Manual de Referência de Lua 5.1

por Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes

(Traduzido por Sérgio Queiroz de Medeiros com apoio da Fábrica Digital e da FINEP)

conteúdo · índice · english · português · español

1 - Introdução

Lua é uma linguagem de programação de extensão projetada para dar suporte à programação procedimental em geral e que oferece facilidades para a descrição de dados. A linguagem também oferece um bom suporte para programação orientada a objetos, programação funcional e programação orientada a dados. Lua foi planejada para ser utilizada por qualquer aplicação que necessite de uma linguagem de script leve e poderosa. Lua é implementada como uma biblioteca, escrita em C *limpo* (isto é, no subconjunto comum de ANSI C e C++).

Por ser uma linguagem de extensão, Lua não possui a noção de um programa principal: ela somente funciona *embarcada* em um programa cliente anfitrião, chamado de *programa hospedeiro* ou simplesmente de *hospedeiro*. Esse programa hospedeiro pode invocar funções para executar um pedaço de código Lua, pode escrever e ler variáveis Lua e pode registrar funções C para serem chamadas pelo código Lua. Através do uso de funções C, Lua pode ser estendida para lidar de maneira apropriada com uma ampla variedade de domínios, permitindo assim a criação de linguagems de programação personalizadas que compartilham um arcabouço sintático. A distribuição Lua inclui um exemplo de um programa hospedeiro chamado `lua`, o qual usa a biblioteca de Lua para oferecer um interpretador de linha de comando Lua completo.

Lua é um software livre e, como de praxe, é fornecido sem garantias, conforme dito na sua licença. A implementação descrita neste manual está disponível no sítio web oficial de Lua, `www.lua.org`.

Como qualquer outro manual de referência, este documento é árido em algumas partes. Para uma discussão das decisões por trás do projeto de Lua, veja os artigos técnicos disponíveis no sítio web oficial de Lua. Para uma introdução detalhada à programação em Lua, veja o livro de Roberto Ierusalimschy, *Programming in Lua (Segunda Edição)*.

2 - A Linguagem

Esta seção descreve os aspectos léxicos, sintáticos e semânticos de Lua. Em outras palavras, esta seção descreve quais *itens léxicos* são válidos, como eles são combinados, e qual o significado da sua combinação.

As construções da linguagem serão explicadas usando a notação BNF estendida usual, na qual {*a*} significa 0 ou mais *a*'s e [*a*] significa um *a* opcional. Não-terminais são mostrados como non-terminal, palavras-chave são mostradas como **kword** e outros símbolos terminais são mostrados como `=´. A sintaxe completa de Lua pode ser encontrada em §8 no fim deste manual.

2.1 - Convenções Léxicas

Em Lua, *Nomes* (também chamados de *identificadores*) podem ser qualquer cadeia de letras, dígitos, e sublinhados que não começam com um dígito. Esta definição está de acordo com a definição de nomes na maioria das linguagens. (A definição de letras depende de qual é o idioma (*locale*): qualquer caractere considerado alfabético pelo idioma corrente pode ser usado como um identificador.) Identificadores são usados para nomear variáveis e campos de tabelas.

As seguintes *palavras-chave* são reservadas e não podem ser utilizadas como nomes:

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
```

```
repeat    return    then      true      until     while
```

Lua é uma linguagem que diferencia minúsculas de maiúsculas: `and` é uma palavra reservada, mas `And` e `AND` são dois nomes válidos diferentes. Como convenção, nomes que começam com um sublinhado seguido por letras maiúsculas (tais como `_VERSION`) são reservados para variáveis globais internas usadas por Lua.

As seguintes cadeias denotam outros itens léxicos:

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

Cadeias de caracteres literais podem ser delimitadas através do uso de aspas simples ou aspas duplas, e podem conter as seguintes seqüências de escape no estilo de C: '`\a`' (campainha), '`\b`' (backspace), '`\f`' (alimentação de formulário), '`\n`' (quebra de linha), '`\r`' (retorno de carro), '`\t`' (tabulação horizontal), '`\v`' (tabulação vertical), '`\\`' (barra invertida), '`\"`' (citação [aspa dupla]) e '`\'`' (apóstrofo [aspa simples]). Além disso, uma barra invertida seguida por uma quebra de linha real resulta em uma quebra de linha na cadeia de caracteres. Um caractere em uma cadeia de caracteres também pode ser especificado pelo seu valor numérico usando a seqüência de escape `\`*`ddd`*, onde *ddd* é uma seqüência de até três dígitos decimais. (Note que se um caractere numérico representado como um seqüência de escape for seguido por um dígito, a seqüência de escape deve possuir exatamente três dígitos.) Cadeias de caracteres em Lua podem conter qualquer valor de 8 bits, incluindo zeros dentro delas, os quais podem ser especificados como '`\0`'.

Cadeias literais longas também podem ser definidas usando um formato longo delimitado por *colchetes longos*. Definimos uma *abertura de colchete longo de nível n* como um abre colchete seguido por *n* sinais de igual seguido por outro abre colchete. Dessa forma, uma abertura de colchete longo de nível 0 é escrita como `[[`, uma abertura de colchete longo de nível 1 é escrita como `[=[` e assim por diante. Um *fechamento de colchete longo* é definido de maneira similar; por exemplo, um fechamento de colchete longo de nível 4 é escrito como `]====]`. Uma cadeia de caracteres longa começa com uma abertura de colchete longo de qualquer nível e termina no primeiro fechamento de colchete longo do mesmo nível. Literais expressos desta forma podem se estender por várias linhas, não interpretam nenhuma seqüência de escape e ignoram colchetes longos de qualquer outro nível. Estes literais podem conter qualquer coisa, exceto um fechamento de colchete longo de nível igual ao da abertura.

Por conveniência, quando uma abertura de colchete longo é imediatamente seguida por uma quebra de linha, a quebra de linha não é incluída na cadeia de caracteres. Como exemplo, em um sistema usando ASCII (no qual '`a`' é codificado como 97, quebra de linha é codificado como 10 e '`1`' é codificado como 49), as cinco cadeias literais abaixo denotam a mesma cadeia:

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

Uma *constante numérica* pode ser escrita com uma parte decimal opcional e com um expoente decimal opcional. Lua também aceita constantes hexadecimais inteiras, através do uso do prefixo `0x`. Exemplos de constantes numéricas válidas são:

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

Um *comentário* começa com um hífen duplo (`--`) em qualquer lugar, desde que fora de uma cadeia de caracteres. Se o texto imediatamente depois de `--` não é uma abertura de colchete longo, o comentário é um *comentário curto*, o qual se estende até o fim da linha. Caso contrário, ele é um *comentário longo*, que se estende até o fechamento de colchete longo correspondente. Comentários longos são freqüentemente usados para desabilitar código temporariamente.

2.2 - Valores e Tipos

Lua é uma *linguagem dinamicamente tipada*. Isto significa que variáveis não possuem tipos; somente valores possuem tipos. Não existe definição de tipos na linguagem. Todos os valores carregam o seu próprio tipo.

Todos os valores em Lua são *valores de primeira classe*. Isto significa que todos os valores podem ser armazenados em variáveis, passados como argumentos para outras funções e retornados como resultados.

Existem oito tipos básicos em Lua: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread* e *table*. *Nil* é o tipo do valor **nil**, cuja propriedade principal é ser diferente de qualquer outro valor; ele geralmente representa a ausência de um valor útil. *Boolean* é o tipo dos valores **false** e **true**. Tanto **nil** como **false** tornam uma condição falsa; qualquer outro valor torna a condição verdadeira. *Number* representa números reais (ponto flutuante de precisão dupla). (É fácil construir interpretadores Lua que usem outra representação interna para números, tais como precisão simples de ponto flutuante ou inteiros longos; veja o arquivo `luaconf.h`.) O tipo *string* representa cadeias de caracteres. Em Lua, cadeias de

caracteres podem conter qualquer caractere de 8 bits, incluindo zeros ('\0') dentro dela (ver §2.1).

Lua pode chamar (e manipular) funções escritas em Lua e funções escritas em C (ver §2.5.8).

O tipo *userdata* permite que dados C arbitrários possam ser armazenados em variáveis Lua. Este tipo corresponde a um bloco de memória e não tem operações pré-definidas em Lua, exceto atribuição e teste de identidade. Contudo, através do uso de *metatables*, o programador pode definir operações para valores userdata (ver §2.8). Valores userdata não podem ser criados ou modificados em Lua, somente através da API C. Isto garante a integridade dos dados que pertencem ao programa hospedeiro.

O tipo *thread* representa fluxos de execução independentes e é usado para implementar co-rotinas (ver §2.11). Não confunda o tipo thread de Lua com processos leves do sistema operacional. Lua dá suporte a co-rotinas em todos os sistemas, até mesmo naqueles que não dão suporte a processos leves.

O tipo *table* implementa arrays associativos, isto é, arrays que podem ser indexados não apenas por números, mas por qualquer valor (exceto **nil**). Tabelas podem ser *heterogêneas*; isto é, elas podem conter valores de todos os tipos (exceto **nil**). Tabelas são o único mecanismo de estruturação de dados em Lua; elas podem ser usadas para representar arrays comuns, tabelas de símbolos, conjuntos, registros, grafos, árvores, etc. Para representar registros, Lua usa o nome do campo como um índice. A linguagem dá suporte a esta representação oferecendo `a.name` como um açúcar sintático para `a["name"]`. Existem várias maneiras convenientes de se criar tabelas em Lua (ver §2.5.7).

Da mesma forma que os índices, o valor de um campo da tabela pode possuir qualquer tipo (exceto **nil**). Em particular, dado que funções são valores de primeira classe, campos de tabela podem conter funções. Portanto, tabelas podem também possuir *metódos* (ver §2.5.9).

Valores do tipo table, function, thread e userdata (completo) são *objetos*: variáveis não *contêm* realmente estes valores, somente *referências* para eles. Atribuição, passagem de parâmetro, e retorno de funções sempre lidam com referências para tais valores; estas operações não implicam em qualquer espécie de cópia.

A função `type` retorna uma cadeia de caracteres descrevendo o tipo de um dado valor.

2.2.1 - Coerção

Lua provê conversão automática entre valores do tipo string e do tipo number em tempo de execução. Qualquer operação aritmética aplicada a uma cadeia de caracteres tenta converter esta cadeia para um número, seguindo as regras de conversão usuais. De forma análoga, sempre que um número é usado onde uma cadeia de caracteres é esperada, o número é convertido para uma cadeia, em um formato razoável. Para um controle completo sobre como números são convertidos para cadeias, use a função `format` da biblioteca string (ver `string.format`).

2.3 - Variáveis

Variáveis são lugares usados para armazenar valores.

Existem três tipos de variáveis em Lua: variáveis globais, variáveis locais e campos de tabelas.

Um nome simples pode denotar uma variável global ou uma variávei local (ou um parâmetro formal de uma função, que é um caso particular de variável local):

```
var ::= Nome
```

Nome denota identificadores, como definido em §2.1.

Assume-se que toda variável é uma variável global a menos que ela seja explicitamente declarada como uma variável local (ver §2.4.7). Variáveis locais possuem *escopo léxico*: variáveis locais podem ser livremente acessadas por funções definidas dentro do seu escopo (ver §2.6).

Antes da variável receber a sua primeira atribuição, o seu valor é **nil**.

Colchetes são usados para indexar uma tabela:

```
var ::= exprefixo `[´ exp `]´
```

A semântica de acessos a variáveis globais e a campos de tabelas pode ser mudada através do uso de metatabelas. Um acesso a uma variável indexada `t[i]` é equivalente a uma chamada `gettable_event(t,i)`. (Veja §2.8 para uma descrição completa da função `gettable_event`. Esta função não é definida nem pode ser chamada em Lua. Ela é usada aqui somente para fins didáticos.)

A sintaxe `var.Nome` é apenas um açúcar sintático para `var["Nome"]`:

```
var ::= exprefixo `.´ Nome
```

Todas as variáveis globais são mantidas como campos em tabelas Lua comuns, chamadas de *tabelas de ambiente* ou simplesmente de *ambientes* (ver §2.9). Cada função tem sua própria referência para um ambiente, de forma que todas

as variáveis globais dentro de uma função irão se referir para esta tabela de ambiente. Quando uma função é criada, ela herda o ambiente da função que a criou. Para obter a tabela de ambiente de uma função Lua, você deve chamar `getfenv`. Para trocar a tabela de ambiente, você deve chamar `setfenv`. (A única maneira de tratar o ambiente de funções C é através da a biblioteca de depuração; (ver §5.9).)

Um acesso a uma variável global `x` é equivalente a `_env.x`, que por sua vez é equivalente a

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

onde `_env` é o ambiente da função corrente. (Veja §2.8 para uma descrição completa da função `gettable_event`. Esta função não é definida nem pode ser chamada em Lua. De modo análogo, a variável `_env` não é definida em Lua. Elas foram usadas aqui somente para fins didáticos.)

2.4 - Comandos

Lua oferece um conjunto quase convencional de comandos, similar ao conjunto de comandos disponíveis em Pascal ou C. Este conjunto inclui atribuições, estruturas de controle, chamadas de funções e declarações de variáveis.

2.4.1 - Trechos

A unidade de execução de Lua é denominada de *trecho*. Um trecho é simplesmente uma seqüência de comandos, os quais são executados sequencialmente. Cada comando pode opcionalmente ser seguido por um ponto-e-vírgula:

```
trecho ::= {comando [`;´]}
```

Não existem comandos vazios e portanto a construção '**;;**' não é válida.

Lua trata um trecho como o corpo de uma função anônima com um número variável de argumentos (ver §2.5.9). Desta forma, trechos podem definir variáveis locais, receber argumentos e retornar valores.

Um trecho pode ser armazenado em um arquivo ou em uma cadeia de caracteres dentro do programa hospedeiro. Para executar um trecho, Lua primeiro pré-compila o trecho em instruções para uma máquina virtual e depois executa o código compilado com um interpretador para a máquina virtual.

Trechos também podem ser pré-compilados em uma forma binária; veja o programa `luac` para mais detalhes. Programas na forma de código fonte e na forma de um arquivo fonte já compilado são intercambiáveis; Lua automaticamente determina qual é o tipo do arquivo e age em conformidade com ele.

2.4.2 - Blocos

Um bloco é uma lista de comandos; sintaticamente, um bloco é a mesma coisa que um trecho:

```
bloco ::= trecho
```

Um bloco pode ser explicitamente delimitado para produzir um único comando:

```
comando ::= do bloco end
```

Blocos explícitos são úteis para controlar o escopo de declarações de variáveis. Blocos explícitos são também usados às vezes para adicionar um comando **return** ou **break** no meio de outro bloco (ver §2.4.4).

2.4.3 - Atribuição

Lua permite atribuições múltiplas. Em virtude disto, a sintaxe para atribuição define uma lista de variáveis no lado esquerdo e uma lista de expressões no lado direito. Os elementos em ambos os lados são separados por vírgulas:

```
comando ::= listavar `=´ listaexp
listavar ::= var {`,´ var}
listaexp ::= exp {`,´ exp}
```

Expressões são discutidas em §2.5.

Antes da atribuição ser realizada, a lista de valores é *ajustada* para o comprimento da lista de variáveis. Se há mais valores do que o necessário, os valores em excesso são descartados. Se há menos valores do que o necessário, a lista é estendida com tantos **nil**'s quantos sejam necessários. Se a lista de expressões termina com uma chamada de função, então todos os valores retornados por esta chamada entram na lista de valores, antes do ajuste ser realizado (exceto quando a chamada é delimitada por parênteses; veja §2.5).

Um comando de atribuição primeiro avalia todas as suas expressões e somente depois é que a atribuição é realizada. Desta forma, o código

```
i = 3
```

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

atribui 20 a `a[3]`, sem afetar `a[4]` porque o `i` em `a[i]` é avaliado (para 3) antes de receber o valor 4. De modo similar, a linha

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

troca os valores de `x` e `y` e

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

permuta de maneira cíclica os valores de `x`, `y` e `z`.

A semântica de atribuições para variáveis globais e campos de tabelas pode ser mudada através do uso de metatabelas. Uma atribuição para uma variável indexada `t[i] = val` é equivalente a `settable_event(t,i,val)`. (Veja §2.8 para uma descrição completa da função `settable_event`. Esta função não é definida nem pode ser chamada em Lua. Ela foi usada aqui somente para fins didáticos.)

Uma atribuição a uma variável global `x = val` é equivalente à atribuição `_env.x = val`, que por sua vez é equivalente a

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

onde `_env` é o ambiente da função sendo executada. (A variável `_env` não é definida em Lua. Ela foi usada aqui somente para fins didáticos.)

2.4.4 - Estruturas de Controle

As estruturas de controle **if**, **while** e **repeat** possuem o significado usual e a sintaxe familiar:

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

Lua também possui um comando **for**, o qual possui duas variações (ver §2.4.5).

A expressão da condição de uma estrutura de controle pode retornar qualquer valor. Tanto **false** como **nil** são considerados um valor falso. Todos os valores diferentes de **nil** e **false** são considerados como verdadeiros (em particular, o número 0 e a cadeia de caracteres vazia também são considerados valores verdadeiros).

No laço **repeat–until**, o bloco mais interno não termina na palavra-chave **until**, mas somente depois da condição. Desta forma, a condição pode referenciar variáveis locais declaradas dentro do bloco do laço.

O comando **return** é usado para retornar valores de uma função ou de um trecho (que nada mais é do que uma função). Funções e trechos podem retornar mais de um valor, de modo que a sintaxe para o comando **return** é

```
comando ::= return [listaexp]
```

O comando **break** é usado para terminar a execução de um laço **while**, **repeat** ou **for**, pulando para o próximo comando depois do laço:

```
comando ::= break
```

Um **break** termina a execução do laço mais interno.

Os comandos **return** e **break** somente podem ser escritos como o *último* comando de um bloco. Se é realmente necessário ter um **return** ou **break** no meio de um bloco, então um bloco interno explícito pode ser usado, como nas expressões idiomáticas `do return end` e `do break end`, pois agora tanto o **return** como o **break** são os últimos comandos em seus respectivos blocos (internos).

2.4.5 - Comando for

O comando **for** possui duas variações: uma numérica e outra genérica.

O laço **for** numérico repete um bloco de código enquanto uma variável de controle varia de acordo com uma progressão aritmética. Ele possui a seguinte sintaxe:

```
comando ::= for nome `=´ exp `,´ exp [`,´ exp] do bloco end
```

O *bloco* é repetido para *nome* começando com o valor da primeira *exp*, até que ele passe o valor da segunda *exp* através de seguidos passos, sendo que a cada passo o valor da terceira *exp* é somado a *nome*. De forma mais precisa, um comando **for** como

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

é equivalente ao código:

```
do
  local var, limite, passo = tonumber(e1), tonumber(e2), tonumber(e3)
  if not (var and limite and passo) then error() end
  while (passo > 0 and var <= limite) or (passo <= 0 and var >= limite) do
    local v = var
    bloco
    var = var + passo
  end
end
```

Note o seguinte:

- Todas as três expressões de controle são avaliadas um única vez, antes do laço começar. Elas devem obrigatoriamente produzir números.
- *var*, *limite* e *passo* são variáveis invisíveis. Os nomes foram utilizados aqui somente para fins didáticos.
- Se a terceira expressão (o passo) está ausente, então um passo de tamanho 1 é usado.
- É possível usar **break** para sair de um laço **for**.
- A variável de laço `v` é local ao laço; não é possível usar o valor desta variável após o fim do **for** ou depois do **for** ter sido interrompido pelo uso de um **break**. Se você precisa do valor desta variável, atribua-o a outra variável antes de interromper ou sair do laço.

O comando **for** genérico funciona utilizando funções, chamadas de *iteradoras*. A cada iteração, a função iteradora é chamada para produzir um novo valor, parando quando este novo valor é **nil**. O laço **for** genérico possui a seguinte sintaxe:

```
comando ::= for listadenomes in listaexp do bloco end
listadenomes ::= Nome {`,´ Nome}
```

Um comando **for** como

```
for var_1, ···, var_n in explist do block end
```

é equivalente ao 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
    block
  end
end
```

Note o seguinte:

- *explist* é avaliada somente uma vez. Os seus resultados são uma função *iteradora*, um *estado* e um valor inicial para a primeira *variável iteradora*.
- *f*, *s* e *var* são variáveis invisíveis. Os nomes foram utilizados aqui somente para fins didáticos.
- É possível usar **break** para sair de um laço **for**.
- As variáveis de laço *var_i* são locais ao laço; não é possível usar os valores delas após o término do **for**. Se você precisa destes valores, você deve atribuí-los a outras variáveis antes de interromper o laço ou sair do mesmo.

2.4.6 - Chamadas de Função como Comandos

Para permitir possíveis efeitos colaterais, funções podem ser executadas como comandos:

```
comando ::= chamadadefuncao
```

Neste caso, todos os valores retornados pela função são descartados. Chamadas de função são explicadas em §2.5.8.

2.4.7 - Declarações Locais

Variáveis locais podem ser declaradas em qualquer lugar dentro de um bloco. A declaração pode incluir uma atribuição inicial:

```
comando ::= local listadenomes [`=´ listaexp]
```

Caso ocorra uma atribuição inicial, a sua semântica é a mesma de uma atribuição múltipla (ver §2.4.3). Caso contrário, todas as variáveis são inicializadas com **nil**.

Um trecho também é um bloco (ver §2.4.1) e portanto variáveis locais podem ser declaradas em um trecho fora de qualquer bloco explícito. O escopo de uma variável declarada desta forma se estende até o fim do trecho.

As regras de visibilidade para variáveis locais são explicadas em §2.6.

2.5 - Expressões

As expressões básicas em Lua são as seguintes:

```
exp ::= expprefixo
exp ::= nil | false | true
exp ::= Numero
exp ::= Cadeia
exp ::= funcao
exp ::= construtortabela
exp ::= `...´
exp ::= exp opbin exp
exp ::= opunaria exp
expprefixo ::= var | chamadadefuncao | `(´ exp `)´
```

Números e cadeias literais são explicados em §2.1; variáveis são explicadas em §2.3; definições de funções são explicadas em §2.5.9; chamadas de funções são explicadas em §2.5.8; construtores de tabelas são explicados em §2.5.7. Expressões *vararg*, denotadas por três pontos ('`...`'), somente podem ser usadas quando estão imediatamente dentro de uma função que possui um número variável de argumentos; elas são explicadas em §2.5.9.

Operadores binários compreendem operadores aritméticos (ver §2.5.1), operadores relacionais (ver §2.5.2), operadores lógicos (ver §2.5.3) e o operador de concatenação (ver §2.5.4). Operadores unários compreendem o menos unário (ver §2.5.1), o **not** unário (ver §2.5.3) e o *operador de comprimento* unário (ver §2.5.5).

Tanto chamadas de funções como expressões vararg podem resultar em múltiplos valores. Se uma expressão é usada como um comando (o que somente é possível para chamadas de funções (ver §2.4.6)), então a sua lista de retorno é ajustada para zero elementos, descartando portanto todos os valores retornados. Se uma expressão é usada como o último (ou o único) elemento de uma lista de expressões, então nenhum ajuste é feito (a menos que a chamada seja delimitada por parênteses). Em todos os demais contextos, Lua ajusta a lista de resultados para um elemento, descartando todos os valores exceto o primeiro.

