devuelve el seno de `x` (se asume que está en radianes).

math.sinh (x)

Devuelve el seno hiperbólico de x.

math.sqrt (x)

Devuelve la raiz cuadrada de x. (Se puede usar también la expresión `x^0.5` para calcular este valor.)

math.tan (x)

Devuelve la tangente de x (se asume que está en radianes).

math.tanh (x)

Devuelve la tangente hiperbólica de x.

5.7 – Utilidades de entrada/salida

La biblioteca de entrada/salida (I/O de sus siglas en inglés) proporciona dos estilos diferentes de manejo de ficheros. El primero de ellos usa descriptores de fichero implícitos; esto es, existen dos ficheros por defecto, uno de entrada y otro de salida, y las operaciones se realizan sobre éstos. El segundo estilo usa descriptores de fichero explícitos.

Cuando se usan descriptores implícitos todas las operaciones soportadas están en la tabla `io`. Cuando se usan descriptores explícitos, la operación `io.open` devuelve un descriptor de fichero y todas las operaciones se proporcionan como métodos asociados al descriptor.

La tabla `io` también proporciona tres descriptores de fichero predefinidos con sus significados usuales en C: `io.stdin`, `io.stdout` e `io.stderr`. La biblioteca de entrada/salida nunca cierra esos ficheros.

A no ser que se especifique, todas las funciones de entrada/salida devuelven **nil** en caso de fallo (más un mensaje de error como segundo resultado y un código de error dependiente del sistema como un tercer resultado) y valores diferentes de **nil** si hay éxito.

io.close ([descriptor_de_fichero])

Equivalente a `descriptor_de_fichero:close()`. Sin argumento cierra el fichero de salida por defecto.

io.flush ()

Equivalente a `descriptor_de_fichero:flush` aplicado al fichero de salida por defecto.

io.input ([descriptor_de_fichero | nombre_de_fichero])

Cuando se invoca con un nombre de fichero entonces lo abre (en modo texto), y establece su manejador de fichero como fichero de entrada por defecto. Cuando se llama con un descriptor de fichero simplemente lo establece como manejador para el fichero de entrada por defecto. Cuando se invoca sin argumento devuelve el fichero por defecto actual.

En caso de errores esta función activa `error` en lugar de devolver un código de error.

io.lines ([nombre_de_fichero])

Abre el fichero de nombre dado en modo lectura y devuelve una función iteradora que, cada vez que es invocada, devuelve una nueva línea del fichero. Por tanto, la construcción

```
for linea in io.lines(nombre_de_fichero) do bloque end
```

iterará sobre todas las líneas del fichero. Cuando la función iteradora detecta el final del fichero devuelve **nil** (para acabar el bucle) y cierra automáticamente el fichero.

La llamada a `io.lines()` (sin nombre de fichero) equivale a `io.input():lines()`; esto es, itera sobre todas las líneas del fichero por defecto de entrada. En ese caso no cierra el fichero cuando acaba el bucle.

io.open (nombre_de_fichero [, modo])

Esta función abre un fichero, en el modo especificado en el *string* `mode`. Devuelve un descriptor de fichero o, en caso de error, **nil** además de un mensaje de error.

El *string* que indica `modo` puede ser uno de los siguientes:

- **"r":** modo lectura (por defecto);
- **"w":** modo escritura;
- **"a":** modo adición;
- **"r+":** modo actualización, todos los datos preexistentes se mantienen;
- **"w+":** modo actualización, todos los datos preexistentes se borran;
- **"a+":** modo adición con actualización, todos los datos preexistentes se mantienen, y la escritura se permite sólo al final del fichero.

