Manual de Referência de Lua 5.2

por Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes

(traduzido por Sérgio Queiroz de Medeiros)

conteúdo · índice · english · português

1 – Introdução

Lua é uma linguagem de programação de extensão projetada para dar suporte à programação procedimental em geral com facilidades para a descrição de dados. Ela também oferece um bom suporte para programação orientada a objetos, programação funcional, e programação orientada a dados. Lua é planejada para ser usada como uma linguagem de script poderosa, leve, e embarcável por qualquer programa que necessite de uma. Lua é implementada como uma biblioteca, escrita em *C puro*, o subconjunto comum de C Padrão e C++.

Por ser uma linguagem de extensão, Lua não possui a noção de um programa "principal": ela somente funciona *embarcada* em um cliente hospedeiro, chamado de *programa embarcante* ou simplesmente de *hospedeiro*. O 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 por código Lua. Através do uso de funções C, Lua pode ser aumentada para lidar com uma variedade ampla de domínios diferentes, criando assim linguagens 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 Lua para oferecer um interpretador Lua de linha de comando completo, para uso interativo ou em lote.

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

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

2 – Conceitos Básicos

Esta seção descreve os conceitos básicos da linguagem.

2.1 – Valores e Tipos

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

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

Há 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 a torna verdadeira. *Number* representa tanto números inteiros como números reais (ponto flutuante de precisão dupla). *String* representa sequências imutáveis de bytes. Lua é 8 bits pura: cadeias podem conter qualquer valor de 8 bits, incluindo zeros ('`\0`') dentro delas.

Lua pode chamar (e manipular) funções escritas em Lua e funções escritas em C (veja §3.4.9).

O tipo *userdata* é oferecido para permitir que dados C arbitrários sejam guardados em variáveis Lua. Um valor userdata é um ponteiro para um bloco de memória bruta. Há dois tipos de userdata: userdata completo, onde o bloco de memória é gerenciado por Lua, e userdata leve, onde o bloco de memória é gerenciado pelo hospedeiro. Userdata não possui operações pré-definidas em Lua, exceto atribuição e teste de identidade. Através do uso de *metatabelas*, o programador pode definir operações para valores userdata completos (veja §2.4). Valores userdata não podem ser criados ou modificados em Lua, somente através da API C. Isso garante a integridade de dados que pertencem ao programa hospedeiro.

O tipo *thread* representa fluxos de execução independentes e é usado para implementar co-rotinas (veja §2.6). Não confunda fluxos de execução 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 com números, mas com qualquer valor Lua exceto **nil** e NaN (*Not a Number*, um valor numérico especial usado para representar resultados indefinidos ou não representáveis, tais como `0/0`). Tabelas podem ser *heterogêneas*; isto é, elas podem conter valores de todos os tipos (exceto **nil**). Qualquer chave com valor **nil** não é considerada parte da tabela. De modo recíproco, qualquer chave que não é parte da tabela possui um valor **nil** associado. Tabelas são o único mecanismo de estruturação de dados em Lua; elas podem ser usadas para representar arrays comuns, sequências, 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 essa representação fornecendo `a.nome` como açúcar sintático para `a["nome"]`. Há várias maneiras convenientes para criar tabelas em Lua (veja §3.4.8).

Usamos o termo *sequência* para denotar uma tabela onde o conjunto de todas as chaves numéricas positivas é igual a *{1..n}* para algum inteiro *n*, que é chamado o comprimento da sequência (veja §3.4.6).

Assim como os índices, os valores dos campos de uma tabela podem ser de qualquer tipo. Em particular, por causa que funções são valores de primeira classe, campos de tabela podem conter funções. Portanto, tabelas podem também conter *métodos* (veja §3.4.10).

A indexação de tabelas segue a definição de igualdade primitiva na linguagem. As expressões `a[i]` e `a[j]` denotam o mesmo elemento da tabela se e somente se `i` e `j` são iguais primitivos (isto é, iguais sem metamétodos).