Aqui estão alguns exemplos:

```
f()                -- ajusta para 0 resultados
g(f(), x)          -- f() é ajustado para 1 resultado
g(x, f())          -- g recebe x mais todos os resultados de f()
a,b,c = f(), x     -- f() é ajustado para 1 resultado (c recebe nil)
a,b = ...          -- a recebe o primeiro parâmetro da lista vararg,
                   -- b recebe o segundo (tanto a como b podem receber nil caso não
                   -- exista um parâmetro correspondente na lista)

a,b,c = x, f()     -- f() é ajustado para 2 resultados
a,b,c = f()        -- f() é ajustado para 3 resultados
return f()         -- retorna todos os resultados de f()
return ...         -- retorna todos os resultados recebidos da lista vararg
return x,y,f()     -- retorna x, y e todos os resultados de f()
{f()}              -- cria uma lista com todos os resultados de f()
{...}              -- cria uma lista com todos os parâmetros da lista vararg
{f(), nil}         -- f() é ajustado para 1 resultado
```

Qualquer expressão delimitada por parênteses sempre resulta em um único valor. Dessa forma, `(f(x,y,z))` é sempre um único valor, mesmo que `f` retorne múltiplos valores. (O valor de `(f(x,y,z))` é o primeiro valor retornado por `f`, ou **nil** se `f` não retorna nenhum valor.)

2.5.1 - Operadores Aritméticos

Lua provê os operadores aritméticos usuais: os operadores binários `+` (adição), `-` (subtração), `*` (multiplicação), `/` (divisão), `%` (módulo) e `^` (exponenciação); e o operador unário `-` (negação). Se os operandos são números ou cadeias de caracteres que podem ser convertidas para números (ver §2.2.1), então todas as operações possuem o seu significado usual. A exponenciação funciona para qualquer expoente. Por exemplo, `x^(-0.5)` calcula o inverso da raiz quadrada de `x`. Módulo é definido como

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

Ou seja, é o resto de uma divisão arredondada em direção a menos infinito.

2.5.2 - Operadores Relacionais

Os operadores relacionais em Lua são

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

Estes operadores sempre possuem como resultado **false** ou **true**.

A igualdade (==) primeiro compara o tipo de seus operandos. Se os tipos são diferentes, então o resultado é **false**. Caso contrário, os valores dos operandos são comparados. Números e cadeias de caracteres são comparados de maneira usual. Objetos (valores do tipo table, userdata, thread e function) são comparados por *referência*: dois objetos são considerados iguais somente se eles são o *mesmo* objeto. Toda vez que um novo objeto é criado (um valor com tipo table, userdata, thread ou function) este novo objeto é diferente de qualquer outro objeto que existia anteriormente.

É possível mudar a maneira como Lua compara os tipos table e userdata através do uso do metamétodo "eq" (ver §2.8).

As regras de conversão em §2.2.1 *não* se aplicam a comparações de igualdade. Portanto, `"0"==0` é avaliado como **false** e `t[0]` e `t["0"]` denotam posições diferentes em uma tabela.

O operador ~= é exatamente a negação da igualdade (==).

Os operadores de ordem trabalham da seguinte forma. Se ambos os argumentos são números, então eles são comparados como tais. Caso contrário, se ambos os argumentos são cadeias de caracteres, então seus valores são comparados de acordo com a escolha de idioma atual. Caso contrário, Lua tenta chamar o metamétodo "lt" ou o metamétodo "le" (ver §2.8). Uma comparação `a > b` é traduzida para `b < a`, ao passo que `a >= b` é traduzida para `b <= a`.

2.5.3 - Operadores Lógicos

Os operadores lógicos em Lua são **and**, **or** e **not**. Assim como as estruturas de controle (ver §2.4.4), todos os operadores lógicos consideram **false** e **nil** como falso e qualquer coisa diferente como verdadeiro.

O operador de negação **not** sempre retorna **false** ou **true**. O operador de conjunção **and** retorna seu primeiro argumento se este valor é **false** ou **nil**; caso contrário, **and** retorna seu segundo argumento. O operador de disjunção **or** retorna seu primeiro argumento se o valor deste é diferente de **nil** e de **false**; caso contrário, **or** retorna o seu segundo argumento. Tanto **and** como **or** usam avaliação de curto-circuito; isto é, o segundo operando é avaliado somente quando é necessário. Aqui estão alguns exemplos:

```
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
```

(Neste manual, `-->` indica o resultado da expressão precedente.)

2.5.4 - Concatenação

O operador de concatenação de cadeias de caracteres em Lua é denotado por dois pontos ('`..`'). Se ambos os operandos são cadeias de caracteres ou números, então eles são convertidos para cadeias de caracteres de acordo com as regras mencionadas em §2.2.1. Caso contrário, o metamétodo "concat" é chamado (ver §2.8).

2.5.5 - O Operador de Comprimento

O operador de comprimento é denotado pelo operador unário `#`. O comprimento de uma cadeia de caracteres é o seu número de bytes (isto é, o significado usual de comprimento de uma cadeia quando cada caractere ocupa um byte).

O comprimento de uma tabela `t` é definido como qualquer índice inteiro `n` tal que `t[n]` não é **nil** e `t[n+1]` é **nil**; além disso, se `t[1]` é **nil**, `n` pode ser zero. Para um array comum, com todos os valores diferentes de **nil** indo de 1 até um dado `n`, o seu comprimento é exatamente aquele `n`, o índice do seu último valor. Se o array possui "buracos" (isto é, valores **nil** entre dois outros valores diferentes de **nil**), então `#t` pode ser qualquer um dos índices que imediatamente precedem um valor **nil** (isto é, ele pode considerar qualquer valor **nil** como o fim do array).

2.5.6 - Precedência

A precedência de operadores em Lua segue a tabela abaixo, da menor prioridade para a maior:

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
```

```
*     /     %
not   #     - (unary)
^
```

Como é de costume, você pode usar parênteses para mudar as precedências de uma expressão. Os operadores de concatenação ('`..`') e de exponenciação ('^') são associativos à direita. Todos os demais operadores binários são associativos à esquerda.

2.5.7 - Construtores de Tabelas

Construtores de tabelas são expressões que criam tabelas. Toda vez que um construtor é avaliado, uma nova tabela é criada. Um construtor pode ser usado para criar uma tabelas vazia ou para criar uma tabela e inicializar alguns dos seus campos. A sintaxe geral de construtores é

```
construtortabela ::= `{´ [listadecampos] `}´
listadecampos ::= campo {separadordecampos campo} [separadordecampos]
campo ::= `[´ exp `]´ `=´ exp | Nome `=´ exp | exp
separadordecampos ::= `,´ | `;´
```

Cada campo da forma `[exp1] = exp2` adiciona à nova tabela uma entrada cuja chave é `exp1` e cujo valor é `exp2`. Um campo da forma `Nome = exp` é equivalente a `["Nome"] = exp`. Finalmente, campos da forma `exp` são equivalentes a `[i] = exp`, onde `i` representa números inteiros consecutivos, iniciando com 1. Campos nos outros formatos não afetam esta contagem. Por exemplo,

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

é equivalente a

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

Se o último campo na lista possui a forma `exp` e a expressão é uma chamada de função ou uma expressão com um número variável de argumentos, então todos os valores retornados pela expressão entram na lista consecutivamente (ver §2.5.8). Para evitar isto, coloque parênteses ao redor da chamada de função ou da expressão com número variável de argumentos (ver §2.5).

A lista de campos pode ter um separador a mais no fim, como uma conveniência para código gerado automaticamente.

2.5.8 - Chamadas de Função

Uma chamada de função em Lua tem a seguinte sintaxe:

```
chamadadefuncao ::= expprefixo args
```

Em uma chamada de função, primeiro expprefixo e args são avaliados. Se o valor de expprefixo possui tipo *function*, então esta função é chamada com os argumentos fornecidos. Caso contrário, o metamétodo "call" de expprefixo é chamado, tendo como primeiro parâmetro o valor de expprefixo, seguido pelos argumentos originais da chamada (ver §2.8).

A forma

```
chamadadefuncao ::= expprefixo `:´ Nome args
```

pode ser usada para chamar "métodos". Uma chamada `v:nome(`*`args`*`)` é um açúcar sintático para `v.nome(v,`*`args`*`)`, com a diferença de que `v` é avaliado somente uma vez.

Argumentos possuem a seguinte sintaxe:

```
args ::= `(´ [listaexp] `)´
args ::= construtordetabela
args ::= Cadeia
```

Todas as expressões fornecidas como argumento são avaliadas antes da chamada. Uma chamada da forma `f{`*`campos`*`}` é uma açúcar sintático para `f({`*`campos`*`})`; ou seja, a lista de argumentos consiste somente em uma tabela nova. Uma chamada da forma `f'`*`cadeia`*`'` (ou `f"`*`cadeia`*`"` ou `f[[`*`cadeia`*`]]`) é um açúcar sintático para

`f('cadeia')`; ou seja, a lista de argumentos consiste somente em uma cadeia de caracteres literal.

Uma exceção em relação à sintaxe de formato livre de Lua é que não é possível colocar uma quebra de linha antes do '`(`' em uma chamada de função. Esta restrição evita algumas ambigüidades na linguagem. Se você escrevesse

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

Lua poderia ver isto como um comando único, `a = f(g).x(a)`. Portanto, se você deseja dois comandos, você deve obrigatoriamente colocar um ponto-e-vírgula entre eles. Se você realmente quer chamar `f`, você deve remover a quebra de linha antes de `(g)`.

Uma chamada da forma `return` *chamadadefuncao* é denominada de *chamada final*. Lua implementa *chamadas finais próprias* (ou *recursões finais próprias*): em uma chamada final, a função chamada reusa a entrada na pilha da função que a chamou. Portanto, não há limite no número de chamadas finais aninhadas que um programa pode executar. Contudo, uma chamada final apaga qualquer informação de depuração sobre a função chamadora. Note que uma chamada final somente acontece com uma sintaxe particular, onde o **return** possui uma única chamada de função como argumento; esta sintaxe faz com que a chamada de função retorne exatamente os valores de retorno da função chamada. Dessa forma, nenhum dos exemplos a seguir são chamadas finais:

```
return (f(x))        -- o número de resultados é ajustado para 1
return 2 * f(x)
return x, f(x)       -- resultados adicionais
f(x); return         -- resultados descartados
return x or f(x)     -- o número de resultados é ajustado para 1
```

2.5.9 - Definições de Funções

A sintaxe para a definição de uma função é

```
funcao ::= function corpodafuncao
funcao ::= `(´ [listapar] `)´ bloco end
```

O seguinte açúcar sintático simplifica definições de funções:

```
comando ::= function nomedafuncao corpodafuncao
comando ::= local function Nome corpodafuncao
nomedafuncao ::= Nome {`.´ Nome} [`:´ Nome]
```

O comando

```
function f () corpo end
```

é traduzido para

```
f = function () corpo end
```

O comando

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

é traduzido para

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

O comando

```
local function f () corpo end
```

é traduzido para

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

e *não* para

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

(Isto somente faz diferença quando o corpo da função contém uma referência para `f`.)

Uma definição de função é uma expressão executável, cujo valor tem tipo *function*. Quando Lua pré-compila um trecho, todos os corpos das funções do trecho são pré-compilados também. Então, sempre que Lua executa a definição de uma função, a função é *instanciada* (ou *fechada*). Esta instância da função (ou *fecho*) é o valor final da expressão. Instâncias diferentes da mesma função podem se referir a diferentes variáveis locais externas e podem ter diferentes tabelas de ambiente.

Parâmetros comportam-se como variáveis locais que são inicializadas com os valores dos argumentos:

```
listapar ::= listadenomes [`,´ `...´] | `...´
```

Quando uma função é chamada, a lista de argumentos é ajustada para o comprimento da lista de parâmetros, a não ser que a função seja de aridade variável ou *vararg*, o que é indicado por três pontos ('`...`') no final da sua lista de parâmetros. Uma função vararg não ajusta sua lista de argumentos; ao invés disso, ela coleta todos os argumentos extras e os fornece para a função através de uma *expressão vararg*, a qual também é representada como três pontos. O valor desta expressão é uma lista de todos os argumentos extras correntes, similar a uma função com múltiplos valores de retorno. Se uma expressão vararg é usada dentro de outra expressão ou no meio de uma lista de expressões, então a sua lista de valores de retorno é ajustada para um elemento. Se a expressão é usada como o último elemento de uma lista de expressões, então nenhum ajuste é feito (a menos que a última expressão seja delimitada por parênteses).

Como um exemplo, considere as seguintes definições:

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

Neste caso, nós temos o seguinte mapeamento de argumentos para parâmetros e para as expressões vararg:

```
CHAMADA           PARÂMETROS

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
```

Resultados são retornados usando o comando **return** (ver §2.4.4). Se o controle alcança o fim de uma função sem encontrar um comando **return**, então a função retorna sem nenhum resultado.

A sintaxe de *dois pontos* é usada para definir *métodos*, isto é, funções que possuem um parâmetro extra implícito `self`. Desta forma, o comando

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

é uma açúcar sintático para

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

2.6 - Regras de Visibilidade

Lua é uma linguagem com escopo léxico. O escopo das variáveis começa no primeiro comando *depois* da sua declaração e vai até o fim do bloco mais interno que inclui a declaração. Considere o seguinte exemplo:

```
x = 10                -- variável global
do                    -- bloco novo
  local x = x         -- novo 'x', com valor 10
  print(x)            --> 10
  x = x+1
  do                  -- outro bloco
    local x = x+1     -- outro 'x'
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10  (o x global)
```

Note que, em uma declaração como `local x = x`, o novo `x` sendo declarado não está no escopo ainda e portanto o segundo `x` se refere a uma variável externa.

Por causa das regras de escopo léxico, variáveis locais podem ser livremente acessadas por funções definidas dentro do seu escopo. Uma variável local usada por uma função mais interna é chamada de *upvalue* ou *variável local externa*, dentro da função mais interna.

Note que cada execução de um comando **local** define novas variáveis locais. Considere o exemplo a seguir:

```
a = {}
local x = 20
```

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

O laço cria dez fechos (isto é, dez instâncias da função anônima). Cada um destes fechos usa uma variável `y` diferente, enquanto todos eles compartilham a mesma variável `x`.

2.7 - Tratamento de Erros

Dado que Lua é uma linguagem embarcada de extensão, todas as ações de Lua começam a partir de código C no programa hospedeiro que chama uma função da biblioteca de Lua (ver `lua_pcall`). Sempre que um erro ocorre durante a compilação ou execução, o controle retorna para C, que pode tomar as medidas apropriadas (tais como imprimir uma mensagem de erro).

O código Lua pode explicitamente gerar um erro através de uma chamada à função `error`. Se você precisa capturar erros em Lua, você pode usar a função `pcall`.

2.8 - Metatabelas

Todo valor em Lua pode ter uma *metatabela*. Esta *metatabela* é uma tabela Lua comum que define o comportamento do valor original com relação a certas operações especiais. É possível mudar vários aspectos do comportamento de operações sobre um valor especificando campos específicos na metatabela do valor. Por exemplo, quando um valor não numérico é o operando de uma adição, Lua verifica se existe uma função associada com o campo `"__add"` na metatabela do valor. Se a função existe, Lua chama esta função para realizar a adição.

Chamamos as chaves em uma metatabela de *eventos* e os valores de *metamétodos*. No exemplo anterior, o evento é `"add"` e o metamétodo é a função que realiza a adição.

É possível obter a metatabela de qualquer valor usando a função `getmetatable`.

Você pode mudar a metatabela de tabelas através da função `setmetatable`. Você não pode mudar a metatabela de outros tipos de Lua (a menos que você use a biblioteca de depuração); você deve obrigatoriamente usar a API C para fazer isto.

Tabelas e objetos do tipo userdata completos possuem metatabelas individuais (embora múltiplas tabelas e objetos userdata possam compartilhar suas metatabelas). Valores de todos os outros tipos compartilham um única metatabela por tipo; ou seja, há somente uma metatabela para todos os números, uma para todas as cadeias de caracteres, etc.

Uma metatabela controla como um objeto se comporta em operações aritméticas, comparações com relação à ordem, concatenação, operação de comprimento e indexação. Uma metatabela também pode definir uma função a ser chamada quando um objeto userdata é coletado pelo coletor de lixo. Para cada uma destas operações Lua associa uma chave específica chamada um *evento*. Quando Lua realiza uma destas operações sobre um valor, Lua verifica se este valor possui uma metatabela com o evento correspondente. Se este é o caso, o valor associado àquela chave (o metamétodo) controla como Lua irá realizar a operação.

Metatabelas controlam as operações listadas a seguir. Cada operação é identificada por seu nome correspondente. A chave para cada operação é uma cadeia de caracteres começando com o nome da operação sendo precedido por dois sublinhados, '__'; por exemplo, a chave para a operação "add" é a cadeia `"__add"`. A semântica destas operações é melhor explicada por meio de uma função Lua que descreve como o interpretador executa a operação.

O código mostrado aqui é meramente ilustrativo; o comportamento real está codificado no interpretador e é muito mais eficiente do que esta simulação. Todas as funções usadas nestes descrições (`rawget`, `tonumber`, etc.) são descritas em §5.1. Em particular, para recuperar o metamétodo de um dado objeto, usamos a expressão

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

Isto deve ser lido como

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

Isto é, o acesso a um metamétodo não invoca outros metamétodos e o acesso a objetos que não possuem metatabelas não falha (ele simplesmente resulta em **nil**).

- **"add":** a operação +.

  A função `getbinhandler` abaixo define como Lua escolhe um tratador para uma operação binária. Primeiro, Lua tenta o primeiro operando. Se o seu tipo não definie um tratador para a operação, então Lua tenta o segundo operando.

  ```
  function getbinhandler (op1, op2, event)
  ```

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

Usando esta função, o comportamento da expressão `op1 + op2` é

```
function add_event (op1, op2)
  local o1, o2 = tonumber(op1), tonumber(op2)
  if o1 and o2 then  -- os dois operandos são numéricos?
    return o1 + o2   -- '+' aqui é a 'add' primitiva
  else  -- pelo menos um dos operandos nao é numérico
    local h = getbinhandler(op1, op2, "__add")
    if h then
      -- chama o tratador com ambos os operandos
      return (h(op1, op2))
    else  -- nenhum tratador disponível: comportamento padrão
      error(···)
    end
  end
end
```

- **"sub":** a operação `-`. Comportamento similar ao da operação "add".
- **"mul":** a operação `*`. Comportamento similar ao da operação "add".
- **"div":** a operação `/`. Comportamento similar ao da operação "add".
- **"mod":** a operação `%`. Comportamento similar ao da operação "add", tendo a operação `o1 - floor(o1/o2)*o2` como operação primitiva.
- **"pow":** a operação `^` (exponenciação). Comportamento similar ao da operação "add", com a função `pow` (da biblioteca matemática de C) como operação primitiva.
- **"unm":** a operação `-` unária.

```
function unm_event (op)
  local o = tonumber(op)
  if o then  -- operando é numérico?
    return -o  -- '-' aqui é a 'unm' primitiva
  else  -- o operando não é numérico.
    -- Tenta obter um tratador do operando
    local h = metatable(op).__unm
    if h then
      -- chama o tratador com o operando
      return (h(op))
    else  -- nenhum tratador disponível: comportamento padrão
      error(···)
    end
  end
end
```

- **"concat":** a operação `..` (concatenação).

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

- **"len":** a operação #.

```
function len_event (op)
  if type(op) == "string" then
    return strlen(op)         -- comprimento de string primitiva
  elseif type(op) == "table" then
    return #op                -- comprimento de tabela primitiva
  else
    local h = metatable(op).__len
    if h then
      -- chama o tratador com o operando
      return (h(op))
    else  -- nenhum tratador disponível: comportamento padrão
```

```
      error(···)
    end
  end
end
```

Veja §2.5.5 para uma descrição do comprimento de um tabela.

- **"eq":** a operação ==. A função `getcomphandler` define como Lua escolhe um metamétodo para operadores de comparação. Um metamétodo somente é selecionado quando os dois objetos que estão sendo comparados possuem o mesmo tipo e o mesmo metamétodo para a operação selecionada.

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

O evento "eq" é definido da seguinte forma:

```
function eq_event (op1, op2)
  if type(op1) ~= type(op2) then  -- tipos diferentes?
    return false   -- objetos diferentes
  end
  if op1 == op2 then   -- igual primitivo?
    return true   -- objetos são iguais
  end
  -- tenta metamétodo
  local h = getcomphandler(op1, op2, "__eq")
  if h then
    return (h(op1, op2))
  else
    return false
  end
end
```

`a ~= b` é equivalente a `not (a == b)`.

- **"lt":** a operação <.

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

`a > b` é equivalente a `b < a`.

- **"le":** a operação <=.

```
function le_event (op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 <= op2   -- comparação numérica
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 <= op2   -- comparação 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` é equivalente a `b <= a`. Note que, na ausência de um metamétodo "le", Lua tenta o "lt", assumindo que `a <= b` é equivalente a `not (b < a)`.

- **"index":** A indexação de leitura `table[key]`.

```
function gettable_event (table, key)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    if v ~= nil then return v end
    h = metatable(table).__index
    if h == nil then return nil end
  else
    h = metatable(table).__index
    if h == nil then
      error(···)
    end
  end
  if type(h) == "function" then
    return (h(table, key))    -- chama o tratador
  else return h[key]          -- ou repete a operação sobre ele
  end
end
```

- **"newindex":** A indexação de atribuição `table[key] = value`.

```
function settable_event (table, key, value)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    if v ~= nil then rawset(table, key, value); return end
    h = metatable(table).__newindex
    if h == nil then rawset(table, key, value); return end
  else
    h = metatable(table).__newindex
    if h == nil then
      error(···)
    end
  end
  if type(h) == "function" then
    h(table, key, value)       -- chama o tratador
  else h[key] = value          -- ou repete a operação sobre
  end
end
```

- **"call":** chamada quando Lua chama um valor.

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

2.9 - Ambientes

Além de metatabelas, objetos do tipo thread, function e userdata possuem outra tabela associada com eles, chamada de seu *ambiente*. Assim como metatabelas, ambientes são tabelas normais e vários objetos podem compartilhar o mesmo ambiente.

Objetos do tipo thread são criados compartilhando o ambiente da thread que os criou. Objetos do tipo userdata e funções C são criados compartilhando o ambiente da função C que os criou. Funções Lua não aninhadas (criadas por

`loadfile`, `loadstring` ou `load`) são criadas compartilhando o ambiente da thread que as criou. Funções Lua aninhadas são criadas compartilhando o ambiente da função Lua que as criou.

Ambientes associados com objetos do tipo userdata não possuem significado para Lua. É apenas uma conveniência para programadores associarem uma tabela a um objeto userdata.