El *string* que indica el `modo` puede contener también 'b' al final, lo que es necesario en algunos sistemas para abrir el fichero en modo binario. Este *string* es exactamente el que se usa en la función estándar de C `fopen`.

io.output ([descriptor_de_fichero | nombre_de_fichero])

Similar a `io.input`, pero operando sobre el fichero por defecto de salida.

io.popen (prog [, modo])

Comienza a ejecutar el programa `prog` en un proceso separado y retorna un descriptor de fichero que se puede usar para leer datos que escribe `prog` (si `modo` es `"r"`, el valor por defecto) o para escribir datos que lee `prog` (si `modo` es `"w"`).

Esta función depende del sistema operativo y no está disponible en todas las plataformas.

io.read (···)

Equivalente a `io.input():read`.

io.tmpfile ()

Devuelve un descriptor de fichero para un fichero temporal. Éste se abre en modo actualización y se elimina automáticamente cuando acaba el programa.

io.type (objeto)

Verifica si `objeto` es un descriptor válido de fichero. Devuelve el *string* `"file"` si `objeto` es un descriptor de fichero abierto, `"closed file"` si `objeto` es un descriptor de fichero cerrado, o **nil** si `objeto` no es un descriptor de fichero.

io.write (···)

Equivalente a `io.output():write`.

descriptor_de_fichero:close ()

Cierra el descriptor de fichero `descriptor_de_fichero`. Téngase en cuenta que los ficheros son cerrados automáticamente cuando sus descriptores se eliminan en un ciclo de liberación de memoria, pero que esto toma un tiempo impredecible de ejecución.

descriptor_de_fichero:flush ()

Salva cualquier dato escrito en `descriptor_de_fichero`.

descriptor_de_fichero:lines ()

Devuelve una función iteradora que, cada vez que es invocada, devuelve una nueva línea leída del fichero. Por tanto, la construcción

```
for linea in descriptor_de_fichero:lines() do bloque end
```

iterará sobre todas las líneas del fichero. (A diferencia de `io.lines`, esta función no cierra el fichero cuando acaba el bucle.)

file:read (···)

Lee en el fichero dado por el `descriptor_de_fichero`, de acuerdo el formato proporcionado, el cual especifica qué leer. Para cada formato, la función devuelve un *string* (o un número) con los caracteres leídos, o **nil** si no pudo leer los datos con el formato especificado. Cuando se invoca sin formato se usa uno por defecto que lee la próxima línea completa (véase más abajo).

Los formatos disponibles son

- **"*n":** lee un número; éste es el único formato que devuelve un número en lugar de un *string*.
- **"*a":** lee el resto del fichero completo, empezando en la posición actual. Al final del fichero devuelve un *string* vacío.
- **"*l":** lee la próxima línea (saltándose el final de línea), retornando **nil** al final del fichero. Éste es el formato por defecto.
- ***un número*:** lee un *string* con como máximo este número de caracteres, devolviendo **nil** si se llega al final del fichero. Si el número es cero no lee nada y devuelve un *string* vacío, o **nil** si se alcanza el final del fichero.

file:seek ([de_dónde] [, desplazamiento])

Establece (o solicita) la posición actual (del puntero de lectura/escritura) en el `descriptor_de_fichero`, medida desde el principio del fichero, en la posición dada por `desplazamiento` más la base especificada por el *string* `dónde`, como se especifica a continuación:

- **"set":** sitúa la posición base en 0 (comienzo del fichero);
- **"cur":** sitúa la posición base en la actual;
- **"end":** sitúa la posición base al final del fichero.

En caso de éxito la función `seek` retorna la posición final (del puntero de lectura/escritura) en el fichero medida en bytes desde el principio del fichero. Si la llamada falla retorna **nil**, más un *string* describiendo el error.