Valores do tipo table, function, thread, e userdata (completo) são *objetos*: variáveis não *contêm* realmente esses valores, somente *referências* para eles. Atribuição, passagem de parâmetro, e retornos de função sempre manipulam referências para tais valores; essas operações não implicam em qualquer espécie de cópia.

A função da biblioteca `type` retorna uma cadeia descrevendo o tipo de um dado valor (veja §6.1).

2.2 – Ambientes e o Ambiente Global

Como será discutido em §3.2 e §3.3.3, qualquer referência a um nome global `var` é sintaticamente traduzido para `_ENV.var`. Além disso, todo trecho é compilado no escopo de uma variável local externa chamada `_ENV` (veja §3.3.2), então o próprio `_ENV` nunca é um nome global em um trecho.

Apesar da existência dessa variável `_ENV` externa e da tradução de nomes globais, `_ENV` é um nome completamente normal. Em particular, você pode definir novas variáveis e parâmetros com esse nome. Cada referência a um nome global usa a `_ENV` que é visível naquele ponto do programa, seguindo as regras de visibilidade usuais de Lua (veja §3.5).

Qualquer tabela usada como o valor de `_ENV` é chamada de um *ambiente*.

Lua mantém um ambiente distinto chamado de o *ambiente global*. Esse valor é mantido em um índice especial no registro C (veja §4.5). Em Lua, a variável `_G` é inicializada com esse mesmo valor.

Quando Lua compila um trecho, ela inicializa o valor de seu upvalue `_ENV` com o ambiente global (veja `load`). Assim, por padrão, variáveis globais em código Lua se referem a entradas no ambiente global. Além disso, todas as bibliotecas padrão são carregadas no ambiente global e várias funções são operadas nesse ambiente. Você pode usar `load` (ou `loadfile`) para carregar um trecho com um ambiente diferente. (Em C, você pode carregar o trecho e então mudar o valor de seu primeiro upvalue.)

Se você mudar o ambiente global no registro (através de código C ou da biblioteca de depuração), todos os trechos carregados após a mudança terão o novo ambiente. Trechos carregados anteriormente não são afetados, contudo, uma vez que cada um tem sua própria referência para o ambiente na sua variável `_ENV`. Além disso, a variável `_G` (que é guardada no ambiente global original) nunca é atualizada por Lua.

2.3 – Tratamento de Erros

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

Código Lua pode explicitamente gerar um erro através de um chamada à função `error`. Se você precisa capturar erros em Lua, você pode usar `pcall` ou `xpcall` para chamar uma dada função em *modo protegido*. Sempre que há um erro, um *objeto de erro* (também chamado de uma *mensagem de erro*) é propagado com informação a respeito do erro. Lua em si somente gera erros onde o objeto de erro é uma cadeia, mas programas podem gerar erros com qualquer valor para o objeto de erro.

Quando você usa `xpcall` ou `lua_pcall`, você pode fornecer um *tratador de mensagens* para ser chamado em caso de erros. Essa função é chamada com a mensagem de erro original e retorna uma nova mensagem de erro. Ela é chamada antes que o erro desenrole a pilha, de modo que ela pode colher mais informação sobre o erro, por exemplo através da inspeção da pilha e da criação de um traço (traceback) da pilha. Esse tratador de mensagens é ainda protegido por uma chamada protegida; assim, um erro dentro do tratador de mensagens chamará o tratador de mensagens novamente. Se esse laço continua, Lua o interrompe e retorna uma mensagem de erro apropriada.

2.4 – Metatabelas e Metamétodos

Todo valor em Lua pode ter uma *metatabela*. Essa *metatabela* é uma tabela Lua comum que define o comportamento do valor original sob certas operações especiais. Você pode mudar vários aspectos do comportamento de operações sobre um valor especificando campos específicos em sua metatabela. Por exemplo, quando um valor não numérico é o operando de uma adição, Lua verifica se há uma função no campo "`__add`" da metatabela do valor. Se ela acha uma, Lua chama essa função para realizar a adição.

As chaves em uma metatabela são derivadas a partir dos nomes dos *eventos*; os valores correspondentes são chamados de *metamétodos*. No exemplo anterior, o evento é "`add`" e o metamétodo é a função que realiza a adição.