Ambientes associados com fluxos de execução (threads) são chamados de *ambientes globais*. Eles são usados como o ambiente padrão pelos fluxos de execução e funções não aninhadas criadas pelo fluxo de execução e podem ser diretamente acessados pelo código C (ver §3.3).

O ambiente associado com uma função C pode ser diretamente acessado pelo código C (ver §3.3). Ele é usado como o ambiente padrão para outras funções C e objetos userdata criados pela função.

Ambientes associados com funções Lua são usados para resolver todos os acessos a variáveis globais dentro da função (ver §2.3). Eles são usados como o ambiente padrão para outras funções Lua criadas pela função.

É possível mudar o ambiente de uma função Lua ou do fluxo de execução que está sendo executado atualmente chamando `setfenv`. É possível obter o ambiente de uma função Lua ou do fluxo de execução sendo executado atualmente chamando `getfenv`. Para tratar o ambiente de outros objetos (userdata, funções C, outros fluxos de execução) você deve obrigatoriamente usar a API C.

2.10 - Coleta de Lixo

Lua realiza gerenciamento automático da memória. Isto significa que você não precisa se preocupar com a alocação de memória para novos objetos nem com a liberação de memória quando os objetos não são mais necessários. Lua gerencia a memória automaticamente executando um *coletor de lixo* de tempos em tempos para coletar todos os *objetos mortos* (ou seja, objetos que não são mais acessíveis a partir de Lua). Toda memória usada por Lua está sujeita ao gerenciamento automático de memória: tabelas, userdata, funções, fluxos de execução, cadeias de caracteres, etc.

Lua implementa um coletor de lixo marca-e-limpa (*mark-and-sweep*) incremental. O coletor usa dois números para controlar o seu ciclo de coleta de lixo: a *pausa do coletor de lixo* e o *multiplicador de passo do coletor de lixo*. O valor de ambos é expresso de forma percentual (ou seja, um valor de 100 representa um valor interno de 1).

A pausa do coletor de lixo controla quanto tempo o coletor espera antes de iniciar um novo ciclo. Valores maiores fazem o coletor ser menos agressivo. Valores menores do que 100 significam que o coletor não irá esperar para iniciar um novo ciclo. Um valor de 200 significa que o coletor irá esperar até que a memória total em uso dobre antes de iniciar um novo ciclo.

O multiplicador de passo controla a velocidade relativa do coletor em relação à alocação de memória. Valores maiores fazem o coletor ser mais agressivo mas também aumentam o tamanho de cada passo incremental. Valores menores do que 100 fazem com que o coletor seja muito lento e pode ocorrer que o coletor nunca termine um ciclo. O valor padrão, 200, significa que o coletor é executado a uma velocidade que é "duas vezes" a velocidade de alocação de memória.

É possível mudar estes números através de chamadas às funções `lua_gc` em C ou `collectgarbage` em Lua. Com estas funções você também pode controlar o coletor diretamente (e.g., pará-lo e reiniciá-lo).

2.10.1 - Metamétodos de Coleta de Lixo

Usando a API C, você pode configurar os metamétodos do coletor de lixo para objetos userdata (ver §2.8). Estes metamétodos também são chamados de *finalizadores*. Finalizadores permitem que você coordene a coleta de lixo de Lua com o gerenciamento de recursos externos (tais como o fechamento de arquivos, conexões de rede ou de bancos de dados ou a liberação de sua própria memória).

Objetos userdata com um campo `__gc` em suas metatabelas não são recolhidos imediatamente pelo coletor de lixo. Ao invés disso, Lua os coloca naquela lista. Depois que a coleta é realizada, Lua faz o equivalente da seguinte função para cada objeto userdata em uma lista:

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

Ao final do ciclo de coleta de lixo, os finalizadores para os objetos userdata são chamados na ordem *reversa* ao de sua criação, entre aqueles coletados naquele ciclo. Isto é, o primeiro finalizador a ser chamado é aquele associado com o objeto userdata que foi criado por último no programa. O userdata só é efetivamente liberado no próximo ciclo de coleta de lixo.

2.10.2 - Tabelas Fracas

Uma *tabela fraca* é uma tabela cujos elementos são *referências fracas*. Uma referência fraca é ignorada pelo coletor de

lixo. Em outras palavras, se as únicas referências para um objeto são referências fracas, então o coletor de lixo irá coletar este objeto.

Uma tabela fraca pode ter chaves fracas, valores fracos ou ambos. Uma tabela com chaves fracas permite a coleta de suas chaves mas impede a coleta de seus valores. Uma tabela com chaves fracas e valores fracos permite a coleta tanto das chaves como dos valores. Em qualquer caso, se a chave é coletada ou o valor é coletado, o par inteiro é removido da tabela. A fragilidade de uma tabela é controlada pelo campo `__mode` de sua metatabela. Se o campo `__mode` é uma cadeia de caracteres contendo o caractere '`k`', as chaves da tabela são fracas. Se `__mode` contém '`v`', os valores na tabela são fracos.

Depois de usar uma tabela como uma metatabela, não se deve mudar o valor de seu campo `__mode`. Caso contrário, o comportamento fraco das tabelas controladas por esta metatabela é indefinido.

2.11 - Co-rotinas

Lua oferece suporte a co-rotinas, também conhecidas como *fluxos de execução (threads) colaborativos*. Uma co-rotina em Lua representa um fluxo de execução independente. Ao contrário de processos leves em sistemas que dão suporte a múltiplos fluxos de execução, uma co-rotina somente suspende sua execução através de uma chamada explícita a uma função de cessão.

É possível criar uma co-rotina com uma chamada à `coroutine.create`. O seu único argumento é uma função que é a função principal da co-rotina. A função `create` somente cria uma nova co-rotina e retorna uma referência para ela (um objeto do tipo *thread*); ela não inicia a execução da co-rotina.

Quando a função `coroutine.resume` é chamada pela primeira vez, recebendo como seu primeiro argumento um objeto do tipo thread retornado por `coroutine.create`, a co-rotina inicia a sua execução, na primeira linha de sua função principal. Depois que a co-rotina começa a ser executada, ela continua executando até terminar ou *ceder*.

Uma função pode terminar sua execução de duas maneiras: normalmente, quando sua função principal retorna (explicitamente ou implicitamente, depois da última instrução); e de maneira anormal, se ocorre um erro não protegido. No primeiro caso, `coroutine.resume` retorna **true** mais quaisquer valores retornados pela função principal da co-rotina. No caso de acontecerem erros, `coroutine.resume` retorna **false** mais uma mensagem de erro.

Uma co-rotina cede a execução através de uma chamada à função `coroutine.yield`. Quando uma co-rotina cede, a `coroutine.resume` correspondente retorna imediatamente, mesmo se a cessão aconteceu dentro de uma chamada de função aninhada (isto é, não ocorreu dentro da função principal, mas em uma função chamada direta ou indiretamente pela função principal). No caso de uma cessão, `coroutine.resume` também retorna **true**, mais quaisquer valores passados para `coroutine.yield`. Na próxima vez que você recomeça a execução da mesma co-rotina, ela continua sua execução do ponto onde ela cedeu, com a chamada para `coroutine.yield` retornando quaisquer argumentos extras passados para `coroutine.resume`.

Como `coroutine.create`, a função `coroutine.wrap` também cria uma co-rotina, mas ao invés de retornar a própria co-rotina, ela retorna uma função que, quando chamada, retoma a execução da co-rotina. Quaisquer argumentos passados para esta função vão como argumentos extras para `coroutine.resume`. `coroutine.wrap` retorna todos os valores retornados por `coroutine.resume`, exceto o primeiro (o código booleano de erro). Diferentemente de `coroutine.resume`, `coroutine.wrap` não captura erros; qualquer erro é propagado para o chamador.

Como um exemplo, considere o seguinde 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"))
```

Quando você executá-lo, ele produzirá a seguinte saída:

```
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 - A Interface de Programação da Aplicação (API)

Esta seção descreve a API C para Lua, ou seja, o conjunto de funções C disponíveis para o programa hospedeiro se comunicar com Lua. Todas as funções da API, bem como os tipos e constantes relacionados, estão declarados no arquivo de cabeçalho `lua.h`.

Mesmo quando usamos o termo "função", qualquer operação na API pode, de forma alternativa, ser provida como uma macro. Tais macros usam cada um dos seus argumentos exatamente uma vez (com exceção do primeiro argumento, que é sempre um estado Lua) e portanto não geram qualquer efeito colateral oculto.

Como na maioria das bibliotecas C, as funções da API Lua não verificam a validade ou a consistência dos seus argumentos. Contudo, é possível mudar este comportamento compilando Lua com uma definição apropriada para a macro `luai_apicheck`, no arquivo `luaconf.h`.

3.1 - A Pilha

Lua usa uma *pilha virtual* para passar e receber valores de C. Cada elemento nesta pilha representa um valor Lua (**nil**, um número, uma cadeia de caracteres, etc.).

Sempre que Lua chama C, a função chamada recebe uma nova pilha, que é independente de pilhas anteriores e de pilhas de funções C que ainda estejam ativas. Esta pilha contém inicialmente quaisquer argumentos para a função C e é onde a função C empilha os seus resultados para serem retornados ao chamador (ver `lua_CFunction`).

Por conveniência, a maioria das operações de consulta na API não segue uma disciplina estrita de pilha. Ao invés disso, elas podem se referir a qualquer elemento na pilha usando um *índice*: Um índice positivo representa uma posição *absoluta* na pilha (começando em 1); um índice negativo representa uma posição *relativa* ao topo da pilha. De maneira mais específica, se a pilha possui *n* elementos, então o índice 1 representa o primeiro elemento (isto é, o elemento que foi empilhado na pilha primeiro) e o índice *n* representa o último elemento; o índice -1 também representa o último elemento (isto é, o elemento no topo) e o índice *-n* representa o primeiro elemento. Dizemos que um índice é *válido* se ele está entre 1 e o topo da pilha (isto é, se `1 ≤ abs(índice) ≤ topo`).

3.2 - Tamanho da Pilha

Quando você interage com a API de Lua, você é responsável por assegurar consistência. Em particular, *você é responsável por controlar estouro da pilha*. Você pode usar a função `lua_checkstack` para aumentar o tamanho da pilha.

Sempre que Lua chama C, ela assegura que pelo menos `LUA_MINSTACK` posições na pilha estão disponíveis. `LUA_MINSTACK` é definida como 20, então geralmente você não precisa se preocupar com o espaço da pilha a menos que o seu código possua laços empilhando elementos na pilha.

A maioria das funções de consulta aceita como índices qualquer valor dentro do espaço da pilha disponível, isto é, índices até o tamanho máximo da pilha que você configurou através da função `lua_checkstack`. Tais índices são chamados *índices aceitáveis*. Mais formalmente, definimos um *índice aceitável* como a seguir:

```
(índice < 0 && abs(índice) <= topo) ||
(índice > 0 && índice <= espaçodapilha)
```

Note que 0 nunca é um índice aceitável.

3.3 - Pseudo-Índices

A menos que seja dito o contrário, qualquer função que aceita índices válidos pode também ser chamada com *pseudo-índices*, que representam alguns valores Lua que são acessíveis para o código C mas que não estão na pilha. Pseudo-índices são usados para acessar o ambiente do fluxo de execução, o ambiente da função, o registro e os upvalues da função C (ver §3.4).

O ambiente do fluxo de execução (onde as variáveis globais existem) está sempre no pseudo-índice

LUA_GLOBALSINDEX. O ambiente da função C rodando está sempre no pseudo-índice LUA_ENVIRONINDEX.

Para acessar e mudar o valor de variáveis globais, você pode usar operações de tabelas usuais sobre uma tabela de ambiente. Por exemplo, para acessar o valor de uma variável global, faça

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

3.4 - Fechos C

Quando uma função C é criada, é possível associar alguns valores a ela, criando então um *fecho C*; estes valores são chamados de *upvalues* e são acessíveis para a função sempre que ela é chamada (ver lua_pushcclosure).

Sempre que uma função C é chamada, seus upvalues são posicionados em pseudo-índices específicos. Estes pseudo-índices são gerados pela macro lua_upvalueindex. O primeiro valor associado com uma função está na posição lua_upvalueindex(1), e assim por diante. Qualquer acesso a lua_upvalueindex(*n*), onde *n* é maior do que o número de upvalues da função atual (mas não é maior do que 256), produz um índice aceitável (embora inválido).

3.5 - Registro

Lua provê um *registro*, uma tabela pré-definida que pode ser usada por qualquer código C para armazenar qualquer valor Lua que o código C precise armazenar. Esta tabela está sempre localizada no pseudo-índice LUA_REGISTRYINDEX. Qualquer biblioteca de C pode armazenar dados nesta tabela, mas ela deve tomar cuidado para escolher chaves diferentes daquelas usadas por outras bibliotecas, para evitar colisões. Tipicamente, você deve usar como chave uma cadeia de caracteres contendo o nome da sua biblioteca ou um objeto do tipo userdata leve com o endereço de um objeto C em seu código.

As chaves inteiras no registro são usadas pelo mecanismo de referência, implementado pela biblioteca auxiliar, e portanto não devem ser usadas para outros propósitos.

3.6 - Tratamento de Erros em C

Internamente, Lua usa o mecanismo de longjmp de C para tratar erros. (Você pode também utilizar exceções se você usar C++; veja o arquivo luaconf.h.) Quando Lua se depara com qualquer erro (tais como erros de alocação de memória, erros de tipo, erros de sintaxe e erros de tempo de execução) ela *dispara* um erro; isto é, ela faz um desvio longo. Um *ambiente protegido* usa setjmp para estabelecer um ponto de recuperação; qualquer erro desvia o fluxo de execução para o ponto de recuperação ativado mais recentemente.

A maioria das funções na API pode disparar um erro, por exemplo devido a um erro de alocação de memória. A documentação para cada função indica se ela pode disparar erros.

Dentro de uma função C você pode disparar um erro chamando lua_error.

3.7 - Funções e Tipos

Listamos aqui todas as funções e tipos da API C em ordem alfabética. Cada função tem um indicador como este:

[-o, +p, *x*]

O primeiro campo, o, representa quantos elementos a função desempilha da pilha. O segundo campo, p, indica quantos elementos a função empilha na pilha. (Qualquer função sempre empilha seus resultados depois de desempilhar seus argumentos.) Um campo na forma x|y significa que a função pode empilhar (ou desempilhar) x ou y elementos, dependendo da situação; uma marca de interrogação '?' significa que não podemos saber quantos elementos a função desempilha/empilha olhando somente os seus argumentos (e.g., o número de elementos pode depender do que está na pilha). O terceiro campo, x, diz se a funçao pode disparar erros: '-' significa que a função nunca dispara qualquer erro; 'm' significa que a função pode disparar um erro somente devido à falta de memória; 'e' significa que a função pode disparar outros tipos de erro; 'v' significa que a função pode disparar um erro de maneira proposital.

lua_Alloc

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

O tipo da função de alocação de memória usada pelos estados Lua. A função de alocação deve prover uma funcionalidade similar à de realloc, mas não exatamente a mesma. Seus argumentos são ud, um ponteiro opaco passado para lua_newstate; ptr, um ponteiro para o bloco sendo alocado/realocado/liberado; osize, o tamanho

original do bloco; e `nsize`, o novo tamanho do bloco. `ptr` é `NULL` se e somente se `osize` é zero. Quando `nsize` é zero, a função de alocação deve retornar `NULL`; se `osize` é diferente de zero, o bloco de memória apontado por `ptr` deve ser liberado. Quando `nsize` não é zero, a função de alocação retorna `NULL` se e somente se ela não pode alocar o tamanho do bloco requisitado. Quando `nsize` não é zero e `osize` é zero, a função de alocação deve comportar-se como `malloc`. Quando `nsize` e `osize` não são zero, a função de alocação comporta-se como `realloc`. Lua assume que a função de alocação nunca falha quando `osize >= nsize`.

Temos a seguir uma implementação simples para a função de alocação. Ela é usada na biblioteca auxiliar por `luaL_newstate`.

```
static void *l_alloc (void *ud, void *ptr, size_t osize,
                                           size_t nsize) {
  (void)ud;  (void)osize;  /* não utilizados */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

Este código assume que `free(NULL)` não possui nenhum efeito e que `realloc(NULL, size)` é equivalente a `malloc(size)`. ANSI C garante esses dois comportamentos.

lua_atpanic

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

Estabelece uma nova função de pânico e retorna a função de pânico antiga.

Se um erro ocorre fora de qualquer ambiente protegido, Lua chama uma *função de pânico* e então chama `exit(EXIT_FAILURE)`, terminando então a aplicação hospedeira. A sua função de pânico pode evitar esta saída caso ela nunca retorne (e.g., fazendo uma desvio longo).

A função de pânico pode acessar a mensagem de erro no topo da pilha.

lua_call

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

Chama uma função.

Para chamar uma função você deve usar o seguinte protocolo: primeiro, a função a ser chamada é empilhada na pilha; em seguida, os argumentos da função são empilhados em ordem direta; isto é, o primeiro argumento é empilhado primeiro. Por último você chama `lua_call`; `nargs` é o número de argumentos que você empilhou na pilha. Todos os argumentos e o valor da função são desempilhados da pilha quando a função é chamada. Os resultados da função são empilhados na pilha quando a função retorna. O número de resultados é ajustado para `nresults`, a menos que `nresults` seja `LUA_MULTRET`. Neste caso, *todos* os resultados da função são empilhados. Lua cuida para que os valores retornados caibam dentro do espaço da pilha. Os resultados da função são empilhados na pilha em ordem direta (o primeiro resultado é empilhado primeiro), de modo que depois da chamada o último resultado está no topo da pilha.

Qualquer erro dentro da função chamada é propagado para cima (com um `longjmp`).

O seguinte exemplo mostra como o programa hospedeiro pode fazer o equivalente a este código Lua:

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

Aqui está o mesmo código em C:

```
lua_getfield(L, LUA_GLOBALSINDEX, "f");  /* função a ser chamada */
lua_pushstring(L, "how");                  /* primeiro argumento */
lua_getfield(L, LUA_GLOBALSINDEX, "t"); /* tabela a ser indexada */
lua_getfield(L, -1, "x"); /* empilha o resultado de t.x (2º arg) */
lua_remove(L, -2);                        /* remove 't' da pilha */
lua_pushinteger(L, 14);                          /* 3º argumento */
lua_call(L, 3, 1);   /* chama 'f' com 3 argumentos e 1 resultado */
lua_setfield(L, LUA_GLOBALSINDEX, "a"); /* estabelece 'a' global */
```

Note que o código acima é "balanceado": ao seu final, a pilha está de volta à sua configuração original. Isto é considerado uma boa prática de programação.

lua_CFunction

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

O tipo para funções C.

A fim de se comunicar apropriadamente com Lua, uma função C deve usar o seguinte protocolo, o qual define o modo como parâmetros e resultados são passados: uma função C recebe seus argumentos de Lua na sua pilha em ordem direta (o primeiro argumento é empilhado primeiro). Portanto, quando a função inicia, `lua_gettop(L)` retorna o número de argumentos recebidos pela função. O primeiro argumento (se houver) está no índice 1 e seu último argumento está no índice `lua_gettop(L)`. Para retornar valores para Lua, uma função C apenas os empilha na pilha, em ordem direta (o primeiro resultado é empilhado primeiro) e retorna o número de resultados. Qualquer outro valor na pilha abaixo dos resultados será devidamente descartado por Lua. Como uma função Lua, uma função C chamada por Lua também pode retornar muitos resultados.

Como um exemplo, a seguinte função recebe um número variável de argumentos numéricos e retorna a média e a soma deles:

```
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, "incorrect argument");
      lua_error(L);
    }
    sum += lua_tonumber(L, i);
  }
  lua_pushnumber(L, sum/n);  /* primeiro 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*]

Garante que existem pelo menos `extra` posições disponíveis na pilha. A função retorna falso se ela não puder aumentar o tamanho da pilha para o tamanho desejado. Esta função nunca comprime a pilha; se a pilha já é maior do que o novo tamanho, ela não terá o seu tamanho modificado.

lua_close

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

Destrói todos os objetos no estado Lua fornecido (chamando os metamétodos de coleta de lixo correspondentes, se houver) e libera toda a memória dinâmica usada por aquele estado. Em várias plataformas, pode não ser necessário chamar esta função, porque todos os recursos são naturalmente liberados quando o programa hospedeiro morre. Por outro lado, programas que ficam rodando por muito tempo, como um *daemon* ou um servidor web, podem precisar liberar estados tão logo eles não sejam mais necessários, para evitar um crescimento demasiado do uso da memória.

lua_concat

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

Concatena os `n` valores no topo da pilha, desempilha-os e deixa o resultado no topo da pilha. Se `n` é 1, o resultado é o único valor na pilha (isto é, a função não faz nada); se `n` é 0, o resultado é a cadeia de caracteres vazia. A concatenação é realizada de acordo com a semântica usual de Lua (ver §2.5.4).

lua_cpcall

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

Chama a função C `func` em modo protegido. `func` inicia somente com um único elemento na sua pilha, o objeto userdata leve contendo `ud`. Em caso de erros, `lua_cpcall` retorna o mesmo código de erro de `lua_pcall`, mais o objeto de erro no topo da pilha; caso contrário, ela retorna zero e não muda a pilha. Todos os valores retornados por `func` são descartados.

lua_createtable

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

Cria uma nova tabela vazia e a empilha no topo da pilha. A nova tabela possui espaço pré-alocado para `narr` elementos array e `nrec` elementos não-array. Esta pré-alocação é útil quando você sabe exatamente quantos elementos a tabela irá ter. Caso contrário você pode usar a função `lua_newtable`.

lua_dump

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

Descarrega uma função como um trecho de código binário. Recebe um função Lua no topo da pilha e produz um trecho de código binário que, se carregado novamente, resulta em uma função equivalente àquela que foi descarregada. Para produzir partes do trecho de código, `lua_dump` chama a função `writer` (ver `lua_Writer`) com o argumento `data` fornecido para escrevê-los.

O valor retornado é o código de erro retornado pela última chamada à função `writer`; 0 significa que não ocorreram erros.

Esta função não desempilha a função Lua da pilha.

lua_equal

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

Retorna 1 se os dois valores nos índices aceitáveis `index1` e `index2` são iguais, seguindo a semântica do operador == de Lua (ou seja, pode chamar metamétodos). Caso contrário retorna 0. Também retorna 0 se qualquer um dos índices não é válido.

lua_error

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

Gera um erro Lua. A mensagem de erro (que pode ser de fato um valor Lua de qualquer tipo) deve estar no topo da pilha. Esta função faz um desvio longo e portanto nunca retorna. (ver `luaL_error`).

lua_gc

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

Controla o coletor de lixo.