El valor por defecto de `dónde` es `"cur"`, y para `desplazamiento` es 0. Por tanto, la llamada `descriptor_de_fichero:seek()` devuelve la posición actual, sin cambiarla; la llamada `descriptor_de_fichero:seek("set")` establece la posición al principio del fichero (y devuelve 0); y la llamada `descriptor_de_fichero:seek("end")` establece la posición al final del fichero y devuelve su tamaño.

file:setvbuf (modo [, tamaño])

Establece un modo *buffer* para un fichero de salida. El argumento `modo` puede ser uno de estos tres:

- **"no":** sin *buffer*; el resultado de cualquier operación de salida se produce inmediatamente.
- **"full":** con *buffer* completo; la operación de salida se realiza sólo cuando el *buffer* está lleno o cuando se invoca explícitamente la función `flush` en el descriptor del fichero.
- **"line":** con *buffer* de línea; la salida se demora hasta que se produce una nueva línea en la salida o existe una entrada de algún fichero especial (como una terminal).

Para los dos últimos casos, `tamaño` especifica el tamaño del *buffer*, en bytes. El valor por defecto es un tamaño adecuado.

file:write (···)

Escribe el valor de sus argumentos en el fichero dado por su `descriptor_de_fichero`. Los argumentos pueden ser *strings* o números. Para escribir otros valores úsese `tostring` o `string.format` antes que `write`.

5.8 – Utilidades del sistema operativo

Esta biblioteca está implementada a través de la tabla `os`.

os.clock ()

Devuelve una aproximación al total de segundos de CPU usados por el programa.

os.date ([formato [, tiempo]])

Devuelve un *string* o una tabla conteniendo la fecha y hora, formateada de acuerdo con el *string* dado en `formato`.

Si el argumento `tiempo` está presente entonces ese tiempo concreto es el que se formatea (véase la función `os.time` para una descripción de este valor). En caso contrario, `date` formatea el tiempo actual.

Si `formato` comienza con '`!`' entonces el tiempo se formatea de acuerdo al Tiempo Universal Coordinado. Después de este carácter opcional, si `formato` es `*t` entonces `date` devuelve una tabla con los siguientes campos: `year` (cuatro dígitos), `month` (1--12), `day` (1--31), `hour` (0--23), `min` (0--59), `sec` (0--61), `wday` (día de la semana, el domingo es 1), `yday` (día dentro del año), e `isdst` (booleano, verdadero si es horario de verano).

Si `formato` no es `*t` entonces `date` devuelve el tiempo como un *string*, formateado de acuerdo con las mismas reglas que la función `strftime` de C.

Cuando se invoca sin argumentos `date` devuelve una representación razonable de la fecha y la hora que depende de la máquina y del sistema local (esto es, `os.date()` equivale a `os.date("%c")`).

os.difftime (t2, t1)

Devuelve el número de segundos desde el instante `t1` hasta el `t2`. En POSIX, Windows y algunos otros sistemas este valor es exactamente `t2`-`t1`.

os.execute ([comando])

Esta función equivale a la función `system` de C. Pasa la orden `comando` para que sea ejecutada en el intérprete de comandos del sistema operativo. Devuelve un código de estatus, que es dependiente del sistema. Si el argumento `comando` está ausente devuelve un valor no cero si está disponible un intérprete de comandos y cero si no está disponible.

os.exit ([código])

Invoca la función `exit` de C, con un `código` entero opcional, para terminar el programa anfitrión. El valor por defecto de `código` es el valor correspondiente a éxito.

os.getenv (variable)

Devuelve el valor asignado a la variable de entorno `variable`, o **nil** si la variable no está definida.

os.remove (nombre_de_fichero)

Elimina el fichero o directorio dado. Los directorios deben estar vacíos para poder ser eliminados. Si la función falla retorna **nil**, más un *string* describiendo el error.

os.rename (nombre_viejo, nombre_nuevo)

Renombra un fichero o directorio de `nombre_viejo` a `nombre_nuevo`. Si la función falla retorna **nil**, más un *string* describiendo el error.

os.setlocale (local [, categoría])

Establece valores en el sistema local del programa. `local` es un *string* que especifica un valor local; `categoría` es un *string* opcional que describe qué categoría cambiar: `"all"`, `"collate"`,

`"ctype"`, `"monetary"`, `"numeric"`, or `"time"`; la categoría por defecto es `"all"`. Esta función retorna el nombre del nuevo local o **nil** si la petición no pudo ser aceptada.