Esta função realiza várias tarefas, de acordo com o valor do parâmetro `what`:

- **`LUA_GCSTOP`:** pára o coletor de lixo.
- **`LUA_GCRESTART`:** reinicia o coletor de lixo.
- **`LUA_GCCOLLECT`:** realiza um ciclo completo de coleta de lixo.
- **`LUA_GCCOUNT`:** retorna a quantidade de memória (em Kbytes) que está sendo usada correntemente por Lua.
- **`LUA_GCCOUNTB`:** retorna o resto da divisão da quantidade de bytes de memória usada correntemente por Lua por 1024.
- **`LUA_GCSTEP`:** realiza um passo incremental de coleta de lixo. O "tamanho" do passo é controlado por `data` (valores maiores significam mais passos) de maneira não especificada. Se você quer controlar o tamanho do passo você deve ajustar de maneira experimental o valor de `data`. A função retorna 1 se o passo finalizou um ciclo de coleta de lixo
- **`LUA_GCSETPAUSE`:** estabelece `data` como o novo valor para a *pausa* do coletor (ver §2.10). A função retorna o valor anterior da pausa.
- **`LUA_GCSETSTEPMUL`:** estabelece `data` como o novo valor para o *multiplicador de passo* do coletor (ver §2.10). A função retorna o valor anterior do multiplicador de passo.

lua_getallocf

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

Retorna a função de alocação de memória de um dado estado. Se `ud` não é `NULL`, Lua armazena em `*ud` o ponteiro opaco passado para `lua_newstate`.

lua_getfenv

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

Coloca na pilha a tabela de ambiente do valor no índice fornecido.

lua_getfield

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

Coloca na pilha o valor `t[k]`, onde `t` é o valor no índice válido fornecido. Como em Lua, esta função pode disparar um metamétodo para o evento "index" (ver §2.8).

lua_getglobal

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

Coloca na pilha o valor da global `name`. Esta função é definida como uma 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 na pilha a metatabela do valor no índice aceitável fornecido. Se o índice não é válido ou se o valor não possui uma metatabela, a função retorna 0 e não coloca nada na pilha.

lua_gettable

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

Coloca na pilha o valor `t[k]`, onde `t` é o valor no índice válido fornecido e `k` é o valor no topo da pilha.

Esta função desempilha a chave '`k`' (colocando o resultado no seu lugar). Como em Lua, esta função pode disparar um metamétodo para o evento "index" (ver §2.8).

lua_gettop

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

Retorna o índice do elemento no topo da pilha. Visto que os índices começam em 1, este resultado é igual ao número de elementos na pilha (e portanto 0 significa uma pilha vazia).

lua_insert

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

Move o elemento no topo para o índice válido fornecido, deslocando os elementos acima deste índice para abrir espaço. Esta função não pode ser chamada com um pseudo-índice, porque um pseudo-índice não é uma posição real da pilha.

lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

O tipo usado pela API Lua para representar valores inteiros.

O tipo padrão é um `ptrdiff_t`, que é usualmente o maior tipo inteiro com sinal que a máquina manipula "confortavelmente".

lua_isboolean

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

Retorna 1 se o valor no índice aceitável fornecido possui tipo booleano e 0 caso contrário.

lua_iscfunction

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

Retorna 1 se o valor no índice aceitável fornecido é uma função C e 0 caso contrário.

lua_isfunction

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

Retorna 1 se o valor no índice aceitável fornecido é uma função (C ou Lua) e 0 caso contrário.

lua_islightuserdata

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

Retorna 1 se o valor no índice aceitável fornecido é um objeto userdata leve e 0 caso contrário.

lua_isnil

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

Retorna 1 se o valor no índice aceitável fornecido é **nil** e 0 caso contrário.

lua_isnone

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

Retorna 1 se o índice aceitável fornecido não é válido (isto é, se ele se refere a um elemento fora do espaço da pilha corrente) e 0 caso contrário.

lua_isnoneornil

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

Retorna 1 se o índice aceitável fornecido não é válido (isto é, se ele se refere a um elemento fora do espaço da pilha corrente) ou se o valor neste índice é **nil** e 0 caso contrário.

lua_isnumber

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

Retorna 1 se o valor no índice aceitável fornecido é um número ou uma cadeia de caracteres que pode ser convertida para um número e 0 caso contrário.

lua_isstring

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

Retorna 1 se o valor no índice aceitável fornecido é uma cadeia de caracteres ou um número (o qual sempre pode ser convertido para uma cadeia) e 0 caso contrário.

lua_istable

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

Retorna 1 se o valor no índice aceitável fornecido é uma tabela e 0 caso contrário.

lua_isthread

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

Retorna 1 se o valor no índice aceitável fornecido é do tipo thread e 0 caso contrário.

lua_isuserdata

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

Retorna 1 se o valor no índice aceitável fornecido é um objeto userdata (completo ou leve) e 0 caso contrário.

lua_lessthan

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

Retorna 1 se o valor no índice aceitável `index1` é menor do que o valor no índice aceitável `index2`, seguindo a semântica do operador < de Lua (ou seja, pode chamar metamétodos). Caso contrário retorna 0. Também retorna 0 se qualquer um dos índices não for válido.

lua_load

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

Carrega um trecho de código Lua. Se não ocorrer nenhum erro, `lua_load` empilha o trecho compilado como uma função Lua no topo da pilha. Caso contrário, empilha uma mensagem de erro. Os valores de retorno de `lua_load` são:

- **0:** sem erros;
- **`LUA_ERRSYNTAX`:** erro de sintaxe durante a pré-compilação;
- **`LUA_ERRMEM`:** erro de alocação de memória.

Esta função somente carrega um trecho; ela não o executa.

`lua_load` automaticamente detecta se o trecho está na forma de texto ou na forma binária e o carrega de maneira correta (veja o programa `luac`).

A função `lua_load` usa uma função `reader` fornecida pelo usuário para ler o trecho de código (ver `lua_Reader`). O argumento `data` é um valor opaco passado para a função de leitura.

O argumento `chunkname` dá um nome ao trecho, o qual é usado para mensagens de erro e em informações de depuração (ver §3.8).

lua_newstate

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

Cria um estado novo independente. Retorna `NULL` se não puder criar o estado (devido à falta de memória). O argumento `f` é a função de alocação; Lua faz toda a alocação de memória para este estado através desta função. O segundo argumento, `ud`, é um ponteiro opaco que Lua simplesmente passa para a função de alocação a cada chamada.

lua_newtable

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

Cria uma nova tabela vazia e a coloca na pilha. É equivalente a `lua_createtable(L, 0, 0)`.

lua_newthread

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

Cria um novo objeto do tipo thread, coloca-o na pilha e retorna um ponteiro para um `lua_State` que representa este novo fluxo de execução. O novo fluxo de execução retornado por esta função compartilha todos os objetos globais (tais como tabelas) com o estado original, mas possui uma pilha de execução independente.

Não há uma função explícita para terminar ou destruir um fluxo de execução. Objetos do tipo thread estão sujeitos à

coleta de lixo, assim como qualquer outro objeto de Lua.

lua_newuserdata

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

Esta função aloca um novo bloco de memória com o tamanho fornecido, coloca na pilha um novo objeto userdata completo com o endereço do bloco e retorna este endereço.

Objetos userdata representam valores C em Lua. Um *userdata completo* representa um bloco de memória. Ele é um objeto (assim como uma tabela): você deve criá-lo, ele pode ter sua própria metatabela e você pode detectar quando ele está sendo coletado. Um objeto userdata completo somente é igual a ele mesmo (usando a igualdade primitiva, sem o uso de metamétodos).

Quando Lua coleta um userdata completo com um metamétodo `gc`, Lua chama o metamétodo e marca o userdata como finalizado. Quando este userdata é coletado novamente então Lua libera sua memória correspondente.

lua_next

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

Desempilha uma chave da pilha e empilha um par chave-valor da tabela no índice fornecido (o "próximo" par depois da chave fornecida). Se não há mais elementos na tabela, então `lua_next` retorna 0 (e não empilha nada).

Um percorrimento típico parece com este:

```
/* tabela está na pilha no índice 't'' */
lua_pushnil(L);  /* primeira chave */
while (lua_next(L, t) != 0) {
  /* usa 'key' (no índice -2) e 'value' (no índice -1) */
  printf("%s - %s\n",
         lua_typename(L, lua_type(L, -2)),
         lua_typename(L, lua_type(L, -1)));
  /* remove 'value'; guarda 'key'  para a próxima iteração */
  lua_pop(L, 1);
}
```

Durante o percorrimento de uma tabela, não chame `lua_tolstring` diretamente sobre uma chave, a menos que você saiba que a chave é realmente uma cadeia de carecteres. Lembre-se que `lua_tolstring` *altera* o valor no índice fornecido; isto confunde a próxima chamada para `lua_next`.

lua_Number

```
typedef double lua_Number;
```

O tipo de números em Lua. Por padrão, ele é double, mas pode ser mudado em `luaconf.h`.

Através do arquivo de configuração é possível mudar Lua para operar com outro tipo para números (e.g., float ou long).

lua_objlen

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

Retorna o "comprimento" do valor no índice aceitável fornecido: para cadeias de caracteres, isto é o comprimento da cadeia; para tabelas, isto é o resultado do operador de comprimento ('#'); para objetos do tipo userdata, isto é o tamanho do bloco de memória alocado para o userdata; para outros valores, o tamanho é 0.

lua_pcall

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

Chama uma função em modo protegido.

Tanto `nargs` quanto `nresults` possuem o mesmo significado que possuíam em `lua_call`. Se não há erros durante a chamada, `lua_pcall` comporta-se exatamente como `lua_call`. Contudo, se há qualquer erro, `lua_pcall` o captura, coloca um único valor na pilha (a mensagem de erro) e retorna um código de erro. Como `lua_call`, `lua_pcall` sempre remove a função e seus argumentos da pilha.

Se `errfunc` é 0, então a mensagem de erro retornada na pilha é exatamente a mensagem de erro original. Caso contrário, `errfunc` é o índice na pilha de um *função de tratamento de erros*. (Na implementação atual, este índice não pode ser um pseudo-índice.) No caso de erros de tempo de execução, esta função será chamada com a mensagem de erro e seu valor de retorno será a mensagem retornada na pilha por `lua_pcall`.

Tipicamente, a função de tratamento de erros é usada para adicionar mais informação de depuração à mensagem de erro, como um traço da pilha. Tal informação não pode ser obtida após o retorno de `lua_pcall`, pois neste ponto a pilha já foi desfeita.

A função `lua_pcall` retorna 0 em caso de sucesso ou um dos seguintes códigos de erro (definidos em `lua.h`):

- **`LUA_ERRRUN`:** um erro de tempo de execução.
- **`LUA_ERRMEM`:** erro de alocação de memória. Para tais erros, Lua não chama a função de tratamento de erros.
- **`LUA_ERRERR`:** erro quando estava executando a função de tratamento de erros.

lua_pop

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

Desempilha `n` elementos da pilha.

lua_pushboolean

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

Empilha um valor booleano com valor `b` na pilha.

lua_pushcclosure

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

Empilha um novo fecho C na pilha.

Quando uma função C é criada, é possível associar alguns valores a ela, criando então um fecho C (ver §3.4); estes valores são então acessíveis para a função sempre que ela é chamada. Para associar valores com uma função C, primeiro estes valores devem ser colocados na pilha (quando há múltiplos valores, o primeiro valor é empilhado primeiro). Então `lua_pushcclosure` é chamada para criar e colocar a função C na pilha, com o argumento `n` informando quantos valores devem ser associados com a função. `lua_pushcclosure` também desempilha estes valores da pilha.

O valor máximo para `n` é 255.

lua_pushcfunction

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

Empilha uma função C na pilha. Esta função recebe um ponteiro para uma função C e coloca na pilha um valor Lua do tipo `function` que, quando chamado, invoca a função C correspondente.

Qualquer função para ser registrada em Lua deve seguir o protocolo correto para receber seus parâmetros e retornar seus resultados (ver `lua_CFunction`).

`lua_pushcfunction` é definida como uma 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 na pilha uma cadeia de caracteres formatada e retorna um ponteiro para esta cadeia. Ela é similar à função C `sprintf`, mas possui algumas diferenças importantes:

- Você não precisa alocar espaço para o resultado: o resultado é uma cadeia de caracteres e Lua cuida da alocação de memória (e da desalocação, através da coleta de lixo).
- Os especificadores de conversão são bastante restritos. Não há *flags*, tamanhos ou precisões. Os especificadores de conversão podem ser somente '`%%`' (insere um '`%`' na cadeia), '`%s`' (insere uma cadeia terminada por zero, sem restrições de tamanho), '`%f`' (insere um `lua_Number`), '`%p`' (insere um ponteiro como um número hexadecimal),

'%d' (insere um int) e '%c' (insere um int como um caractere).

lua_pushinteger

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

Coloca um número com valor n na pilha.

lua_pushliteral

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

Esta macro é equivalente a lua_pushlstring, mas somente pode ser usada quando s é uma cadeia literal. Nestes casos, a macro automaticamente provê o comprimento da cadeia.

lua_pushlightuserdata

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

Coloca um objeto do tipo userdata leve na pilha.

Um userdata representa valores de C em Lua. Um *userdata leve* representa um ponteiro. Ele é um valor (como um número): você não pode criá-lo, ele não possui uma metatabela individual e ele não é coletado (uma vez que ele nunca foi criado). Um userdata leve é igual a "qualquer" userdata leve com o mesmo endereço C.

lua_pushlstring

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

Empilha a cadeia de caracteres apontada por s com tamanho len na pilha. Lua cria (ou reusa) uma cópia interna da cadeia fornecida, de forma que a memória apontada por s pode ser liberada ou reutilizada imediatamente após o retorno da função. A cadeia pode conter zeros dentro dela.

lua_pushnil

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

Coloca um valor nil na pilha.

lua_pushnumber

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

Coloca um número com valor n na pilha.

lua_pushstring

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

Empilha a cadeia terminada por zero apontada por s na pilha. Lua cria (ou reusa) uma cópia interna da cadeia fornecida, de forma que a memória apontada por s pode ser liberada ou reutilizada imediatamente após o retorno da função. A cadeia não pode conter zeros dentro dela; assume-se que a cadeia termina no primeiro zero.

lua_pushthread

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

Empilha o fluxo de execução representado por L na pilha. Retorna 1 se este fluxo de execução é o fluxo de execução principal do seu estado.

lua_pushvalue

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

Empilha uma cópia do elemento no índice válido fornecido na pilha.

lua_pushvfstring

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

Equivalente a `lua_pushfstring`, exceto que esta função recebe uma `va_list` ao invés de um número variável de argumentos.

lua_rawequal

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

Retorna 1 se os dois valores nos índices aceitáveis `index1` e `index2` são iguais primitivamente (isto é, sem fazer chamadas a metamétodos). Caso contrário retorna 0. Também retorna 0 se qualquer um dos índices não for válido.

lua_rawget

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

Similar a `lua_gettable`, mas faz um acesso primitivo (i.e., sem usar metamétodos).

lua_rawgeti

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

Coloca na pilha o valor `t[n]`, onde `t` é o valor no índice válido fornecido. O acesso é primitivo; isto é, ele não invoca metamétodos.

lua_rawset

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

Similar a `lua_settable`, mas faz uma atribuição primitiva (i.e., sem usar metamétodos).

lua_rawseti

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

Faz o equivalente a `t[n] = v`, onde `t` é o valor no índice válido fornecido e `v` é o valor no topo da pilha.

Esta função desempilha o valor da pilha. A atribuição é primitiva; isto é, ela não invoca metamétodos.

lua_Reader

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

A função de leitura usada por `lua_load`. Toda vez que ela precisa de outro pedaço do trecho, `lua_load` chama a função de leitura, passando junto o seu parâmetro `data`. A função de leitura deve retornar um ponteiro para um bloco de memória com um novo pedaço do trecho e atribuir a `*size` o tamanho do bloco. O bloco deve existir até que a função de leitura seja chamada novamente. Para sinalizar o fim do trecho, a função de leitura deve retornar `NULL` ou atribuir zero a `size`. A função de leitura pode retornar pedaços de qualquer tamanho maior do que zero.

lua_register

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

Estabelece a função C `f` como o novo valor da global `name`. Esta função é definida como uma 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, -]

Remove o elemento no índice válido fornecido, deslocando para baixo os elementos acima deste índice para preencher o buraco. Esta função não pode ser chamada com um pseudo-índice, visto que o pseudo-índice não é uma posição real da pilha.

lua_replace

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

Move o elemento do topo para a posição fornecida (e desempilha-o), sem deslocar qualquer elemento (substituindo portanto o valor na posição fornecida).

lua_resume

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

Inicia e recomeça uma co-rotina em um fluxo de execução.

Para iniciar uma co-rotina, você deve primeiro criar um novo fluxo de execução (ver `lua_newthread`); em seguida você deve colocar na sua pilha a função principal mais quaisquer argumentos; por último você chama `lua_resume`, com `narg` sendo o número de argumentos. Esta chamada retorna quando a co-rotina suspende ou finaliza sua execução. Quando ela retorna, a pilha contém todos os valores passados para `lua_yield` ou todos os valores retornados pelo corpo da função. `lua_resume` retorna `LUA_YIELD` se a co-rotina cede, 0 se a co-rotina termina sua execução sem erros ou um código de erro no caso de acontecerem erros (ver `lua_pcall`). No caso de erros, a pilha não é desfeita, de forma que você pode usar a API de depuração sobre ela. A mensagem de erro está no topo da pilha. Para reiniciar uma co-rotina, você deve colocar na pilha dela somente os valores a serem passados como resultados de `yield` e então chamar `lua_resume`.

lua_setallocf

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

Muda a função de alocação de um dado estado para `f` com objeto userdata `ud`.

lua_setfenv

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

Desempilha uma tabela da pilha e estabelece esta tabela como sendo o novo ambiente para o valor no índice fornecido. Se o valor no índice fornecido não é nem uma função, nem um fluxo de execução e nem um objeto userdata, `lua_setfenv` retorna 0. Caso contrário a função retorna 1.

lua_setfield

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

Faz o equivalente a `t[k] = v`, onde `t` é o valor no índice válido fornecido e `v` é o valor no topo da pilha.

Esta função desempilha o valor da pilha. Como em Lua, esta função pode disparar um metamétodo para o evento "newindex" (ver §2.8).

lua_setglobal

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

Desempilha um valor da pilha e o estabelece como o novo valor da global `name`. Esta função é definida como uma macro:

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

lua_setmetatable

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

Desempilha uma tabela da pilha e estabelece esta tabela como a nova metatabela para o valor no índice aceitável fornecido.

lua_settable

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

Faz o equivalente a `t[k] = v`, onde `t` é o valor no índice válido fornecido, `v` é o valor no topo da pilha e `k` é o valor logo abaixo do topo.

Esta função desempilha tanto a chave como o valor da pilha. Da mesma forma que em Lua, esta função pode disparar um metamétodo para o evento "newindex" (ver §2.8).

lua_settop

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

Aceita qualquer índice aceitável, ou 0, e estabelece este índice como o topo da pilha. Se o novo topo é maior do que o antigo, então os novos elementos são preenchidos com **nil**. Se `index` é 0, então todos os elementos da pilha são removidos.

lua_State

```
typedef struct lua_State lua_State;
```

Estrutura opaca que guarda o estado completo de um interpretador Lua. A biblioteca de Lua é totalmente reentrante: não existem variáveis globais. Toda a informação sobre um estado é mantida nesta estrutura.

Um ponteiro para este estado deve ser passado como o primeiro argumento para toda função na biblioteca, exceto para `lua_newstate`, que cria um novo estado Lua a partir do zero.

lua_status

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

Retorna o status do fluxo de execução `L`.

O status pode ser 0 para um fluxo de execução normal, um código de erro se o fluxo de execução terminou sua execução com um erro ou `LUA_YIELD` se o fluxo de execução está suspenso.

lua_toboolean

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

Converte um valor Lua no índice aceitável fornecido para um valor booleano C (0 ou 1). Como todos os testes em Lua, `lua_toboolean` retorna 1 para qualquer valor Lua diferente de **false** e de **nil**; caso contrário a função retorna 0. A função também retorna 0 quando chamada com um índice não válido. (Se você quiser aceitar somente valores booleanos de fato, use `lua_isboolean` para testar o tipo do valor.)

lua_tocfunction

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

Converte um valor no índice aceitável fornecido para uma função C. Tal valor deve ser uma função C; caso contrário, retorna `NULL`.

lua_tointeger

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

Converte o valor Lua no índice aceitável fornecido para o tipo inteiro com sinal `lua_Integer`. O valor Lua deve ser um número ou uma cadeia que pode ser convertida para um número (ver §2.2.1); caso contrário, `lua_tointeger` retorna 0.

Se o número não é um inteiro, ele é truncado de alguma maneira não especificada.

lua_tolstring

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

Converte o valor Lua no índice aceitável fornecido para uma cadeia C. Se `len` não é `NULL`, a função também estabelece `*len` como o comprimento da cadeia. O valor Lua deve ser uma cadeia de caracteres ou um número; caso contrário, a função retorna `NULL`. Se o valor é um número, então `lua_tolstring` também *muda o valor real na pilha para uma cadeia*. (Esta mudança confunde `lua_next` quando `lua_tolstring` é aplicada a chaves durante um percorrimento de tabela.)

`lua_tolstring` retorna um ponteiro totalmente alinhado para uma cadeia de caracteres dentro do estado Lua. Esta cadeia sempre tem um zero ('`\0`') após o seu último caractere (como em C), mas pode conter outros zeros no seu corpo. Visto que Lua possui coleta de lixo, não há garantia de que o ponteiro retornado por `lua_tolstring` será válido após o valor correspondente ser removido da pilha.

lua_tonumber

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

Converte o valor Lua no índice aceitável fornecido para o tipo C `lua_Number` (ver `lua_Number`). O valor Lua deve ser um número ou uma cadeia que pode ser convertida para um número (ver §2.2.1); caso contrário, `lua_tonumber` retorna 0.

lua_topointer

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

Converte o valor no índice aceitável fornecido para um ponteiro C genérico (`void*`). O valor pode ser um objeto userdata, uma tabela, um fluxo de execução ou uma função; objetos diferentes irão fornecer ponteiros diferentes. Não há maneira de converter o ponteiro de volta ao seu valor original.

Tipicamente esta função é usada somente para informações de depuração.

lua_tostring

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

Equivalente a `lua_tolstring` com `len` sendo igual a `NULL`.

lua_tothread

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

Converte o valor no índice aceitável fornecido para um fluxo de execução (representado como `lua_State*`). Este valor deve ser um fluxo de execução; caso contrário, a função retorna `NULL`.

lua_touserdata

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

Se o valor no índice aceitável fornecido é um objeto userdata completo, a função retorna o endereço do seu bloco. Se o valor é um userdata leve, a função retorna seu ponteiro. Caso contrário, retorna `NULL`.

lua_type

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

Retorna o tipo do valor no índice aceitável fornecido ou `LUA_TNONE` para um índice não válido (isto é, um índice para uma posição da pilha "vazia"). Os tipos retornados por `lua_type` são codificados pelas seguintes constantes definidas

em `lua.h`: `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD` e `LUA_TLIGHTUSERDATA`.

lua_typename

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

Retorna o nome do tipo codificado pelo valor `tp`, que deve ser um dos valores retornados por `lua_type`.

lua_Writer

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

O tipo da função de escrita usada por `lua_dump`. Toda vez que ela produz outro pedaço de trecho, `lua_dump` chama a função de escrita, passando junto o buffer a ser escrito (`p`), seu tamanho (`sz`) e o parâmetro `data` fornecido para `lua_dump`.