Si `local` es el *string* vacío, el local actual se establece como el local nativo (que depende de la implementación). Si `local` es el string `"C"`, el local actual se establece en el local estándar de C.

Cuando se invoca con **nil** como primer argumento, esta función retorna sólo el nombre del local actual en la categoría dada.

os.time ([tabla])

Devuelve el tiempo actual cuando se llama sin argumentos, o un tiempo representando la fecha y hora especificadas en la tabla dada. Ésta debe tener los campos `year`, `month` y `day`, y puede tener los campos `hour`, `min`, `sec` e `isdst` (para una descripción de esos campos, véase la función `os.date`).

El valor retornado es un número, cuyo significado depende del sistema. En POSIX, Windows y algunos otros sistemas este número cuenta el número de segundos desde alguna fecha inicial dada (la "época"). En otros sistemas el significado no está especificado, y el número retornado por `time` puede ser usado sólo como argumento de las funciones `date` y `difftime`.

os.tmpname ()

Devuelve un *string* con un nombre de fichero que puede ser usado como fichero temporal. El fichero debe ser abierto explícitamente antes de su uso y también eliminado explícitamente cuando no se necesite más.

5.9 – La biblioteca de depuración

Esta biblioteca proporciona a los programas en Lua las funcionalidades de la interface de depuración. Se debe usar con cuidado. Las funciones proporcionadas aquí deben ser usadas exclusivamente para depuración y labores similares, tales como el análisis de código. Por favor, resístase la tentación de usar la biblioteca como una herramienta de programación: puede llegar a ser muy lenta. Además, alguna de sus funciones viola alguna de las asunciones acerca del código en Lua (por ejemplo, que las variables locales de una función no pueden ser accedidas desde fuera de la función o que los *userdata* no pueden ser cambiados desde el código Lua) y por tanto pueden comprometer código de otra manera seguro.

Todas las funciones de esta biblioteca se proporcionan en la tabla `debug`.

debug.debug ()

Entra en modo interactivo con el usuario, ejecutando cada *string* que introduce el usuario. Usando comandos simples y otras utilidades de depuración el usuario puede inspeccionar variables globales y locales, cambiar sus valores, evaluar expresiones, etc. Una línea que contiene sólo la palabra `cont` finaliza esta función, por lo que el programa invocante continúa su ejecución.

Téngase presente que los comandos para `degub.debug` no están léxicamente anidados dentro de ninguna función, y no tienen acceso directo a las variables locales.

debug.getfenv (o)

Devuelve el entorno del objeto o.

debug.gethook ([proceso])

Devuelve información sobre el *hook* actual del proceso, en forma de tres valores: la función del *hook* actual, la máscara del *hook* actual y el contador del *hook* actual (como fue establecida por la función `debug.sethook`).

debug.getinfo ([proceso,] func [, qué])

Devuelve una tabla con información acerca de la función `func`. Se puede dar la función directamente, o se puede dar un número en el lugar de `func`, lo que significa la función al nivel de ejecución de la llamada de pila: nivel 0 es el de la función actual (`getinfo` misma); nivel 1 es la función que llamó a `getinfo`; y así sucesivamente. Si `func` es un número mayor que el total de funciones activas entonces `getinfo` retorna **nil**.

La tabla devuelta contiene todos los campos retornados por `lua_getinfo`, con el *string* `qué` describiendo los campos a rellenar. Por defecto, si no se proporciona `qué`, se obtiene toda la información disponible. Si está presente, la opción '`f`' añade un campo denominado `func` con la función misma. Si está presente, la opción '`L`' añade un campo denominado `activelines` con la tabla de líneas válidas.

Por ejemplo, la expresión `debug.getinfo(1,"n").nombre` retorna una tabla con un nombre para la función actual, si pudo encontrar un nombre razonable, y `debug.getinfo(print)` retorna una tabla con toda la información disponible sobre la función `print`.

debug.getlocal ([proceso,] nivel, local)

Esta función devuelve el nombre y el valor de una variable local con índice `local` de la función al `nivel` dado de la pila. (El primer argumento o variable local tiene índice 1, y así sucesivamente, hasta la última variable local activa.) La función retorna **nil** si no existe una variable local con el índice dado, y activa un error cuando se invoca con `nivel` fuera de rango. (Se puede llamar a `debug.getinfo` para verificar si el nivel es válido.)

Los nombres de variable que comienzan con '`(`' (paréntesis de abrir) representan variables internas (de control de bucles, temporales y locales de funciones C).

debug.getmetatable (objeto)

Devuelve la metatabla del `objeto` dado o **nil** si éste no tiene metatabla.

debug.getregistry ()

Retorna la tabla de registro (véase §3.5).

debug.getupvalue (func, up)

Esta función retorna el nombre y el valor del *upvalue* con índice `up` de la función `func`. La función retorna **nil** si no hay un *upvalue* con el índice dado.

debug.setfenv (objeto, tabla)

Establece la `tabla` de entorno de un `objeto` dado.

debug.sethook ([proceso,] func_hook, máscara [, contador])

Establece la función `func_hook` como *hook*. El *string* dado en `máscara` y el número `contador` describen como se invoca al *hook*. La máscara puede tener los siguientes caracteres, con el significado indicado:

- **`"c"`:** El *hook* se invoca cada vez que Lua llama a una función;
- **`"r"`:** El *hook* se invoca cada vez que Lua retorna de una función;
- **`"l"`:** El *hook* se invoca cada vez que Lua entra en una nueva línea de código.

Con un `contador` diferente de cero el *hook* se invoca cada ese número de instrucciones.

Cuando se invoca sin argumentos `debug.sethook` desactiva el *hook*.

Cuando se invoca el *hook* su primer argumento es un *string* describiendo el evento que ha activado su invocación: `"call"`, `"return"` (o `"tail return"`), `"line"` y `"count"`. Para los eventos de línea, el *hook* también devuelve el número de línea como segundo valor. Dentro de un *hook* se puede invocar a `getinfo` con nivel 2 para obtener más información acerca de la función en ejecución (nivel 0 es la función `getinfo` y nivel 1 es la función *hook*), a no ser que el evento sea `"tail return"`. En ese caso Lua sólo simula el retorno, y una llamada a `getinfo` devolverá datos inválidos.

debug.setlocal ([proceso,] nivel, local, valor)

Esta función asigna el `valor` a la variable local con índice `local` de la función al `nivel` dado en la pila, retornando el nombre de la variable local. La función retorna **nil** si no existe una variable local con el índice dado, y activa un error cuando se llama con un `nivel` fuera de rango. (Se puede invocar `getinfo` para verificar si el nivel es válido.)

debug.setmetatable (objeto, tabla)

Establece `tabla` (que puede ser `nil`) como la metatabla del `objeto` dado.

debug.setupvalue (func, up, valor)

Esta función asigna el `valor` al *upvalue* con índice `up` de la función `func`, retornando el nombre del *upvalue*. La función retorna **nil** si no existe el *upvalue* con el índice dado.

debug.traceback ([proceso,] [mensaje] [, nivel])

Devuelve un *string* con el "trazado inverso" de la llamada en la pila. Un `mensaje` opcional se añade al principio del "trazado inverso". Un número de `nivel` opcional indica en qué nivel se comienza el "trazado inverso" (por defecto es 1, la función que está invocando a `traceback`).