A função de escrita retorna um código de erro: 0 significa nenhum erro; qualquer outro valor significa um erro e faz `lua_dump` parar de chamar a função de escrita.

lua_xmove

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

Troca valores entre diferentes fluxos de execução do *mesmo* estado global.

Esta função desempilha `n` valores da pilha `from` e os empilha na pilha `to`.

lua_yield

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

Cede uma co-rotina.

Esta função somente deve ser chamada como a expressão de retorno de uma função C, como a seguir:

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

Quando uma função C chama `lua_yield` desta maneira, a co-rotina sendo executada suspende a sua execução e a chamada a `lua_resume` que iniciou esta co-rotina retorna. O parâmetro `results` é o número de valores da pilha que são passados como resultados para `lua_resume`.

3.8 - A Interface de Depuração

Lua não possui mecanismos de depuração pré-definidos. Ao invés disto, ela oferece uma interface especial por meio de funções e *ganchos*. Esta interface permite a construção de diferentes tipos de depuradores, medidores e outras ferramentas que necessitam de "informação interna" do interpretador.

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) número de upvalues */
  int linedefined;            /* (S) */
  int lastlinedefined;        /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* parte privada */
  outros campos
```

```
} lua_Debug;
```

Uma estrutura usada para guardar diferentes pedaços de informação sobre uma função ativa. `lua_getstack` preenche somente a parte privada desta estrutura, para uso posterior. Para preencher os outros campos de `lua_Debug` com informação útil, chame `lua_getinfo`.

Os campos de `lua_Debug` possuem o seguinte significado:

- **`source`:** Se a função foi definida em uma cadeia de caracters, então `source` é essa cadeia. Se a função foi definida em um arquivo, então `source` inicia com um '`@`' seguido pelo nome do arquivo.
- **`short_src`:** uma versão "adequada" para impressão de `source`, para ser usada em mensagens de erro.
- **`linedefined`:** o número da linha onde a definição da função começa.
- **`lastlinedefined`:** o número da linha onde a definição da função termina.
- **`what`:** a cadeia `"Lua"` se a função é uma função Lua, `"C"` se ela é uma função C, `"main"` se ela é a parte principal de um trecho e `"tail"` se ela foi uma função que fez uma recursão final. No último caso, Lua não possui nenhuma outra informação sobre a função.
- **`currentline`:** a linha corrente onde a função fornecida está executando. Quando nenhuma informação sobre a linha está disponível, atribui-se -1 a `currentline`.
- **`name`:** um nome razoável para a função fornecida. Dado que funções em Lua são valores de primeira classe, elas não possuem um nome fixo: algumas funções podem ser o valor de múltiplas variáveis globais, enquanto outras podem estar armazenadas somente em um campo de uma tabela. A função `lua_getinfo` verifica como a função foi chamada para encontrar um nome adequado. Se não é possível encontrar um nome, então atribui-se `NULL` a `name`.
- **`namewhat`:** explica o campo `name`. O valor de `namewhat` pode ser `"global"`, `"local"`, `"method"`, `"field"`, `"upvalue"` ou `""` (a cadeia vazia), de acordo com como a função foi chamada. (Lua usa a cadeia vazia quando nenhuma outra opção parece se aplicar.)
- **`nups`:** o número de upvalues da função.

lua_gethook

```
lua_Hook lua_gethook (lua_State *L);
```
[-0, +0, -]

Retorna a função de gancho atual.

lua_gethookcount

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

Retorna a contagem de gancho atual.

lua_gethookmask

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

Retorna a máscara de gancho atual.

lua_getinfo

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

Retorna informação sobre uma função específica ou uma invocação de função específica.

Para obter informação sobre uma invocação de função, o parâmetro `ar` deve ser um registro de ativação válido que foi preenchido por uma chamada anterior a `lua_getstack` ou foi fornecido como argumento para um gancho (ver `lua_Hook`).

Para obter informação sobre uma função você deve colocá-la na pilha e iniciar a cadeia `what` com o caractere '>'. (Neste caso, `lua_getinfo` desempilha a função no topo da pilha.) Por exemplo, para saber em qual linha uma função `f` foi definida, você pode escrever o seguinte código:

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

Cada caractere na cadeia `what` seleciona alguns campos da estrutura `ar` para serem preenchidos ou um valor a ser empilhado na pilha:

- **'`n`':** preenche os campos `name` e `namewhat`;

- **'S':** preenche os campos `source`, `short_src`, `linedefined`, `lastlinedefined` e `what`;
- **'l':** preenche o campo `currentline`;
- **'u':** preenche o campo `nups`;
- **'f':** coloca na pilha a função que está executando no nível fornecido;
- **'L':** coloca na pilha uma tabela cujos índices são o número das linhas que são válidas na função. (Uma *linha válida* é uma linha com algum código associado, isto é, uma linha onde você pode colocar um ponto de parada. Linhas não válidas incluem linhas vazias e comentários.)

Esta função retorna 0 em caso de erro (por exemplo, no caso de uma opção inválida em `what`).

lua_getlocal

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

Obtém informação sobre uma variável local de um registro de ativação fornecido. O parâmetro `ar` deve ser um registro de ativação válido que foi preenchido por uma chamada anterior a `lua_getstack` ou foi fornecido como um argumento para um gancho (ver `lua_Hook`). O índice `n` seleciona qual variável local inspecionar (1 é o primeiro parâmetro ou variável local ativa e assim por diante, até a última variável local ativa). `lua_getlocal` coloca o valor da variável na pilha e retorna o nome dela.

Nomes de variáveis começando com '`(`' (abre parênteses) representam variáveis internas (variáveis de controle de laços, temporários e funções C locais.).

Retorna `NULL` (e não empilha nada) quando o índice é maior do que o número de variáveis locais ativas.

lua_getstack

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

Obtém informação sobre a pilha de tempo de execução do interpretador.

Esta função preenche partes de uma estrutura `lua_Debug` com uma identificação do *registro de ativação* da função executando em um dado nível. O nível 0 é a função executando atualmente, ao passo que o nível *n+1* é a função que chamou o nível *n*. Quando não há erros, `lua_getstack` retorna 1; quando chamada com um nível maior do que a profundidade da pilha, a função retorna 0.

lua_getupvalue

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

Obtém informação sobre um upvalue de um fecho. (Para funções Lua, upvalues são variáveis locais externas que a função usa e que são conceitualmente incluídas no fecho dela.) `lua_getupvalue` obtém o índice `n` de um upvalue, coloca o valor do upvalue na pilha e retorna o nome dele. `funcindex` aponta para o fecho na pilha. (Upvalues não possuem uma ordem específica, uma vez que eles são ativos ao longo de toda a função. Então, eles são numerados em uma ordem arbitrária.)

Retorna `NULL` (e não empilha nada) quando o índice é maior do que o número de upvalues. Para funções C, esta função usa a cadeia vazia `""` como um nome para todos os upvalues.

lua_Hook

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

O tipo para funções de gancho de depuração.

Sempre que um gancho é chamado, atribui-se ao campo `event` de seu argumento `ar` o evento específico que disparou o gancho. Lua identifica estes eventos com a seguintes constantes: `LUA_HOOKCALL`, `LUA_HOOKRET`, `LUA_HOOKTAILRET`, `LUA_HOOKLINE` e `LUA_HOOKCOUNT`. Além disso, para eventos de linha, o campo `currentline` também é atribuído. Para obter o valor de qualquer campo em `ar`, o gancho deve chamar `lua_getinfo`. Para eventos de retorno, `event` pode ser `LUA_HOOKRET`, o valor normal, ou `LUA_HOOKTAILRET`. No último caso, Lua está simulando um retorno de uma função que fez uma recursão final; neste caso, é inútil chamar `lua_getinfo`.

Enquanto Lua está executando um gancho, ela desabilita outras chamadas a ganchos. Portanto, se um gancho chama Lua de volta para executar uma função ou um trecho, esta execução ocorre sem quaisquer chamadas a ganchos.

lua_sethook

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

Estabelece a função de gancho de depuração.

O argumento `f` é uma função de gancho. `mask` especifica sobre quais eventos o gancho será chamado: ele é formado por uma conjunção bit-a-bit das constantes `LUA_MASKCALL`, `LUA_MASKRET`, `LUA_MASKLINE` e `LUA_MASKCOUNT`. O argumento `count` somente possui significado quando a máscara inclui `LUA_MASKCOUNT`. Para cada evento, o gancho é chamado como explicado abaixo:

- **O gancho de chamada (CALL):** é chamado quando o interpretador chama uma função. O gancho é chamado logo após Lua entrar na nova função, antes da função receber seus argumentos.
- **O gancho de retorno (RET):** é chamado quando o interpretador retorna de uma função. O gancho é chamado logo após Lua sair da função. Você não tem acesso aos valores a serem retornados pela função.
- **O gancho de linha (LINE):** é chamado quando o interpretador está para iniciar a execução de uma nova linha de código ou quando ele volta atrás no código (mesmo que para a mesma linha). (Este evento somente acontece quando Lua está executando uma função Lua.)
- **O gancho de contagem (COUNT):** é chamado após o interpretador executar cada uma das instruções `count`. (Este evento somente ocorre quando Lua está executando uma função Lua.)

Um gancho é desabilitado atribuindo-se zero a `mask`.

lua_setlocal

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

Estabelece o valor de uma variável local de um registro de ativação fornecido. Os parâmetros `ar` e `n` são como em `lua_getlocal` (ver `lua_getlocal`). `lua_setlocal` atribui o valor no topo da pilha à variável e retorna o nome dela. A função também desempilha o valor da pilha.

Retorna `NULL` (e não desempilha nada) quando o índice é maior do que o número de variáveis locais ativas.

lua_setupvalue

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

Estabelece o valor de um upvalue de um fecho. A função atribui o valor no topo da pilha ao upvalue e retorna o nome dele. Ela também desempilha o valor da pilha. Os parâmetros `funcindex` e `n` são como na função `lua_getupvalue` (ver `lua_getupvalue`).

Retorna `NULL` (e não desempilha nada) quando o índice é maior do que o número de upvalues.

4 - A Biblioteca Auxiliar

A *biblioteca auxiliar* fornece várias funções convenientes para a interface de C com Lua. Enquanto a API básica fornece as funções primitivas para todas as interações entre C e Lua, a biblioteca auxiliar fornece funções de mais alto nível para algumas tarefas comuns.

Todas as funções da biblioteca auxiliar são definidas no arquivo de cabeçalho `lauxlib.h` e possuem um prefixo `luaL_`.

Todas as funções na biblioteca auxiliar são construídas sobre a API básica e portanto elas não oferecem nada que não possa ser feito com a API básica.

Várias funções na biblioteca auxiliar são usadas para verificar argumentos de funções C. O nome delas sempre é `luaL_check*` ou `luaL_opt*`. Todas essas funções disparam um erro se a verificação não é satisfeita. Visto que a mensagem de erro é formatada para argumentos (e.g., "`bad argument #1`"), você não deve usar estas funções para outros valores da pilha.

4.1 - Funções e Tipos

Listamos aqui todas as funções e tipos da biblioteca auxiliar em ordem alfabética.

luaL_addchar

```
void luaL_addchar (luaL_Buffer *B, char c);
```
[-0, +0, *m*]

Adiciona o caractere `c` ao buffer `B` (ver `luaL_Buffer`).

luaL_addlstring

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```
[-0, +0, *m*]

Adiciona a cadeia de caracteres apontada por `s` com comprimento `l` ao buffer `B` (ver `luaL_Buffer`). A cadeia pode conter zeros dentro dela.

luaL_addsize

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```
[-0, +0, *m*]

Adiciona ao buffer `B` (ver `luaL_Buffer`) uma cadeia de comprimento `n` copiada anteriormente para a área de buffer (ver `luaL_prepbuffer`).

luaL_addstring

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```
[-0, +0, *m*]

Adiciona a cadeia terminada por 0 apontada por `s` ao buffer `B` (ver `luaL_Buffer`). A cadeia não pode conter zeros dentro dela.

luaL_addvalue

```
void luaL_addvalue (luaL_Buffer *B);
```
[-1, +0, *m*]

Adiciona o valor no topo da pilha ao buffer `B` (ver `luaL_Buffer`). Desempilha o valor.

Esta é a única função sobre buffers de cadeias que pode (e deve) ser chamada com um elemento extra na pilha, que é o valor a ser adicionado ao buffer.

luaL_argcheck

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

Verifica se `cond` é verdadeira. Se não, dispara um erro com a seguinte mensagem, onde `func` é recuperada a partir da pilha de chamada:

```
bad argument #<narg> to <func> (<extramsg>)
```

luaL_argerror

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

Dispara um erro com a seguinte mensagem, onde `func` é recuperada a partir da pilha de chamada:

```
bad argument #<narg> to <func> (<extramsg>)
```

Esta função nunca retorna, mas é idiomático usá-la em funções C como `return luaL_argerror(`*`args`*`)`.

luaL_Buffer

```
typedef struct luaL_Buffer luaL_Buffer;
```

O tipo para um *buffer de cadeia de caracteres*.

Um buffer de cadeia permite código C construir cadeias Lua pouco a pouco. O seu padrão de uso é como a seguir:

- Primeiro você declara uma variável `b` do tipo `luaL_Buffer`.
- Em seguida você a inicializa com uma chamada `luaL_buffinit(L, &b)`.
- Depois você adiciona pedaços da cadeia ao buffer chamando qualquer uma das funções `luaL_add*`.
- Você termina fazendo uma chamada `luaL_pushresult(&b)`. Esta chamada deixa a cadeia final no topo da

pilha.

Durante essa operação normal, um buffer de cadeia usa um número variável de posições da pilha. Então, quando você está usando um buffer, você não deve assumir que sabe onde o topo da pilha está. Você pode usar a pilha entre chamadas sucessivas às operações de buffer desde que este uso seja balanceado; isto é, quando você chama uma operação de buffer, a pilha está no mesmo nível em que ela estava imediatamente após a operação de buffer anterior. (A única exceção a esta regra é `luaL_addvalue`.) Após chamar `luaL_pushresult` a pilha está de volta ao seu nível quando o buffer foi inicializado, mais a cadeia final no seu topo.

luaL_buffinit

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

Inicializa um buffer `B`. Esta função não aloca qualquer espaço; o buffer deve ser declarado como uma variável (ver `luaL_Buffer`).

luaL_callmeta

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

Chama um metamétodo.

Se o objeto no índice `obj` possui uma metatabela e esta metatabela possui um campo `e`, esta função chama esse campo e passa o objeto como seu único argumento. Neste caso esta função retorna 1 e coloca na pilha o valor retornado pela chamada. Se não há metatabela ou metamétodo, esta função retorna 0 (sem empilhar qualquer valor na pilha).

luaL_checkany

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

Verifica se a função tem um argumento de qualquer tipo (incluindo **nil**) na posição `narg`.

luaL_checkint

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

Verifica se o argumento `narg` da função é um número e retorna este número convertido para um `int`.

luaL_checkinteger

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

Verifica se o argumento `narg` da função é um número e retorna este número convertido para um `lua_Integer`.

luaL_checklong

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

Verifica se o argumento `narg` da função é um número e retorna este número convertido para um `long`.

luaL_checklstring

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

Verifica se o argumento `narg` da função é uma cadeia e retorna esta cadeia; se `l` não é `NULL` preenche `*l` com o comprimento da cadeia.

Esta função usa `lua_tolstring` para obter seu resultado, de modo que todas as conversões e advertências relacionadas a `lua_tolstring` se aplicam aqui também.

luaL_checknumber

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

Verifica se o argumento `narg` da função é um número e retorna este número.

luaL_checkoption

```
int luaL_checkoption (lua_State *L,                                    [-0, +0, v]
                      int narg,
                      const char *def,
                      const char *const lst[]);
```

Verifica se o argumento `narg` da função é uma cadeia e procura por esta cadeia no array `lst` (o qual deve ser terminado por NULL). Retorna o índice no array onde a cadeia foi encontrada. Dispara um erro se o argumento não é uma cadeia ou se a cadeia não pôde ser encontrada.

Se `def` não é `NULL`, a função usa `def` como um valor padrão quando não há argumento `narg` ou se este argumento é **nil**.

Esta é uma função útil para mapear cadeias para enumerações de C. (A convenção usual em bibliotecas Lua é usar cadeias ao invés de números para selecionar opções.)

luaL_checkstack

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);          [-0, +0, v]
```

Aumenta o tamanho da pilha para `top + sz` elementos, disparando um erro se a pilha não pode ser aumentada para aquele tamanho. `msg` é um texto adicional a ser colocado na mensagem de erro.

luaL_checkstring

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

Verifica se o argumento `narg` da função é uma cadeia e retorna esta cadeia.

Esta função usa `lua_tolstring` para obter seu resultado, de modo que todas as conversões e advertências relacionadas a `lua_tolstring` se aplicam aqui também.

luaL_checktype

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

Verifica se o argumento `narg` da função tem tipo `t`. Veja `lua_type` para a codificação de tipos para `t`.

luaL_checkudata

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

Verifica se o argumento `narg` da função é um objeto userdata do tipo `tname` (ver `luaL_newmetatable`).

luaL_dofile

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

Carrega e executa o arquivo fornecido. É definida como a seguinte macro:

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

A função retorna 0 se não há erros ou 1 em caso de erros.

luaL_dostring

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

Carrega e executa a cadeia fornecida. É definida como a seguinte macro:

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

A função retorna 0 se não há erros ou 1 em caso de erros.

luaL_error

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

Dispara um erro. O formato da mensagem de erro é dado por `fmt` mais quaisquer argumentos extras, seguindo as mesmas regras de `lua_pushfstring`. Também adiciona no início da mensagem o nome do arquivo e o número da linha onde o erro ocorreu, caso esta informação esteja disponível.

Esta função nunca retorna, mas é idiomático usá-la em funções C como `return luaL_error(`*`args`*`)`.

luaL_getmetafield

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

Coloca na pilha o campo `e` da metatabela do objeto no índice `obj`. Se o objeto não possui uma metatabela ou se a metatabela não possui este campo, retorna 0 e não empilha nada.

luaL_getmetatable

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

Coloca na pilha a metatabela associada com o nome `tname` no registro (ver `luaL_newmetatable`).

luaL_gsub

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

Cria uma cópia da cadeia `s` substituindo qualquer ocorrência da cadeia `p` pela cadeia `r`. Coloca a cadeia resultante na pilha e a retorna.

luaL_loadbuffer

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

Carrega um buffer como um trecho de código Lua. Esta função usa `lua_load` para carregar o trecho no buffer apontado por `buff` com tamanho `sz`.

Esta função retorna os mesmos resultados de `lua_load`. `name` é o nome do trecho, usado para informações de depuração e mensagens de erro.

luaL_loadfile

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

Carrega um arquivo como um trecho de código Lua. Esta função usa `lua_load` para carregar o trecho no arquivo chamado `filename`. Se `filename` é `NULL`, então ela carrega a partir da entrada padrão. A primeira linha no arquivo é ignorada se ela começa com #.

Esta função retorna os mesmos resultados de `lua_load`, mas ela possui um código de erro extra `LUA_ERRFILE` se ela não pode abrir/ler o arquivo.

Da mesma forma que `lua_load`, esta função somente carrega o trecho; ela não o executa.

luaL_loadstring

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

Carrega uma cadeia como um trecho de código Lua. Esta função usa `lua_load` para carregar o trecho na cadeia (terminada por zero) `s`.

Esta função retorna os mesmos resultados de `lua_load`.

Assim como `lua_load`, esta função somente carrega o trecho; ela não o executa.

luaL_newmetatable

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

Se o registro já possui a chave `tname`, retorna 0. Caso contrário, cria uma nova tabela para ser usada como uma metatabela para o objeto userdata, adiciona esta tabela ao registro com chave `tname` e retorna 1.

Em ambos os casos coloca na pilha o valor final associado com `tname` no registro.

luaL_newstate

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

Cria um novo estado Lua. Chama `lua_newstate` com uma função de alocação baseada na função padrão de C `realloc` e então estabelece uma função de pânico (ver `lua_atpanic`) que imprime uma mensagem de erro para a saída de erro padrão em caso de erros fatais.

Retorna o novo estado ou `NULL` se ocorreu um erro de alocação de memória.

luaL_openlibs

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

Abre todas as bibliotecas padrões no estado fornecido.

luaL_optint

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

Se o argumento `narg` da função é um número, retorna este número convertido para um `int`. Se este argumento está ausente ou se ele é **nil**, retorna `d`. Caso contrário, dispara um erro.

luaL_optinteger

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

Se o argumento `narg` da função é um número, retorna este número convertido para um `lua_Integer`. Se este argumento está ausente ou se ele é **nil**, retorna `d`. Caso contrário, dispara um erro.

luaL_optlong

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

Se o argumento `narg` da função é um número, retorna este número convertido para um `long`. Se este argumento está ausente ou se ele é **nil**, retorna `d`. Caso contrário, dispara um erro.

luaL_optlstring

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

Se o argumento `narg` da função é uma cadeia, retorna esta cadeia. Se este argumento está ausente ou se ele **nil**, retorna `d`. Caso contrário, dispara um erro.

Se `l` não é `NULL`, preenche a posição `*l` com o comprimento do resultado.

luaL_optnumber

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

Se o argumento `narg` da função é um número, retorna este número. Se este argumento está ausente ou se ele é **nil**, retorna `d`. Caso contrário, dispara um erro.

luaL_optstring

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

Se o argumento `narg` da função é uma cadeia, retorna esta cadeia. Se este argumento está ausente ou se ele é **nil**, retorna `d`. Caso contrário, dispara um erro.

luaL_prepbuffer

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

Retorna um endereço para um espaço de tamanho `LUAL_BUFFERSIZE` onde você pode copiar uma cadeia para ser adicionada ao buffer `B` (ver `luaL_Buffer`). Após copiar a cadeia para este espaço você deve chamar `luaL_addsize` com o tamanho da cadeia para adicioná-la realmente ao buffer.

luaL_pushresult

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

Finaliza o uso do buffer `B` deixando a cadeia final no topo da pilha.

luaL_ref

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

Cria e retorna uma *referência*, na tabela no índice `t`, para o objeto no topo da pilha (e desempilha o objeto).

Uma referência é uma chave inteira única. Desde que você não adicione manualmente chaves inteiras na tabela `t`, `luaL_ref` garante a unicidade da chave que ela retorna. Você pode recuperar um objeto referenciado pelo referência `r` chamando `lua_rawgeti(L, t, r)`. A função `luaL_unref` libera uma referência e o objeto associado a ela.

Se o objeto no topo da pilha é **nil**, `luaL_ref` retorna a constante `LUA_REFNIL`. A constante `LUA_NOREF` é garantidamente diferente de qualquer referência retornada por `luaL_ref`.

luaL_Reg

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

O tipo para arrays de funções a serem registrados por `luaL_register`. `name` é o nome da função e `func` é um ponteiro para a função. Qualquer array de `luaL_Reg` deve terminar com uma entrada sentinela na qual tanto `name` como `func` são `NULL`.

luaL_register

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

Abre uma biblioteca.