6 – Lua como lenguaje independiente

Aunque Lua ha sido diseñado como un lenguaje de extensión, para ser embebido en programas en C, también es frecuentemente usado como lenguaje independiente. Con la distribución estándar se

proporciona un intérprete independiente denominado simplemente `lua`. Éste incluye todas las bibliotecas estándar, incluyendo la de depuración. Se usa así:

```
lua [opciones] [fichero_de_script [argumentos]]
```

Las opciones son:

- **`-e` *sentencia*:** ejecuta el *string sentencia*;
- **`-l` *módulo*:** carga *módulo* con la función `require`;
- **`-i`:** entra en modo interactivo después de ejecutar el *fichero_de_script*;
- **`-v`:** imprime información de la versión;
- **`--`:** deja de procesar opciones en el resto de la línea;
- **`-`:** toma `stdin` como fichero para ejecutar y no procesa más opciones.

Después de gestionar las opciones proporcionadas, `lua` ejecuta el *fichero_de_script* dado, pasándole los *argumentos* dados como *strings*. Cuando se invoca sin argumentos `lua` se comporta como `lua -v -i` cuando la entrada estándar (`stdin`) es una terminal, y como `lua -` en otro caso.

Antes de ejecutar cualquier argumento el intérprete comprueba si existe una variable de entorno `LUA_INIT`. Si su formato es *@nombre_de_fichero* entonces `lua` ejecuta este fichero. En otro caso `lua` ejecuta el propio *string*.

Todas las opciones se procesan en orden, excepto `-i`. Por ejemplo, una invocación como

```
$ lua -e'b=1' -e 'print(b)' script.lua
```

primero establecerá el valor de `b` a 1, luego imprimirá el valor de `b` (que es '1'), y finalmente ejectuará el fichero `script.lua` sin argumentos. (Aquí `$` es el *prompt* del intérprete de comandos del sistema operativo. El de cada sistema concreto puede ser diferente.)

Antes de comenzar a ejecutar *fichero_de_script* `lua` recolecta todos los argumentos de la línea de comandos en una tabla global denominada `arg`. El nombre del *fichero_de_script* se guarda en el índice 0, el primer argumento depués del nombre del programa se guarda en el índice 1, y así sucesivamente. Cualesquiera argumentos antes del nombre del programa (esto es, el nombre del intérprete más las opciones) van a los índices negativos. Por ejemplo, en la invocación

```
$ lua -la b.lua t1 t2
```

el intérprete primero ejecuta el fichero `b.lua`, luego crea la tabla

```
arg = { [-2] = "lua", [-1] = "-la",
        [0] = "b.lua",
        [1] = "t1", [2] = "t2" }
```

y finalmente ejecuta el fichero `b.lua`. Éste se invoca con `arg[1]`, `arg[2]`, ··· como argumentos; también se accede a estos argumentos con la expresión *vararg* '`...`'.

En modo interactivo si se escribe una sentencia incompleta el intérprete espera para que sea completada, indicándolo con otro *prompt* diferente.

Si la variable global `_PROMPT` contiene un *string* entonces su valor se usa como *prompt*. De manera similar, si la variable global `_PROMPT2` contiene un *string* su valor se usa como *prompt* secundario (el que se utiliza durante las sentencias incompletas). Por tanto, ambos *prompts* pueden ser

cambiados directamente en la línea de comandos o en cualquier programa en Lua asignando un valor a `_PROMPT`. Véase el siguiente ejemplo:

```
$ lua -e"_PROMPT='myprompt> '" -i
```

(la pareja externa de comillas es para el intérprete de comandos del sistema operativo; la interna para Lua). Nótese el uso de `-i` para entrar en modo interactivo; en otro caso el programa acabaría silenciosamente justo después de la asignación a `_PROMPT`.

Para permitir el uso de Lua como un intérprete de *scripts* en los sistemas Unix, el intérprete independiente de Lua se salta la primera línea de un *chunk* si ésta comienza con `#`. Por tanto, los programas de Lua pueden convertirse en ejecutables usando `chmod +x` y la forma `#!`, como en

```
#!/usr/local/bin/lua
```

(Por supuesto, la localización del intérprete de Lua puede ser diferente en cada máquina. Si `lua` está en el camino de búsqueda de ejecutables, `PATH`, entonces

```
#!/usr/bin/env lua
```

es una solución más portable.)