Quando chamada com `libname` igual a `NULL`, simplesmente registra todas as funções na lista `l` (ver `luaL_Reg`) na tabela no topo da pilha.

Quando chamada com um valor de `libname` diferente de `NULL`, `luaL_register` cria uma nova tabela `t`, estabelece

ela como o valor da variável global `libname`, estabelece ela como o valor de `package.loaded[libname]` e registra nela todas as funções na lista `l`. Se existe uma tabela em `package.loaded[libname]` ou na variável `libname`, a função reusa esta tabela ao invés de criar uma nova.

Em qualquer caso a função deixa a tabela no topo da pilha.

luaL_typename

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

Retorna o nome do tipo do valor no índice fornecido.

luaL_typerror

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

Gera um erro com uma mensagem como a seguinte:

> *location*`: bad argument `*narg*` to '`*func*`' (`*tname*` expected, got `*rt*`)`

onde *location* é produzida por `luaL_where`, *func* é o nome da função corrente e *rt* é o nome do tipo do argumento.

luaL_unref

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

Libera a referência `ref` da tabela no índice `t` (ver `luaL_ref`). A entrada é removida da tabela, de modo que o objeto referenciado pode ser coletado. A referência `ref` também é liberada para ser usada novamente.

Se `ref` é `LUA_NOREF` ou `LUA_REFNIL`, `luaL_unref` não faz nada.

luaL_where

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

Coloca na pilha uma cadeia identificando a posição atual do controle no nível `lvl` na pilha de chamada. Tipicamente esta cadeia possui o seguinte formato:

> *chunkname*`:`*currentline*`:`

Nível 0 é a função executando correntemente, nível 1 é a função que chamou a função que está executando atualmente, etc.

Esta função é usada para construir um prefixo para mensagens de erro.

5 - Bibliotecas Padrão

As bibliotecas padrão de Lua oferecem funções úteis que são implementadas diretamente através da API C. Algumas dessas funções oferecem serviços essenciais para a linguagem (e.g., `type` e `getmetatable`); outras oferecem acesso a serviços "externos" (e.g., E/S); e outras poderiam ser implementadas em Lua mesmo, mas são bastante úteis ou possuem requisitos de desemepenho críticos que merecem uma implementação em C (e.g., `table.sort`).

Todas as bibliotecas são implementadas atráves da API C oficial e são fornecidas como módulos C separados. Correntemente, Lua possui as seguintes bibliotecas padrão:

- biblioteca básica, que inclui a sub-biblioteca de co-rotinas;
- biblioteca de pacotes;
- manipulação de cadeias de caracteres;
- manipulação de tabelas;
- funções matemáticas (sen, log, etc.);
- entrada e saída;
- facilidades do sistema operacional;
- facilidades de depuração.

Excetuando-se a biblioteca básica e a biblioteca de pacotes, cada biblioteca provê todas as suas funções como campos de uma tabela global ou como métodos de seus objetos.

Para ter acesso a essas bibliotecas, o programa hospedeiro C deve chamar a função `luaL_openlibs`, que abre todas as bibliotecas padrão. De modo alternativo, é possível abri-las individualmente chamando `luaopen_base` (para a biblioteca básica), `luaopen_package` (para a biblioteca de pacotes), `luaopen_string` (para a biblioteca de cadeias de caracteres), `luaopen_table` (para a biblioteca de tabelas), `luaopen_math` (para a biblioteca matemática), `luaopen_io` (para a biblioteca de E/S), `luaopen_os` (para a biblioteca do Sistema Operacional), e `luaopen_debug` (para a biblioteca de depuração). Essas funções estão declaradas em `lualib.h` e não devem ser chamadas diretamente: você deve chamá-las como qualquer outra função C de Lua, e.g., usando `lua_call`.

5.1 - Funções Básicas

A biblioteca de funções básicas oferece algumas funções essenciais a Lua. Se você não incluir esta biblioteca em sua aplicação, você deve verificar cuidadosamente se necessita fornecer implementações para algumas de suas facilidades.

assert (v [, message])

Produz um erro quando o valor de seu argumento `v` é falso (i.e., **nil** ou **false**); caso contrário, retorna todos os seus argumentos. `message` é uma mensagem de erro; quando ausente, a mensagem padrão é "assertion failed!"

collectgarbage (opt [, arg])

Esta função é uma interface genérica para o coletor de lixo. Ela realiza diferentes funções de acordo com o seu primeiro argumento, `opt`:

- **"stop":** pára o coletor de lixo.
- **"restart":** reinicia o coletor de lixo.
- **"collect":** realiza um ciclo de coleta de lixo completo.
- **"count":** retorna a memória total que está sendo usada por Lua (em Kbytes).
- **"step":** realiza um passo de coleta de lixo. O "tamanho" do passo é controlado por `arg` (valores maiores significam mais passos) de maneira não especificada. Se você quer controlar o tamanho do passo você deve ajustar de maneira experimental o valor de `arg`. Retorna **true** se o passo terminou um ciclo de coleta de lixo.
- **"setpause":** estabelece `arg` como o novo valor para a *pausa* do coletor (ver §2.10). Retorna o valor anterior para a *pausa*.
- **"setstepmul":** estabelece `arg` como o novo valor para o *multiplicador de passo* do coletor (ver §2.10). Retorna o valor anterior para a *pausa*.

dofile (filename)

Abre o arquivo indicado e executa o seu conteúdo como um trecho de código Lua. Quando chamada sem argumentos, `dofile` executa o conteúdo da entrada padrão (`stdin`). Retorna todos os valores retornados pelo trecho. Em caso de erros, `dofile` propaga o erro para o seu chamador (isto é, `dofile` não executa em modo protegido).

error (message [, level])

Termina a última função protegida chamada e retorna `message` como a mensagem de erro. A função `error` nunca retorna.

Geralmente, `error` adiciona alguma informação sobre a posição do erro no início da mensagem. O argumento `level` especifica como obter a posição do erro. Quando ele é igual a 1 (o padrão), a posição do erro é onde a função `error` foi chamada. Quando ele é 2, a posição do erro é onde a função que chamou `error` foi chamada; e assim por diante. Passando um valor 0 para `level` evita a adição de informação da posição do erro à mensagem.

_G

Uma variável global (não uma função) que armazena o ambiente global (isto é, `_G._G = _G`). Lua por si só não usa esta variável; uma modificação do seu valor não afeta qualquer ambiente e vice-versa. (Use `setfenv` para mudar ambientes.)

getfenv ([f])

Retorna o ambiente que está sendo usado correntemente pela função. `f` pode ser uma função Lua ou um número que especifica a função naquele nível de pilha: a função que chamou `getfenv` possui nível 1. Se a função fornecida não é uma função Lua ou se `f` é 0, `getfenv` retorna o ambiente global. O valor padrão para `f` é 1.

getmetatable (object)

Se `object` não possui uma metatabela, retorna **nil**. Caso contrário, se a metatabela do objeto possui um campo `"__metatable"`, retorna o valor associado. Caso contrário, retorna a metatabela do objeto fornecido.

ipairs (t)

Retorna três valores: uma função iteradora, a tabela `t` e 0, de modo que a construção

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

irá iterar sobre os pares `(1,t[1])`, `(2,t[2])`, ···, até a primeira chave inteira ausente da tabela.

load (func [, chunkname])

Carrega um trecho usando a função `func` para obter seus pedaços. Cada chamada a `func` deve retornar uma cadeia de caracteres que concatena com resultados anteriores. Quando `func` retorna uma cadeia vazia, **nil** ou quando não retorna nenhum valor, isso indica o fim do trecho.

Se não ocorrerem erros, retorna o trecho compilado como uma função; caso contrário, retorna **nil** mais a mensagem de erro. O ambiente da função retornada é o ambiente global.

`chunkname` é usado como o nome do trecho para mensagens de erro e informação de depuração. Quando ausente, o valor padrão é `"=(load)"`.

loadfile ([filename])

Similar a `load`, mas obtém o trecho do arquivo `filename` ou da entrada padrão, se nenhum nome de arquivo é fornecido.

loadstring (string [, chunkname])

Similar a `load`, mas obtém o trecho da cadeia fornecida.

Para carregar e rodar uma dada cadeia, use a expressão idiomática

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

Quando ausente, o valor padrão para `chunkname` é a cadeia fornecida.

next (table [, index])

Permite a um programa pecorrer todos os campos de uma tabela. Seu primeiro argumento é uma tabela e seu segundo argumento é um índice nesta tabela. `next` retorna o próximo índice da tabela e seu valor associado. Quando chamada com **nil** como seu segundo argumento, `next` retorna um índice inicial e seu valor associado. Quando chamada com o último índice ou com **nil** em uma tabela vazia, `next` retorna **nil**. Se o segundo argumento está ausente, então ele é interpretado como **nil**. Em particular, você pode usar `next(t)` para verificar se a tabela está vazia.

A ordem na qual os índices são enumerados não é especificada, *até mesmo para índices numéricos*. (Para percorrer uma tabela em ordem numérica, use o **for** numérico ou a função `ipairs`.).

O comportamento de `next` é *indefinido* se, durante o percorrimento, você atribuir qualquer valor a um campo não existente na tabela. Você pode contudo modificar campos existentes. Em particular, você pode limpar campos existentes.

pairs (t)

Retorna três valores: a função `next`, a tabela `t` e **nil**, de modo que a construção

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

irá iterar sobre todos os pares chave–valor da tabela `t`.

Veja a função `next` para os cuidados que se deve ter ao modificar a tabela durante o seu percorrimento.

pcall (f, arg1, ···)

Chama a função `f` com os argumentos fornecidos em *modo protegido*. Isto significa que qualquer erro dentro de `f` não é propagado; ao invés disso, `pcall` captura o erro e retorna um código indicando o status. Seu primeiro resultado é o código de status (um booleano), que é verdadeiro se a chamada aconteceu sem erros. Neste caso, `pcall` também retorna todos os resultados da chamada, depois deste primeiro resultado. No caso de acontecer um erro, `pcall` retorna **false** mais a mensagem de erro.

print (···)

Recebe qualquer número de argumentos e imprime os seus valores para `stdout`, usando a função `tostring` para convertê-los para cadeias de caracteres. `print` não é projetada para saída formatada, mas somente como uma maneira rápida de mostrar um valor, tipicamente para depuração. Para saída formatada, use `string.format`.

rawequal (v1, v2)

Verifica se `v1` é igual a `v2`, sem invocar nenhum metamétodo. Retorna um booleano.

rawget (table, index)

Obtém o valor real de `table[index]`, sem invocar nenhum metamétodo. `table` deve ser uma tabela; `index` pode ser qualquer valor.

rawset (table, index, value)

Atribui `value` como o valor real de `table[index]`, sem invocar nenhum metamétodo. `table` deve ser uma tabela, `index` pode ser qualquer valor diferente de **nil** e `value` pode ser qualquer valor Lua.

Essa função retorna `table`.

select (index, ···)

Se `index` é um número, retorna todos os argumentos após o argumento número `index`. Caso contrário, `index` deve ser a cadeia `"#"` e `select` retorna o número total de argumentos extras recebidos.

setfenv (f, table)

Estabelece o ambiente a ser usado pela função fornecida. `f` pode ser uma função Lua ou um número que especifica a função naquele nível de pilha: a função chamando `setfenv` possui nível 1. `setfenv` retorna a função fornecida.

Como um caso especial, quando `f` é 0 `setfenv` muda o ambiente do fluxo de execução corrente. Neste caso, `setfenv` não retorna nenhum valor.

setmetatable (table, metatable)

Estabelece a metatabela para a tabela fornecida. (Você não pode mudar a metatabela de outros tipos a partir de Lua, somente a partir de C.) Se `metatable` é **nil**, remove a metatabela da tabela fornecida. Se a metatabela original tem um campo `"__metatable"`, dispara um erro.

Essa função retorna `table`.

tonumber (e [, base])

Tenta converter seu argumento para um número. Se o argumento já é um número ou uma cadeia de caracteres que pode ser convertida para um número, então `tonumber` retorna este número; caso contrário, retorna **nil**.

Um argumento opcional especifica a base para interpretar o numeral. A base pode ser qualquer inteiro entre 2 e 36, inclusive. Em bases acima de 10, a letra '`A`' (maiúscula ou minúscula) representa 10, '`B`' representa 11 e assim por diante, com '`Z`' representando 35. Na base 10 (o padrão), o número pode ter uma parte decimal, bem como uma parte expoente opcional (ver §2.1). Em outras bases, somente inteiros sem sinal são aceitos.

tostring (e)

Recebe um argumento de qualquer tipo e o converte para uma cadeia de caracteres em um formato razoável. Para um controle completo de como números são convertidos, use `string.format`.

Se a metatabela de `e` possui um campo `"__tostring"`, então `tostring` chama o valor correspondente com `e` como argumento e usa o resultado da chamada como o seu resultado.

type (v)

Retorna o tipo de seu único argumento, codificado como uma cadeia de caracteres. Os resultados possíveis desta função são `"nil"` (uma cadeia de caracteres, não o valor **nil**), `"number"`, `"string"`, `"boolean"`, `"table"`, `"function"`, `"thread"` e `"userdata"`.

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

Retorna os elementos da tabela fornecida. Esta função é equivalente a

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

exceto que o código acima pode ser escrito somente para um número fixo de elementos. Por padrão, `i` é `1` e `j` é o comprimento da lista, como definido pelo operador de comprimento (ver §2.5.5).

_VERSION

Uma variável global (não uma função) que armazena uma cadeia contendo a versão corrente do interpretador. O conteúdo corrente desta variável é `"Lua 5.1"`.

xpcall (f, err)

Esta função é similar a `pcall`, exceto que você pode estabelecer um novo tratador de erros.

`xpcall` chama a função `f` em modo protegido, usando `err` como um tratador de erros. Qualquer erro dentro de `f` não é propagado; ao invés disso, `xpcall` captura o erro, chama a função `err` com o objeto de erro original e retorna um código indicando um status. Seu primeiro resultado é o código de status (um booleano), que é verdadeiro se a chamada ocorreu sem erros. Neste caso, `xpcall` também retorna todos os resultados da chamada, depois deste primeiro resultado. Em caso de erro, `xpcall` retorna **false** mais o resultado de `err`.

5.2 - Manipulação de Co-rotinas

As operações relacionadas a co-rotinas constituem uma sub-biblioteca da biblioteca básica e estão dentro da tabela `coroutine`. Veja §2.11 para uma descrição geral de co-rotinas.

coroutine.create (f)

Cria uma nova co-rotina, com corpo `f`. `f` deve ser uma função Lua. Retorna esta nova co-rotina, um objeto com tipo `"thread"`.

coroutine.resume (co [, val1, ···])

Inicia ou continua a execução da co-rotina `co`. Na primeira vez que você "continua" uma co-rotina, ela começa executando o seu corpo. Os valores `val1`, ··· são passados como os argumentos para o corpo da função. Se a co-rotina já cedeu a execução antes, `resume` a continua; os valores `val1`, ··· são passados como os resultados da cessão.

Se a co-rotina executa sem nenhum erro, `resume` retorna **true** mais quaisquer valores passados para `yield` (se a co-rotina cede) ou quaisquer valores retornados pelo corpo da função (se a co-rotina termina). Se há qualquer erro, `resume` retorna **false** mais a mensagem de erro.

coroutine.running ()

Retorna a co-rotina sendo executada ou **nil** quando chamada pelo fluxo de execução principal.

coroutine.status (co)

Retorna o status da co-rotina `co`, como uma cadeia de caracteres: `"running"`, se a co-rotina está executando (isto é, ela chamou `status`); `"suspended"`, se a co-rotina está suspensa em uma chamada a `yield` ou se ela não começou a sua execução ainda; `"normal"` se a co-rotina está ativa mas não está executando (isto é, ela continuou outra co-rotina); e `"dead"` se a co-rotina terminou sua função principal ou se ela parou com um erro.

coroutine.wrap (f)

Cria uma nova co-rotina, com corpo `f`. `f` deve ser uma função Lua. Retorna uma função que recomeça a co-rotina cada vez que é chamada. Quaisquer argumentos passados para a função comportam-se como os argumentos extras para `resume`. Retorna os mesmos valores retornados por `resume`, exceto o primeiro booleano. Em caso de erro, propaga o erro.

coroutine.yield (···)

Suspende a execução da co-rotina chamadora. A co-rotina não pode estar executando uma função C, um metamétodo ou um iterador. Quaisquer argumentos para `yield` são passados como resultados extras para `resume`.

5.3 - Módulos

A biblioteca de pacotes provê facilidades básicas para carregar e construir módulos em Lua. Ela exporta duas de suas funções diretamente no ambiente global: `require` e `module`. Todas as outras funções são exportadas em uma tabela `package`.

module (name [, ···])

Cria um módulo. Se há uma tabela em `package.loaded[name]`, esta tabela é o módulo. Caso contrário, se existe uma tabela global `t` com o nome fornecido, esta tabela é o módulo. Caso contrário cria uma nova tabela `t` e a estabelece como o valor da global `name` e o valor de `package.loaded[name]`. Esta função também inicializa `t._NAME` com o nome fornecido, `t._M` com o módulo (o próprio `t`) e `t._PACKAGE` com o nome do pacote (o nome do módulo completo menos o último componente; veja abaixo). Finalmente, `module` estabelece `t` como o novo ambiente da função corrente e o novo valor de `package.loaded[name]`, de modo que `require` retorna `t`.

Se `name` é um nome composto (isto é, um nome com componentes separados por pontos), `module` cria (ou reusa, se elas já existem) tabelas para cada componente. Por exemplo, se `name` é `a.b.c`, então `module` armazena a tabela do módulo no campo `c` do campo `b` da global `a`.

Esta função pode receber algumas *opções* depois do nome do módulo, onde cada opção é uma função a ser aplicada sobre o módulo.

require (modname)

Carrega o módulo fornecido. Esta função começa procurando na tabela `package.loaded` para determinar se `modname` já foi carregado. Em caso afirmativo, `require` retorna o valor armazenado em `package.loaded[modname]`. Caso contrário, ela tenta achar um *carregador* para o módulo.

Para encontrar um carregador, `require` é guiada pelo array `package.loaders`. Modificando este array, podemos mudar como `require` procura por um módulo. A seguinte explicação é baseada na configuração padrão para `package.loaders`.

Primeiro `require` consulta `package.preload[modname]`. Se existe um valor nesse campo, este valor (que deve ser uma função) é o carregador. Caso contrário `require` busca por um carregador Lua usando o caminho armazenado em `package.path`. Se isso também falha, ela busca por um carregador C usando o caminho armazenado em `package.cpath`. Se isso também falha, ela tenta um carregador *tudo-em-um* (ver `package.loaders`).

Uma vez que um carregador é encontrado, `require` chama o carregador com um único argumento, `modname`. Se o carregador retorna qualquer valor, `require` atribui o valor retornado a `package.loaded[modname]`. Se o carregador não retorna nenhum valor e não foi atribuído nenhum valor a `package.loaded[modname]`, então `require` atribui **true** a esta posição. Em qualquer caso, `require` retorna o valor final de `package.loaded[modname]`.

Se ocorre um erro durante o carregamento ou a execução do módulo ou se não é possível encontrar um carregador para o módulo, então `require` sinaliza um erro.

package.cpath

O caminho usado por `require` para procurar por um carregador C.

Lua inicializa o caminho C `package.cpath` da mesma forma que inicializa o caminho Lua `package.path`, usando a variável de ambiente `LUA_CPATH` ou um caminho padrão definido em `luaconf.h`.

package.loaded

Uma tabela usada por `require` para controlar quais módulos já foram carregados. Quando você requisita um módulo `modname` e `package.loaded[modname]` não é falso, `require` simplesmente retorna o valor armazenado lá.

package.loaders

Uma tabela usada por `require` para controlar como carregar módulos.

Cada posição nesta tabela é uma *função buscadora*. Quando está procurando um módulo, `require` chama cada uma destas funções buscadoras em ordem crescente, com o nome do módulo (o argumento fornecido a `require`) como seu único parâmetro. A função pode retornar outra função (o *carregador* do módulo) ou uma cadeia de caracteres explicando porque ela não achou aquele módulo (ou **nil** se ela não tem nada a dizer). Lua inicializa esta tabela com quatro funções.

A primeira função buscadora simplesmente procurar um carregador no tabela `package.preload`.

A segunda função buscadora procura um carregador como uma biblioteca Lua, usando o caminho armazenado em `package.path`. Um caminho é uma seqüência de *padrões* separados por ponto-e-vírgulas. Para cada padrão, a função buscadora irá mudar cada ponto de interrogação no padrão para `filename`, que é o nome do módulo com cada ponto substituído por um "separador de diretório" (como `"/"` no Unix); então ela tentará abrir o nome do arquivo resultante. Por exemplo, se o caminho é a cadeia de caracteres

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

a busca por um arquivo Lua para o módulo `foo` tentará abrir os arquivos `./foo.lua`, `./foo.lc` e `/usr/local/foo/init.lua`, nessa ordem.

A terceira função buscadora procura um carregador como uma biblioteca C, usando o caminho fornecido pela variável `package.cpath`. Por exemplo, se o caminho C é a cadeia

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

a função buscadora para o módulo `foo` tentará abrir os arquivos `./foo.so`, `./foo.dll` e `/usr/local/foo/init.so`, nessa ordem. Uma vez que ela encontra uma biblioteca C, esta função buscadora primeiro usa uma facilidade de ligação dinâmica para ligar a aplicação com a biblioteca. Então ela tenta encontrar uma função C dentro da biblioteca para ser usada como carregador. O nome desta função C é a cadeia `"luaopen_"` concatenada com uma cópia do nome do módulo onde cada ponto é substituído por um sublinhado. Além disso, se o nome do módulo possui um hífen, seu prefixo até (e incluindo) o primeiro hífen é removido. Por exemplo, se o nome do módulo é `a.v1-b.c`, o nome da função será `luaopen_b_c`.

A quarta função buscadora tenta um *carregador tudo-em-um*. Ela procura no caminho C uma biblioteca para a raiz do nome do módulo fornecido. Por exemplo, quando requisitando `a.b.c`, ela buscará por uma biblioteca C para `a`. Se encontrar, ela busca nessa biblioteca por uma função de abertura para o submódulo; no nosso exemplo, seria `luaopen_a_b_c`. Com esta facilidade, um pacote pode empacotar vários submódulos C dentro de uma única biblioteca, com cada submódulo guardando a sua função de abertura original.

package.loadlib (libname, funcname)

Liga dinamicamente o programa hospedeiro com a biblioteca C `libname`. Dentro desta biblioteca, procura por uma função `funcname` e retorna essa função como uma função C. (Desse modo, `funcname` deve seguir o protocolo (ver `lua_CFunction`)).

Esta é uma função de baixo nível. Ela contorna completamente o sistema de pacotes e de módulos. Diferentemente de `require`, ela não realiza qualquer busca de caminho e não adiciona extensões automaticamente. `libname` deve ser o nome do arquivo completo da biblioteca C, incluindo se necessário um caminho e uma extensão. `funcname` deve ser o nome exato exportado pela biblioteca C (que pode depender de como o compilador e o ligador C são usados).

Esta função não é provida por ANSI C. Dessa forma, ela está disponível somente em algumas plataformas (Windows, Linux, Mac OS X, Solaris, BSD, mais outros sistemas Unix que dão suporte ao padrão `dlfcn`).

package.path

O caminho usado por `require` para buscar um carregador Lua.

Ao iniciar, Lua inicializa esta variável com o valor da variável de ambiente `LUA_PATH` ou com um caminho padrão

definido em `luaconf.h`, se a variável de ambiente não está definida. Qualquer "`;;`" no valor da variável de ambiente será substituído pelo caminho padrão.

package.preload

Uma tabela para armazenar carregadores para módulos específicos (ver `require`).

package.seeall (module)

Estabelece uma metatabela para `module` com seu campo `__index` se referindo ao ambiente global, de modo que esse módulo herda valores do ambiente global. Para ser usada como uma opção à função `module`.

5.4 - Manipulação de Cadeias de Caracteres

Esta biblioteca provê funções genéricas para a manipulação de cadeias de caracteres, tais como encontrar e extrair subcadeias e casamento de padrões. Ao indexar uma cadeia em Lua, o primeiro caractere está na posição 1 (não na posição 0, como em C). Índices podem ter valores negativos e são interpretados como uma indexação de trás para frente, a partir do final da cadeia. Portanto, o último caractere está na posição -1 e assim por diante.

A biblioteca de cadeias provê todas as suas funções dentro da tabela `string`. Ela também estabelece uma metatabela para cadeias onde o campo `__index` aponta para a tabela `string`. Em conseqüência disso, você pode usar as funções de cadeias em um estilo orientado a objetos. Por exemplo, `string.byte(s, i)` pode ser escrito como `s:byte(i)`.

A biblioteca de manipulação de cadeias assume que cada caracter é codificado usando um byte.

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

Retorna o código numérico interno dos caracteres `s[i]`, `s[i+1]`, ···, `s[j]`. O valor padrão para `i` é 1; o valor padrão para `j` é `i`.

Note que códigos numéricos não são necessariamente portáveis entre plataformas.

string.char (···)

Recebe zero ou mais inteiros. Retorna uma cadeia com comprimento igual ao número de argumentos, na qual cada caractere possui um código numérico interno igual ao seu argumento correspondente.

Note que códigos numéricos não são necessariamente portáveis entre plataformas.

string.dump (function)

Retorna uma cadeia contendo a representação binária da função fornecida, de modo que um `loadstring` posterior nesta cadeia retorna uma cópia da função. `function` deve ser uma função Lua sem upvalues.

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

Procura o primeiro casamento do padrão `pattern` na cadeia `s`. Se a função acha um casamento, então `find` retorna os índices de `s` onde esta ocorrência começou e terminou; caso contrário, retorna **nil**. O terceiro argumento, `init`, é um valor numérico opcional e especifica onde iniciar a busca; seu valor padrão é 1 e pode ser negativo. Um valor **true** para o quarto argumento, `plain`, que é opcional, desabilita as facilidades de casamento de padrões, de modo que a função faz uma operação "encontra subcadeia" simples, sem considerar nenhum caractere em `pattern` como "mágico". Note que se `plain` é fornecido, então `init` deve ser fornecido também.

Se o padrão possui capturas, então em um casamento bem-sucedido os valores capturados são também retornados, após os dois índices.

string.format (formatstring, ···)

Retorna a versão formatada de seu número variável de argumentos seguindo a descrição dada no seu primeiro argumento (que deve ser uma cadeia). O formato da cadeia segue as mesmas regras da família `printf` de funções C padrão. As únicas diferenças são que as opções/modificadores `*`, `l`, `L`, `n`, `p` e `h` não são oferecidas e que há uma opção extra, `q`. A opção `q` formata uma cadeia em uma forma adequada para ser lida de volta de forma segura pelo

interpretador Lua; a cadeia é escrita entre aspas duplas e todas as aspas duplas, quebras de linha, barras invertidas e zeros dentro da cadeia são corretamente escapados quando escritos. Por exemplo, a chamada

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

produzirá a cadeia:

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

As opções `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X` e `x` esperam um número como argumento, enquanto que `q` e `s` esperam uma cadeia.

Esta função não aceita valores de cadeias contendo zeros dentro delas, exceto quando esses valores são argumentos para a opção `q`.

string.gmatch (s, pattern)

Retorna uma função iteradora que, cada vez que é chamada, retorna a próxima captura de `pattern` na cadeia `s`. Se `pattern` não especifica nenhuma captura, então o casamento inteiro é produzido a cada chamada. Como um exemplo, o seguinte laço

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

irá iterar sobre todas as palavras da cadeia `s`, imprimindo uma por linha. O próximo exemplo coleta todos os pares `key=value` da cadeia fornecida e os coloca em uma tabela:

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

Para essa função, um '`^`' no início de um padrão não funciona como uma âncora, visto que isso iria impedir a iteração.

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

Retorna uma cópia de `s` na qual todas as (ou as primeiras `n`, se fornecido) ocorrências de `pattern` são substituídas por uma cadeia de substituição especificada por `repl`, que pode ser uma cadeia, uma tabela ou uma função. `gsub` também retorna, como seu segundo valor, o número total de substituições que ocorreram.

Se `repl` é uma cadeia, então seu valor é usado para a substituição. O caractere `%` funciona como um caractere de escape: qualquer seqüência em `repl` da forma `%`*n*, com *n* entre 1 e 9, representa o valor da *n*-ésima subcadeia capturada (veja abaixo). A seqüência `%0` representa o casamento inteiro. A seqüência `%%` representa um `%` simples.

Se `repl` é uma tabela, então a tabela é consultada a cada casamento, usando a primeira captura como a chave; se o padrão não especifica nenhuma captura, então o casamento inteiro é usado como a chave.

Se `repl` é uma função, então esta função é chamada toda vez que o casamento ocorre, com todas as subcadeias capturadas sendo passadas como argumentos, na ordem em que foram capturadas; se o padrão não especifica nenhuma captura, então o casamento inteiro é passado como um único argumento.

Se o valor retornado pela consulta à tabela ou pela chamada de função é uma cadeia ou um número, então esse valor é usado como a cadeia de substituição; caso contrário, se ele é **false** ou **nil**, então não há substituição (isto é, o casamento original é mantido na cadeia).

Aqui estão alguns exemplos:

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

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

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

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

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

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

string.len (s)

Recebe uma cadeia e retorna seu comprimento. A cadeia vazia `""` tem comprimento 0. Zeros dentro da cadeia são contados, então `"a\000bc\000"` possui comprimento 5.

string.lower (s)

Recebe uma cadeia e retorna uma cópia desta cadeia com todas as letras maiúsculas convertidas para minúsculas. Todos os demais caracteres permanecem iguais. A definição de o que é uma letra maiúscula depende do idioma (*locale*) corrente.

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

Procura o primeiro *casamento* de `pattern` na cadeia `s`. Se encontra um, então `match` retorna as capturas do padrão; caso contrário retorna **nil**. Se `pattern` não especifica nenhuma captura, então o casamento inteiro é retornado. Um terceiro argumento numérico opcional, `init`, especifica onde iniciar a busca; seu valor padrão é 1 e pode ser negativo.

string.rep (s, n)

Retorna uma cadeia que é a concatenação de `n` cópias da cadeia `s`.

string.reverse (s)

Retorna uma cadeia que é a cadeia `s` invertida.

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

Retorna uma subcadeia de `s` que inicia em `i` e continua até `j`; `i` e `j` podem ser negativos. Se `j` está ausente, então assume-se que ele é igual a -1 (que é o mesmo que o comprimento da cadeia). Em particular, a chamada `string.sub(s,1,j)` retorna um prefixo de `s` com comprimento `j` e `string.sub(s, -i)` retorna um sufixo de `s` com comprimento `i`.

string.upper (s)

Recebe uma cadeia e retorna uma cópia desta cadeia com todas as letras minúsculas convertidas para maiúsculas. Todos os demais caracteres permanecem iguais. A definição de o que é uma letra minúscula depende do idioma (*locale*) corrente.

5.4.1 - Padrões

Classes de Caracteres:

Uma *classe de caracteres* é usada para representar um conjunto de caracteres. As seguintes combinações são permitidas em descrições de uma classe de caracteres:

- ***x*:** (onde *x* não é um dos *caracteres mágicos* `^$()%.[]*+-?`) representa o próprio caractere *x*.
- **`.`:** (um ponto) representa todos os caracteres.
- **`%a`:** representa todas as letras.
- **`%c`:** representa todos os caracteres de controle.
- **`%d`:** representa todos os dígitos.
- **`%l`:** representa todas as letras minúsculas.
- **`%p`:** representa todos os caracteres de pontuação.
- **`%s`:** representa todos os caracteres de espaço.
- **`%u`:** representa todas as letras maiúsculas.

- **`%w`:** representa todos os caracteres alfanuméricos.
- **`%x`:** representa todos os dígitos hexadecimais.
- **`%z`:** representa o caractere com representação 0.
- **`%x`:** (onde *x* é qualquer caractere não-alfanumérico) representa o caractere *x*. Esta é a maneira padrão de escapar os caracteres mágicos. Qualquer caractere de pontuação (até mesmo os não mágicos) pode ser precedido por um '`%`' quando usado para representar a si mesmo em um padrão.
- **`[`*`set`*`]`:** representa a classe que é a união de todos os caracteres em *set*. Uma faixa de caracteres pode ser especificada separando os caracteres finais da faixa com um '`-`'. Todas as classes `%`*x* descritas acima também podem ser usadas como componentes em *set*. Todos os outros caracteres em *set* representam eles mesmos. Por exemplo, `[%w_]` (ou `[_%w]`) representa todos os caracteres alfanuméricos mais o sublinhado, `[0-7]` representa os dígitos octais e `[0-7%l%-]` representa os dígitos octais mais as letras minúsculas mais o caractere '`-`'.

  A interação entre faixas e classes não é definida. Portanto, padrões como `[%a-z]` ou `[a-%%]` não possuem significado.

- **`[^`*`set`*`]`:** representa o complemento de *set*, onde *set* é interpretado como acima.

Para todas as classes representadas por uma única letra (`%a`, `%c`, etc.), a letra maiúscula correspondente representa o complemento da classe. Por exemplo, `%S` representa todos os caracteres que não são de espaço.

As definições de letra, espaço e outros grupos de caracteres dependem do idioma (*locale*) corrente. Em particular, a classe `[a-z]` pode não ser equivalente a `%l`.

Item de Padrão:

Um *item de padrão* pode ser

- uma classe de um único caractere, que casa qualquer caractere simples que pentença à classe;
- uma classe de um único caractere seguida por '`*`', que casa 0 ou mais repetições de caracteres da classe. Estes itens de repetição sempre casarão a maior seqüência possível;
- uma classe de um único caractere seguida por '`+`', que casa 1 ou mais repetições de caracteres da classe. Estes itens de repetição sempre casarão a maior seqüência possível;
- uma classe de um único caractere seguida por '`-`', que também casa 0 ou mais repetições de caracteres da classe. Diferentemente de '`*`', estes itens de repetição sempre casarão a *menor* seqüência possível;
- uma classe de um único caractere seguida por '`?`', que casa 0 ou 1 ocorrência de um caractere da classe;
- `%`*n*, para *n* entre 1 e 9; tal item casa uma subcadeia igual à *n*-ésima cadeia capturada (veja abaixo);
- `%b`*xy*, onde *x* e *y* são dois caracteres distintos; tal item casa cadeias que começam com *x*, terminam com *y* e onde o número de *x*s e de *y*s é *balanceado*. Isto significa que, se alguém ler a cadeia da esquerda para a direita, contando *+1* para um *x* e *-1* para um *y*, o *y* final é o primeiro *y* onde o contador alcança 0. Por exemplo, o item `%b()` casa expressões com parênteses balanceados.

Padrão:

Um *padrão* é uma seqüência de itens de padrão. Um '`^`' no início de um padrão ancora o casamento no início da cadeia sendo usada. Um '`$`' no fim de um padrão ancora o casamento no fim da cadeia sendo usada. Em outras posições, '`^`' e '`$`' não possuem significado especial e representam a si mesmos.

Capturas:

Um padrão pode conter subpadrões delimitados por parênteses; eles descrevem *capturas*. Quando um casamento ocorre, as subcadeias da cadeia sendo usada que casaram com as capturas são armazenadas (*capturadas*) para uso futuro. Capturas são numeradas de acordo com os seus parênteses esquerdos. Por exemplo, no padrão `"(a*(.)%w(%s*))"`, a parte da cadeia casando `"a*(.)%w(%s*)"` é armazenada como a primeira captura (e portanto tem número 1); o caractere casando "`.`" é capturado com o número 2 e a parte casando "`%s*`" possui número 3.

Como um caso especial, a captura vazia `()` captura a posição da cadeia corrente (um número). Por exemplo, se aplicarmos o padrão `"()aa()"` na cadeia `"flaaap"`, haverá duas capturas: 3 e 5.

Um padrão não pode conter zeros dentro dele. Use `%z` como alternativa.

5.5 - Manipulação de Tabelas

Esta biblioteca provê funções genéricas para manipulação de tabelas. Ela provê todas as suas funções na tabela `table`.

A maioria das funções na biblioteca de tabelas assume que a tabela representa um array ou uma lista. Para estas funções, quando falamos sobre o "comprimento" de uma tabela estamos falando sobre o resultado do operador de comprimento.

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

Dado um array onde todos os elementos são cadeias ou números, retorna `table[i]..sep..table[i+1] ··· sep..table[j]`. O valor padrão para `sep` é a cadeia vazia, o padrão para `i` é 1 e o padrão para `j` é o comprimento da tabela. Se `i` é maior do que `j`, retorna a cadeia vazia.

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

Insere o elemento `value` na posição `pos` de `table`, deslocando os outros elementos para abrir espaço, se necessário. O valor padrão para `pos` é `n+1`, onde `n` é o comprimento da tabela (ver §2.5.5), de modo que uma chamada `table.insert(t,x)` insere `x` no fim da tabela `t`.

table.maxn (table)

Retorna o maior índice numérico positivo da tabela fornecida ou zero se a tabela não possui índices numéricos positivos. (Para realizar seu trabalho esta função faz um percorrimento linear da tabela inteira.)

table.remove (table [, pos])

Remove de `table` o elemento na posição `pos`, deslocando os outros elementos para preencher o espaço, se necessário. Retorna o valor do elemento removido. O valor padrão para `pos` é `n`, onde `n` é o comprimento da tabela, de modo que uma chamada `table.remove(t)` remove o último elemento da tabela `t`.

table.sort (table [, comp])

Ordena os elementos da tabela em uma dada ordem, *in-place*, de `table[1]` até `table[n]`, onde `n` é o comprimento da tabela. Se `comp` é fornecido, então ele deve ser uma função que recebe dois elementos da tabela e retorna true quando o primeiro é menor do que o segundo (de modo que `not comp(a[i+1],a[i])` será verdadeiro após a ordenação). Se `comp` não é fornecido, então o operador padrão de Lua < é usado em seu lugar.

O algoritmo de ordenação não é estável; isto é, elementos considerados iguais pela ordem fornecida podem ter suas posições relativas trocadas pela ordenação.

5.6 - Funções Matemáticas

Esta biblioteca é uma interface para a biblioteca matemática de C padrão. Ela provê todas as suas funções na tabela `math`.

math.abs (x)

Retorna o valor absoluto de `x`.

math.acos (x)

Retorna o arco co-seno de `x` (em radianos).

math.asin (x)

Retorna o arco seno de `x` (em radianos).

math.atan (x)

Retorna o arco tangente de `x` (em radianos).

math.atan2 (y, x)

Retorna o arco tangente de `y/x` (em radianos), mas usa o sinal dos dois parâmetros para achar o quadrante do resultado. (Também trata corretamente o caso de `x` ser zero.)

math.ceil (x)

Retorna o menor inteiro maior ou igual a `x`.

math.cos (x)

Retorna o co-seno de `x` (assume que `x` está em radianos).

math.cosh (x)

Retorna o co-seno hiperbólico de `x`.

math.deg (x)

Retorna o ângulo `x` (dado em radianos) em graus.

math.exp (x)

Retorna o valor de e^x.

math.floor (x)

Retorna o maior inteiro menor ou igual a `x`.

math.fmod (x, y)

Retorna o resto da divisão de `x` por `y` que arredonda o quociente em direção a zero.

math.frexp (x)

Retorna `m` e `e` tais que $x = m2^e$, `e` é um inteiro e o valor absoluto de `m` está no intervalo *[0.5, 1)* (ou zero quando `x` é zero).

math.huge

O valor de `HUGE_VAL`, um valor maior ou igual a qualquer outro valor numérico.

math.ldexp (m, e)

Retorna $m2^e$ (`e` deve ser um inteiro).

math.log (x)

Retorna o logaritmo natural de `x`.

math.log10 (x)

Retorna o logaritmo base-10 de `x`.

math.max (x, ···)

Retorna o valor máximo entre os seus argumentos.

math.min (x, ···)

Retorna o valor mínimo entre os seus argumentos.

math.modf (x)

Retorna dois números, a parte integral de `x` e a parte fracionária de `x`.

math.pi

O valor de *pi*.

math.pow (x, y)

Retorna x^y. (Você também pode usar a expressão `x^y` para computar este valor.)

math.rad (x)

Retorna o ângulo `x` (dado em graus) em radianos.

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

Esta função é uma interface para a função geradora pseudo-randômica simples `rand` fornecida por ANSI C. (Nenhuma garantia pode ser dada para suas propriedades estatísticas.)

Quando chamada sem argumentos, retorna um número real pseudo-randômico uniforme no intervalo *[0,1)*. Quando chamada com um número inteiro `m`, `math.random` retorna um inteiro pseudo-randômico uniforme no intervalo *[1, m]*. Quando chamada com dois números inteiros `m` e `n`, `math.random` retorna um inteiro pseudo-randômico uniforme no intervalo *[m, n]*.

math.randomseed (x)

Estabelece `x` como a "semente" para o gerador pseudo-randômico: sementes iguais produzem seqüências iguais de números.

math.sin (x)

Retorna o seno de `x` (assume que `x` está em radianos).

math.sinh (x)

Retorna o seno hiperbólico de `x`.

math.sqrt (x)

Retorna a raiz quadrada de `x`. (Você também pode usar a expressão `x^0.5` para computar este valor.)

math.tan (x)

Retorna a tangente de `x` (assume que `x` está em radianos).

math.tanh (x)

Retorna a tangente hiperbólica de `x`.

5.7 - Facilidades de Entrada e Saída

A biblioteca de E/S provê dois estilos diferentes para manipulação de arquivos. O primeiro usa descritores de arquivo implícitos; isto é, há operações para estabelecer um arquivo de entrada padrão e um arquivo de saída padrão e todas as operações de entrada/saída são realizadas sobre estes arquivos. O segundo estilo usa descritores de arquivo explícitos.

Quando se usa descritores de arquivo implícitos, todas as operações são providas pela tabela `io`. Quando se usa descritores de arquivo explícitos, a operação `io.open` retorna um descritor de arquivo e então todas as operações são providas como métodos do descritor de arquivo.

A tabela `io` também fornece três descritores de arquivo pré-definidos com os seus significados usuais de C: `io.stdin`, `io.stdout` e `io.stderr`. A biblioteca de E/S nunca fecha estes arquivos.

A menos que dito de modo contrário, todas as funções de E/S retornam **nil** em caso de falha (mais uma mensagem de erro como segundo resultado e um código de erro dependente do sistema como um terceiro resultado), ou algum valor diferente de **nil** em caso de sucesso.

io.close ([file])

Equivalente a `file:close()`. Quando não recebe `file`, fecha o arquivo de saída padrão.

io.flush ()

Equivalente a `file:flush` no arquivo de saída padrão.

io.input ([file])

Quando chamada com um nome de arquivo, abre o arquivo com aquele nome (em modo texto) e estabelece seu manipulador como o arquivo de entrada padrão. Quando chamada com um manipulador de arquivo, simplesmente estabelece este manipulador de arquivo como o arquivo de entrada padrão. Quando chamada sem parâmetros, retorna o arquivo de entrada padrão corrente.

Em caso de erros esta função dispara o erro, ao invés de retornar um código de erro.

io.lines ([filename])

Abre o nome de arquivo fornecido em modo de leitura e retorna uma função iteradora que, cada vez que é chamada, retorna uma nova linha do arquivo. Portanto, a construção

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

irá iterar sobre todas as linhas do arquivo. Quando a função iteradora detecta o fim do arquivo, ela retorna **nil** (para finalizar o laço) e automaticamente fecha o arquivo.

A chamada `io.lines()` (sem nenhum nome de arquivo) é equivalente a `io.input():lines()`; isto é, ela itera sobre as linhas do arquivo de entrada padrão. Neste caso ela não fecha o arquivo quando o laço termina.

io.open (filename [, mode])

Esta função abre um arquivo, no modo especificado na cadeia `mode`. Ela retorna um novo manipulador de arquivo ou, em caso de erros, **nil** mais uma mensagem de erro.

A cadeia de caracteres `mode` pode ser qualquer uma das seguintes:

- **"r":** modo de leitura (o padrão);
- **"w":** modo de escrita;
- **"a":** modo de adição;
- **"r+":** modo de atualização, todos os dados anteriores são preservados;
- **"w+":** modo de atualização, todos os dados anteriores sao apagados;
- **"a+":** modo de atualização de adição, dados anteriores são preservados, a escrita somente é permitida no fim do arquivo.

A cadeia `mode` também pode ter um 'b' no fim, que é necessário em alguns sistemas para abrir o arquivo em modo binário. Esta cadeia é exatamente o que é usado na função padrão de C `fopen`.

io.output ([file])

Similar a `io.input`, mas opera sobre o arquivo de saída padrão.

io.popen (prog [, mode])

Inicia o programa `prog` em um processo separado e retorna um manipulador de arquivo que pode ser usado para ler dados deste programa (se `mode` é `"r"`, o padrão) ou escrever dados para este programa (se `mode` é `"w"`).

Esta função é dependente do sistema e não está disponível em todas as plataformas.

io.read (···)

Equivalente a `io.input():read`.

io.tmpfile ()

Retorna um manipulador para um arquivo temporário. Este arquivo é aberto em modo de atualização e é automaticamente removido quando o programa termina.

io.type (obj)

Verifica se `obj` é um manipulador de arquivo válido. Retorna a cadeia `"file"` se `obj` é um manipulador de arquivo aberto, `"close file"` se `obj` é um manipulador de arquivo fechado ou **nil** se `obj` não é um manipulador de arquivo.

io.write (···)

Equivalente a `io.output():write`.

file:close ()

Fecha `file`. Note que arquivos são automaticamente fechados quando seus manipuladores são coletados pelo coletor de lixo, mas leva uma quantidade indeterminada de tempo para isso acontecer.

file:flush ()

Salva qualquer dado escrito para `file`.

file:lines ()

Retorna uma função iteradora que, cada vez que é chamada, retorna uma nova linha do arquivo. Portanto, a construção

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

irá iterar sobre todas as linhas do arquivo. (Ao contrário de `io.lines`, essa função não fecha o arquivo quando o laço termina.)

file:read (···)

Lê o arquivo `file`, de acordo com os formatos fornecidos, os quais especificam o que deve ser lido. Para cada formato, a função retorna uma cadeia (ou um número) com os caracteres lidos ou **nil** se ela não pode retornar dados com o formato especificado. Quando chamada sem formatos, ela usa o formato padrão que lê a próxima linha toda (veja abaixo).