7 – Incompatibilidades con la versión anterior

Aquí se listan las incompatibilidades que pueden aparecer cuando se porta un programa de Lua 5.0 a Lua 5.1. Se pueden evitar la mayoría de ellas compilando Lua con las opciones apropiadas (véase el fichero `luaconf.h`). Sin embargo todas esas opciones de compatibilidad serán eliminadas en la siguiente versión de Lua.

7.1 – Cambios en el lenguaje

- El sistema de gestión de funciones con un número de argumentos variable cambió desde el pseudoargumento `arg` con una tabla con argumentos extra a la expresión *vararg*. (Véase la opción de compilación `LUA_COMPAT_VARARG` en `luaconf.h`.)
- Existe un sutil cambio en el ámbito de las variables implícitas de las sentencias **for** y **repeat**.
- La sintaxis `[[...]]` para *string* largo y para comentario largo no permite anidamientos. Se puede usar la nueva sintaxis `[=[...]=]` en esos casos. (Véase la opción de compilación `LUA_COMPAT_LSTR` en `luaconf.h`.)

7.2 – Cambios en las bibliotecas

- La función `string.gfind` ha sido renombrada a `string.gmatch`. (Véase la opción de compilación `LUA_COMPAT_GFIND` en `luaconf.h`.)
- Cuando se invoca `string.gsub` con una función como su tercer argumento, siempre que esta función devuelva **nil** o **false** el *string* de reemplazamiento es la coincidencia completa en lugar de un *string* vacío.
- Se desaconseja el uso de la función `table.setn`. La función `table.getn` corresponde al nuevo operador de longitud (`#`); úsese el operador en lugar de la función. (Véase la opción de

compilación LUA_COMPAT_GETN en luaconf.h.)
- La función loadlib ha sido renombrada a package.loadlib. (Véase la opción de compilación LUA_COMPAT_LOADLIB en luaconf.h.)
- La función math.mod ha sido renombrada a math.fmod. (Véase la opción de compilación LUA_COMPAT_MOD en luaconf.h.)
- Se desaconseja el uso de las funciones table.foreach y table.foreachi. En su lugar se puede usar un bucle con pairs o ipairs.
- Hay cambios sustanciales en la función require debido al nuevo sistema de módulos. No obstante, el nuevo comportamiento es casi totalmente compatible con el viejo, aunque require obtiene el camino de búsqueda de package.path en lugar de LUA_PATH.
- La función collectgarbage tiene otros argumentos. Se desaconseja el uso de la función gcinfo; úsese collectgarbage("count") en su lugar.

7.3 – Cambios en la API

- Las funciones luaopen_* (para abrir bibliotecas) no pueden ser invocadas directamente como funciones C regulares. Deben ser llamadas a través de Lua, como otra función Lua.
- La función lua_open ha sido reemplazada por lua_newstate para permitir al usuario establecer una función de asignación de memoria. Se puede usar luaL_newstate de la biblioteca estándar para crear un estado con la función estándar de asignación de memoria (basada en realloc).
- Las funciones luaL_getn y luaL_setn (de la biblioteca auxiliar) no deben usarse. Úsese lua_objlen en lugar de luaL_getn y nada en lugar de luaL_setn.
- La función luaL_openlib ha sido reemplazada por luaL_register.
- La función luaL_checkudata ahora provoca un error cuando el valor dado no es un *userdata* del tipo esperado. (En Lua 5.0 retornaba NULL.)

8 – La sintaxis completa de Lua

Aquí aparece la sintaxis completa de Lua en la notación BNF extendida. No describe las prioridades de los operadores.

```
chunk ::= {sentencia [';']} [última_sentencia[';']]

bloque ::= chunk

sentencia ::=  varlist '=' explist  |
               llamada_a_func  |
               do bloque end  |
               while exp do bloque end  |
               repeat bloque until exp  |
               if exp then bloque {elseif exp then bloque} [else bloque] end  |
               for nombre '=' exp ',' exp [',' exp] do bloque end  |
               for lista_de_nombres in explist do bloque end  |
               function nombre_de_func cuerpo_de_func  |
               local function nombre cuerpo_de_func  |
               local lista_de_nombres ['=' explist]

última_sentencia ::= return [explist]  |  break
```

```
nombre_de_func ::= nombre {'.' nombre} [':' nombre]

varlist ::= var {',' var}

var ::=  nombre  |  prefixexp '[' exp ']'  |  prefixexp '.' nombre

lista_de_nombres ::= nombre {',' nombre}

explist ::= {exp ','} exp

exp ::=  nil  |  false  |  true  |  número  |  string  |  '...'  |
         func  |  prefixexp  |  constructor_de_tabla  |
         exp operador_binario exp  | operador_unario exp

prefixexp ::= var  |  llamada_a_func  |  '(' exp ')'

llamada_a_func ::=  prefixexp arg_actuales  |  prefixexp ':' nombre args_actuales

args_actuales ::=  '(' [explist] ')'  |  constructor_de_tabla  |  string

func ::= function cuerpo_de_func

cuerpo_de_func ::= '(' [args_formal_list] ')' bloque end

args_formal_list ::= lista_de_nombres [',' '...']  |  '...'

constructor_de_tabla ::= '{' [lista_de_campos] '}'

lista_de_campos ::= campo {separador_de_campo campo} [separador_de_campo]

campo ::= '[' exp ']' '=' exp  |  nombre '=' exp  |  exp

separador_de_campo ::= ','  |  ';'

operador_binario ::= '+'  |  '-'  |  '*'  |  '/'  |  '^'  |  '%'  |
                     '..'  |  '<'  |  '<='  |  '>'  |  '>='  |  '=='  |
                     '~='  |  and  |  or

operador_unario ::= '-'  |  not  |  '#'
```

Notas sobre la traducción

He intentado ser lo más fiel posible al original; probablemente hay errores y erratas; en caso de duda consúltese el original en inglés.

Algunas palabras son de uso tan común en informática que se utilizan en español sin traducir. Otras tienen por traducción una oración completa y por tanto sería bastante poco coherente

introducir esa frase en cada lugar. He preferido, por tanto, dejarlas en el texto (indicadas en *itálica* como es costumbre en español con palabras de otros idiomas), exponiendo aquí la traducción.

- *arrays*: vectores, matrices, etc.; un conjunto de datos identificado por un nombre (que se le da a todo el conjunto); los *arrays* se pueden indexar mediante números enteros o mediante cualquier tipo de dato (en este caso se habla de *arrays asociativos*).
- *buffer*: se puede traducir por "tampón" (aunque el nombre en inglés es muy utilizado); es un espacio auxiliar de memoria para diversas operaciones.
- *chunk*: unidad de ejecución en Lua, que es simplemente una secuencia de sentencias.
- *hook*: en Lua se denomina así a una función en la sombra que se activa al ocurrir ciertos eventos en el código (por ejemplo un retorno de una función); se usa en la depuración del código.
- *prompt*: un indicador de que el intérprete de comandos del sistema operativo está esperando por una entrada del usuario.
- *script*: normalmente se denominan así a los programas que se interpretan (por oposición a los ejecutables compilados).
- *string*: una tira o secuencia de caracteres.
- *upvalue*: una variable no-local, externa a una función (a la que ésta tiene acceso).
- *userdata*: en Lua es un espacio de memoria utilizado por funciones C.

Algunas otras palabras han tenido la traducción siguiente:

- *closure*: instancia (de función).
- *loop*: bucle.
- *statement*: sentencia.
- *thread*: proceso.

last update: thu aug 29 20:31:59 utc 2019