Os formatos disponíveis são

- **"*n":** lê um número; este é o único formato que retorna um número ao invés de uma cadeia.
- **"*a":** lê o arquivo inteiro, iniciando na posição corrente. Quando está no final do arquivo, retorna a cadeia vazia.
- **"*l":** lê a próxima linha (pulando o fim de linha), retornando **nil** ao final do arquivo. Este é o formato padrão.
- ***number*:** lê uma cadeia até este número de caracteres, retornando **nil** ao final do arquivo. Se o número fornecido é zero, a função não lê nada e retorna uma cadeia vazia ou **nil** quando está no fim do arquivo.

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

Estabelece e obtém a posição do arquivo, medida a partir do início do arquivo, até a posição dada por `offset` mais uma base especificada pela cadeia `whence`, como a seguir:

- **"set":** base é a posição 0 (o início do arquivo);

- **"cur":** base é a posição corrente;
- **"end":** base é o fim do arquivo;

Em caso de sucesso, a função `seek` retorna a posição final do arquivo, medida em bytes a partir do início do arquivo. Se esta função falha, ela retorna **nil**, mais uma cadeia descrevendo o erro.

O valor padrão para `whence` é `"cur"` e para `offset` é 0. Portanto, a chamada `file:seek()` retorna a posição do arquivo corrente, sem modificá-la; a chamada `file:seek("set")` estabelece a posição para o início do arquivo (e retorna 0); e a chamada `file:seek("end")` estabelece a posição para o fim do arquivo e retorna seu tamanho.

file:setvbuf (mode [, size])

Define o modo de bufferização para um arquivo de saída. Há três modos disponíveis:

- **"no":** nenhuma bufferização; o resultado de qualquer operação de saída aparece imediatamente.
- **"full":** bufferização completa; a operação de saída é realizada somente quando o buffer está cheio (ou quando você explicitamente `descarrega` o arquivo (ver `io.flush`)).
- **"line":** bufferização de linha; a saída é bufferizada até que uma nova linha é produzida ou há qualquer entrada a partir de alguns arquivos especiais (como um dispositivo de terminal).

Para os últimos dois casos, `size` especifica o tamanho do buffer, em bytes. O padrão é um tamanho apropriado.

file:write (···)

Escreve o valor de cada um de seus argumentos para `file`. Os argumentos devem ser cadeias de caracteres ou números. Para escrever outros valores, use `tostring` ou `string.format` antes de `write`.

5.8 - Facilidades do Sistema Operacional

Esta biblioteca é implementada através da tabela `os`.

os.clock ()

Retorna uma aproximação da quantidade de tempo de CPU, em segundos, usada pelo programa.

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

Retorna uma cadeia ou uma tabela contendo data e hora, formatada de acordo com a cadeia `format` fornecida.

Se o argumento `time` está presente, este é o tempo a ser formatado (veja a função `os.time` para uma descrição deste valor). Caso contrário, `date` formata a hora corrente.

Se `format` começa com `'!'`, então a data é formatada no Tempo Universal Coordenado. Após esse caractere opcional, se `format` é a cadeia `"*t"`, então `date` retorna uma tabela com os seguintes campos: `year` (quatro dígitos), `month` (1--12), `day` (1--31), `hour` (0--23), `min` (0--59), `sec` (0--61), `wday` (dia da semana, domingo é 1), `yday` (dia do ano) e `isdst` (flag que indica o horário de verão, um booleano).

Se `format` não é `"*t"`, então `date` retorna a data como uma cadeia de caracteres, formatada de acordo com as mesmas regras da função C `strftime`.

Quando chamada sem argumentos, `date` retorna uma representação aceitável da data e da hora que depende do sistema hospedeiro e do idioma (*locale*) corrente. (isto é, `os.date()` é equivalente a `os.date("%c")`).

os.difftime (t2, t1)

Retorna o número de segundos a partir do tempo `t1` até o tempo `t2`. Em POSIX, Windows e alguns outros sistemas, este valor é exatamente `t2-t1`.

os.execute ([command])

Esta função é equivalente à função C `system`. Ela passa `command` para ser executado por um interpretador de comandos do sistema operacional. Ela retorna um código de status, que é dependente do sistema. Se `command` está ausente, então a função retorna um valor diferente de zero se um interpretrador de comandos está disponível e zero caso contrário.

os.exit ([code])

Chama a função C `exit`, com um código `code` opcional, para terminar o programa hospedeiro. O valor padrão para `code` é o código de sucesso.

os.getenv (varname)

Retorna o valor da variável de ambiente do processo `varname` ou **nil** se a variável não está definida.

os.remove (filename)

Remove um arquivo ou diretório com o nome fornecido. Diretórios devem estar vazios para serem removidos. Se esta função falha, ela retorna **nil**, mais uma cadeia descrevendo o erro.

os.rename (oldname, newname)

Renomeia um arquivo ou diretório chamado `oldname` para `newname`. Se esta função falha, ela retorna **nil**, mais uma cadeia descrevendo o erro.

os.setlocale (locale [, category])

Estabelece o idioma (*locale*) corrente do programa. `locale` é uma cadeia de caracteres especificando um idioma; `category` é uma cadeia opcional descrevendo para qual categoria deve-se mudar: `"all"`, `"collate"`, `"ctype"`, `"monetary"`, `"numeric"` ou `"time"`; a categoria padrão é `"all"`. Esta função retorna o nome do novo idioma ou **nil** se a requisição não pode ser honrada.

Se `locale` é a cadeia vazia, estabelece-se o idioma corrente como um idioma nativo definido pela implementação. Se `locale` é a cadeia "C", estabelece-se o idioma corrente como o idioma padrão de C.

Quando chamada com **nil** como o primeiro argumento, esta função retorna somente o nome do idioma corrente para a categoria fornecida.

os.time ([table])

Retorna o tempo corrente quando chamada sem argumentos ou um tempo representando a data e a hora especificados pela tabela fornecida. Esta tabela deve ter campos `year`, `month` e `day` e pode ter campos `hour`, `min`, `sec` e `isdst` (para uma descrição destes campos, veja a função `os.date`).

O valor retornado é um número, cujo significado depende do seu sistema. Em POSIX, Windows e alguns outros sistemas, este número conta o número de segundos desde algum tempo de início dado (a "era"). Em outros sistemas, o significado não é especificado e o número retornado por `time` pode ser usado somente como um argumento para `date` e `difftime`.

os.tmpname ()

Retorna uma cadeia de caracteres com o nome de um arquivo que pode ser usado para um arquivo temporário. O arquivo deve ser explicitamente aberto antes de ser usado e explicitamente removido quando não for mais necessário.

Em alguns sistemas (POSIX), esta função também cria um arquivo com esse nome, para evitar riscos de segurança. (Outro usuário pode criar o arquivo com permissões erradas no intervalo de tempo entre a obtenção do nome e a criação do arquivo.) Você ainda deve abrir o arquivo para poder usá-lo e deve removê-lo (mesmo que você não tenha usado).

Quando possível, você pode preferir usar `io.tmpfile`, que automaticamente remove o arquivo quando o programa termina.

5.9 - A Biblioteca de Depuração

Esta biblioteca provê as funcionalidades da interface de depuração para programas Lua. Você deve ter cuidado ao usar esta biblioteca. As funções fornecidas aqui devem ser usadas exclusivamente para depuração e tarefas similares, tais como medição (*profiling*). Por favor resista à tentação de usá-las como uma ferramenta de programação usual: elas podem ser muito lentas. Além disso, várias dessas funções violam algumas suposições a respeito do código Lua (e.g., que variáveis locais a uma função não podem ser acessadas de fora da função ou que metatabelas de objetos userdata não podem ser modificadas por código Lua) e portanto podem comprometer código que, de outro modo, seria seguro.

Todas as funções nesta biblioteca são fornecidas na tabela `debug`. Todas as funções que operam sobre um objeto do tipo thread possuem um primeiro argumento opcional que é o objeto thread sobre o qual a função deve operar. O padrão é sempre o fluxo de execução corrente.

debug.debug ()

Entra em um modo interativo com o usuário, executando cada cadeia de caracteres que o usuário entra. Usando comandos simples e outros mecanismos de depuração, o usuário pode inspecionar variáveis globais e locais, mudar o valor delas, avaliar expressões, etc. Uma linha contendo somente a palavra `cont` termina esta função, de modo que a função chamadora continua sua execução.

Note que os comandos para `debug.debug` não são aninhados de modo léxico dentro de nenhuma função e portanto não possuem acesso direto a variáveis locais.

debug.getfenv (o)

Retorna o ambiente do objeto `o`.

debug.gethook ([thread])

Retorna as configurações de gancho correntes do fluxo de execução como três valores: a função de gancho corrente, a máscara de ganho corrente e a contagem de ganho corrente (como estabelecido pela função `debug.sethook`).

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

Retorna uma tabela com informação sobre uma função. Você pode fornecer a função diretamente ou você pode fornecer um número como o valor de `function`, que significa a função executando no nível `function` da pilha de chamadas do fluxo de execução fornecido: nível 0 é a função corrente (a própria `getinfo`); nível 1 é a função que chamou `getinfo`; e assim por diante. Se `function` é um número maior do que o número de funções ativas, então `getinfo` retorna **nil**.

A tabela retornada pode conter todos os campos retornados por `lua_getinfo`, com a cadeia `what` descrevendo quais campos devem ser preenchidos. O padrão para `what` é obter todas as informações disponíveis, exceto a tabela de linhas válidas. Se presente, a opção '`f`' adiciona um campo chamado `func` com a própria função. Se presente, a opção '`L`' adiciona um campo chamado `activelines` com a tabela de linhas válidas.

Por exemplo, a expressão `debug.getinfo(1,"n").name` retorna uma tabela com um nome para a função corrente, se um nome razoável pode ser encontrado, e a expressão `debug.getinfo(print)` retorna uma tabela com todas as informações disponíveis sobre a função `print`.

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

Esta função retorna o nome e o valor da variável local com índice `local` da função no nível `level` da pilha. (O primeiro parâmetro ou variável local possui índice 1 e assim por diante, até a última variável local ativa.) A função retorna **nil** se não existe uma variável local com o índice fornecido e dispara um erro quando chamada com um `level` fora da faixa de valores válidos. (Você pode chamar `debug.getinfo` para verificar se o nível é válido.)

Nomes de variáveis que começam com '`(`' (abre parênteses) representam variáveis internas (variáveis de controle de laços, temporários e locais de funções C).

debug.getmetatable (object)

Retorna a metatabela do `object` fornecido ou **nil** se ele não possui uma metatabela.

debug.getregistry ()

Retorna a tabela de registro (ver §3.5).

debug.getupvalue (func, up)

Esta função retorna o nome e o valor do upvalue com índice `up` da função `func`. A função retorna **nil** se não há um upvalue com o índice fornecido.

debug.setfenv (object, table)

Estabelece a tabela `table` como o ambiente do `object` fornecido. Retorna `object`.

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

Estabelece a função fornecida como um gancho. A cadeia `mask` e o número `count` descrevem quando o gancho será chamado. A cadeia `mask` pode ter os seguintes caracteres, com o respectivo significado:

- **`"c"`:** o gancho é chamado toda vez que Lua chama uma função;
- **`"r"`:** o gancho é chamado toda vez que Lua retorna de uma função;
- **`"l"`:** o gancho é chamado toda vez que Lua entra uma nova linha de código.

Com um `count` diferente de zero, o gancho é chamado após cada `count` instruções.

Quando chamada sem argumentos, `debug.sethook` desabilita o gancho.

Quando o gancho é chamado, seu primeiro parâmetro é uma cadeia de caracteres descrevendo o evento que disparou a sua chamada: `"call"`, `"return"` (ou `"tail return"`, quando estiver simulando um retorno de uma recursão final), `"line"` e `"count"`. Para eventos de linha, o gancho também obtém o novo número de linha como seu segundo parâmetro. Dentro do gancho, é possível chamar `getinfo` com nível 2 para obter mais informação sobre a função sendo executada (nível 0 é a função `getinfo` e nível 1 é a função de gancho), a menos que o evento seja `"tail return"`. Neste caso, Lua está somente simulando o retorno e uma chamada a `getinfo` retornará dados inválidos.

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

Esta função atribui o valor `value` à variável local com índice `local` da função no nível `level` da pilha. A função retorna **nil** se não há uma variável local com o índice fornecido e dispara um erro quando chamada com um `level` fora da faixa de valores válidos. (Você pode chamar `getinfo` para verificar se o nível é válido.) Caso contrário, a função retorna o nome da variável local.

debug.setmetatable (object, table)

Estabelece `table` como a metatabela do `object` fornecido (`table` pode ser **nil**).

debug.setupvalue (func, up, value)

Esta função atribui o valor `value` ao upvalue com índice `up` da função `func`. A função retorna **nil** se não há um upvalue com o índice fornecido. Caso contrário, a função retorna o nome do upvalue.

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

Retorna uma cadeia de caracteres com um traço da pilha de chamadas. Uma cadeia opcional `message` é adicionada ao início do traço. Um número opcional `level` diz em qual nível iniciar o traço (o padrão é 1, a função chamando `traceback`).

6 - O Interpretador de Linha de Comando Lua

Embora Lua tenha sido projetada como uma linguagem de extensão, para ser embutida em um programa C hospedeiro, Lua também é freqüentemente usada como uma linguagem auto-suficiente. Um interpretador para Lua como uma linguagem auto-suficiente, chamado simplesmente `lua`, é fornecido com a distribuição padrão. Esse interpretador inclui todas as bibliotecas padrão, inclusive a biblioteca de depuração. Seu uso é:

```
lua [options] [script [args]]
```

As opções são:

- **`-e stat`:** executa a cadeia *stat*;
- **`-l mod`:** "requisita" *mod*;
- **`-i`:** entra em modo interativo após executar *script*;
- **`-v`:** imprime informação de versão;
- **`--`:** pára de tratar opções;
- **`-`:** executa `stdin` como um arquivo e pára de tratar opções.

Após tratar suas opções, `lua` executa o *script* fornecido, passando para ele os *args* fornecidos como cadeias de argumentos. Quando chamado sem argumentos, `lua` comporta-se como `lua -v -i` quando a entrada padrão (`stdin`) é um terminal e como `lua -` em caso contrário.

Antes de executar qualquer argumento, o interpretador verifica se há uma variável de ambiente `LUA_INIT`. Se seu formato é `@`*`filename`*, então `lua` executa o arquivo. Caso contrário, `lua` executa a própria cadeia de caracteres.

Todas as opções são manipuladas na ordem dada, exceto `-i`. Por exemplo, uma invocação como

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

irá primeiro atribuir 1 a `a`, depois imprimirá o valor de `a` (que é '`1`') e finalmente executará o arquivo `script.lua` sem argumentos. (Aqui `$` é o *prompt* do interpretador de comandos. Você pode ter um prompt diferente.)

Antes de começar a executar o script, `lua` guarda todos os argumentos fornecidos na linha de comando em uma tabela global chamada `arg`. O nome do script é armazenado no índice 0, o primeiro argumento após o nome do script fica no índice 1 e assim por diante. Quaisquer argumentos antes do nome do script (isto é, o nome do interpretador mais as opções) ficam em índices negativos. Por exemplo, na chamada

```
$ lua -la b.lua t1 t2
```

o interpretador primeiro executa o arquivo `a.lua`, depois cria a tabela

```
arg = { [-2] = "lua", [-1] = "-la",
        [0] = "b.lua",
        [1] = "t1", [2] = "t2" }
```

e finalmente executa o arquivo `b.lua`. O script é chamado com `arg[1]`, `arg[2]`, ··· como argumentos; ele também pode acessar estes argumentos com a expressão vararg '`...`'.

Em modo interativo, se você escrever um comando incompleto, o interpretador espera que você o complete e indica isto através de um prompt diferente.

Se a variável global `_PROMPT` contém uma cadeia de caracteres, então o seu valor é usado como o prompt. De maneira similar, se a variável global `_PROMPT2` contém uma cadeia, seu valor é usado como o prompt secundário (mostrado durante comandos incompletos). Portanto, os dois prompts podem ser modificados diretamente na linha de comando ou em quaisquer programas Lua fazendo uma atribuição a `_PROMPT`. Veja o exemplo a seguir:

```
$ lua -e"_PROMPT='myprompt> '" -i
```

(O par de aspas mais externo é para o interpretador de comandos e o par mais interno é para Lua.) Note o uso de `-i` para entrar em modo interativo; caso contrário, o programa iria terminar silenciosamente logo após a atribuição a `_PROMPT`.

Para permitir o uso de Lua como um interpretador de scripts em sistemas Unix, o interpretador de linha de comando pula a primeira linha de um trecho de código se ele começa com `#`. Portanto, scripts Lua podem ser usados como programas executáveis usando `chmod +x` e a forma `#!`, como em

```
#!/usr/local/bin/lua
```

(É claro que a localização do interpretador Lua pode ser diferente na sua máquina. Se `lua` está em seu `PATH`, então

```
#!/usr/bin/env lua
```

é uma solução mais portável.)

7 - Incompatibilidades com a Versão Anterior

Listamos aqui as incompatibilidades que você pode encontrar quando passando um programa de Lua 5.0 para Lua 5.1. Você pode evitar a maioria das incompatibilidades compilando Lua com opções apropriadas (veja o arquivo `luaconf.h`). Contudo, todas essas opções de compatibilidade serão removidas na próxima versão de Lua.

7.1 - Mudanças na Linguagem

- O sistema de vararg mudou do pseudo-argumento `arg` com uma tabela com os argumentos extras para a expressão vararg. (Veja a opção de tempo de compilação `LUA_COMPAT_VARARG` em `luaconf.h`.)
- Houve uma mudança sutil no escopo das variáveis implícitas do comando **for** e do comando **repeat**.
- A sintaxe de cadeia longa/comentário longo (`[[`*`string`*`]]`) não permite aninhamento. Você pode usar a nova sintaxe (`[=[`*`string`*`]=]`) nesses casos. (Veja a opção de tempo de compilação `LUA_COMPAT_LSTR` em `luaconf.h`.)

7.2 - Mudanças nas Bibliotecas

- A função `string.gfind` foi renomeada para `string.gmatch`. (Veja a opção de tempo de compilação `LUA_COMPAT_GFIND` em `luaconf.h`.)
- Quando `string.gsub` é chamada com uma função como seu terceiro argumento, sempre que esta função retorna **nil** ou **false** a cadeia de substituição é o casamento inteiro, ao invés da cadeia vazia.
- A função `table.setn` está ultrapassada e não deve ser usada. A função `table.getn` corresponde ao novo operador de comprimento (#); use o operador ao invés da função. (Veja a opção de tempo de compilação `LUA_COMPAT_GETN` em `luaconf.h`.)
- A função `loadlib` foi renomeada para `package.loadlib`. (Veja a opção de tempo de compilação `LUA_COMPAT_LOADLIB` em `luaconf.h`.)
- A função `math.mod` foi renomeada para `math.fmod`. (Veja a opção de tempo de compilação `LUA_COMPAT_MOD` em `luaconf.h`.)
- As funções `table.foreach` e `table.foreachi` estão ultrapassadas e não devem ser usadas. Você pode usar um laço **for** com `pairs` ou `ipairs` ao invés delas.
- Houve mudanças substanciais na função `require` devido ao novo sistema de módulos. O novo comportamento é basicamente compatível com o antigo, porém agora `require` obtém o caminho de `package.path` e não mais de `LUA_PATH`.
- A função `collectgarbage` possui argumentos diferentes. A função `gcinfo` está ultrapassada e não deve ser usada; use `collectgarbage("count")` ao invés dela.

7.3 - Mudanças na API

- As funções `luaopen_*` (para abrir bibliotecas) não podem ser chamadas diretamente, como uma função C comum. Elas devem ser chamadas através de Lua, como uma função Lua.
- A função `lua_open` foi substituída por `lua_newstate` para permitir que o usuário defina uma função de alocação de memória. Você pode usar `luaL_newstate` da biblioteca padrão para criar um estado com uma função de alocação padrão (baseada em `realloc`).
- As funções `luaL_getn` e `luaL_setn` (da biblioteca auxiliar) estão ultrapassadas e não devem ser usadas. Use `lua_objlen` ao invés de `luaL_getn` e nada no lugar de `luaL_setn`.
- A função `luaL_openlib` foi substituída por `luaL_register`.
- A função `luaL_checkudata` agora dispara um erro quando o valor fornecido não é um objeto userdata do tipo esperado. (Em Lua 5.0 ela retornava `NULL`.)

8 - A Sintaxe Completa de Lua

Aqui está a sintaxe completa de Lua na notação BNF estendida. (Ela não descreve as precedências dos operadores.)

```
trecho ::= {comando [`;´]} [ultimocomando [`;´]]

bloco ::= trecho

comando ::=  listavar `=´ listaexp |
         chamadadefuncao |
         do bloco end |
         while exp do bloco end |
         repeat bloco until exp |
         if exp then bloco {elseif exp then bloco} [else bloco] end |
         for Nome `=´ exp `,´ exp [`,´ exp] do bloco end |
         for listadenomes in listaexp do bloco end |
         function nomedafuncao corpodafuncao |
         local function Nome corpodafuncao |
         local listadenomes [`=´ listaexp]

ultimocomando ::= return [listaexp] | break

nomedafuncao ::= Nome {`.´ Nome} [`:´ Nome]

listavar ::= var {`,´ var}

var ::=  Nome | expprefixo `[´ exp `]´ | expprefixo `.´ Nome

listadenomes ::= Nome {`,´ Nome}

listaexp ::= {exp `,´} exp
```

```
exp ::=  nil | false | true | Numero | Cadeia | `...´ | funcao |
         expprefixo | construtortabela | exp opbin exp | opunaria exp

expprefixo ::= var | chamadadefuncao | `(´ exp `)´

chamadadefuncao ::=  expprefixo args | expprefixo `:´ Nome args

args ::=  `(´ [listaexp] `)´ | construtortabela | Cadeia

funcao ::= function corpodafuncao

corpodafuncao ::= `(´ [listapar] `)´ bloco end

listapar ::= listadenomes [`,´ `...´] | `...´

construtortabela ::= `{´ [listadecampos] `}´

listadecampos ::= campo {separadordecampos campo} [separadordecampos]

campo ::= `[´ exp `]´ `=´ exp | Nome `=´ exp | exp

separadordecampos ::= `,´ | `;´

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

opunaria ::= `-´ | not | `#´
```

Last update: Wed Sep 28 01:04:56 BRT 2011