Você pode consultar a metatabela de qualquer valor usando a função `getmetatable`.

Você pode substituir a metatabela de tabelas usando a função `setmetatable`. Você não pode mudar a metatabela de outros tipos a partir de Lua (exceto usando a biblioteca de depuração); você deve usar a API C para isso.

Tabelas e userdatas completos têm metatabelas individuais (embora múltiplas tabelas e userdatas possam compartilhar suas metatabelas). Valores de todos os outros tipos compartilham uma única metatabela por tipo; isto é, há uma única metatabela para todos os números, uma para todas as cadeias, etc. Por padrão, um valor não possui metatabela, mas a biblioteca de cadeias especifica uma metatabela para o tipo string (veja §6.4).

Uma metatabela controla como um objeto se comporta em operações aritméticas, comparações de ordem, concatenação, operação de comprimento, e indexação. Uma metatabela também pode definir uma função

a ser chamada quando um userdata ou uma tabela são recolhidos pelo coletor de lixo. Quando Lua realiza uma dessas operações sobre um valor, ela verifica se esse valor possui uma metatabela com um evento correspondente. Se possui, o valor associado com aquela chave (o metamétodo) controla como Lua realizará 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 com seu nome precedido por dois sublinhados, '__'; por exemplo, a chave para a operação "add" é a cadeia "`__add`".

A semântica dessas operações é melhor explicada por uma função Lua descrevendo como o interpretador executa a operação. O código mostrado aqui em Lua é somente ilustrativo; o comportamento real está codificado no interpretador e é muito mais eficiente do que esta simulação. Todas as funções usadas nestas descrições (`rawget`, `tonumber`, etc.) são descritas em §6.1. Em particular, para recuperar o metamétodo de um dado objeto, usamos a expressão

```
metatabela(obj)[evento]
```

Isso deve ser lido como

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

Isso significa que 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**).

Para os operadores unários `-` e `#`, o metamétodo é chamado com um segundo argumento *dummy*. Esse argumento extra é somente para simplificar a implementação de Lua; ele pode ser removido em versões futuros e portanto não está presente no código a seguir. (Para a maioria dos usos esse argumento extra é irrelevante.)

- **"add":** a `+` operação.

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

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

  Usando essa função, o comportamento da `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 não 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", com a operação `o1 - floor(o1/o2)*o2` como a 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 a partir 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 primitiva de cadeias
  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 cadeias primitivo
  else
    local h = metatable(op).__len
    if h then
      return (h(op))       -- chama tratador com o operando
    elseif type(op) == "table" then
      return #op             -- comprimento de tabela primitivo
    else  -- nenhum tratador disponível: erro
      error(···)
    end
  end
end
```

  Veja §3.4.6 para uma descrição do comprimento de uma tabela.

- **"eq":** a operação `==`. A função `getequalhandler` define como Lua escolhe um metamétodo para igualdade. Um metamétodo é selecionado somente quando ambos os valores sendo comparados possuem o mesmo tipo e o mesmo metamétodo para a operação selecionada, e os valores são ou tabelas ou userdatas completos.

```
function getequalhandler (op1, op2)
  if type(op1) ~= type(op2) or
     (type(op1) ~= "table" and type(op1) ~= "userdata") then
    return nil     -- valores diferentes
  end
  local mm1 = metatable(op1).__eq
  local mm2 = metatable(op2).__eq
  if mm1 == mm2 then return mm1 else return nil end
end
```

  O evento "eq" é definido como a seguir:

```
function eq_event (op1, op2)
  if op1 == op2 then   -- igual primitivo?
    return true   -- valores são iguais
  end
  -- tenta metamétodo
  local h = getequalhandler(op1, op2)
  if h then
    return not not h(op1, op2)
  else
    return false
  end
end
```

Note que o resultado é sempre um booleano.

- **"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 = getbinhandler(op1, op2, "__lt")
    if h then
      return not not h(op1, op2)
    else
      error(···)
    end
  end
end
```

Note que o resultado é sempre um booleano.

- **"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 = getbinhandler(op1, op2, "__le")
    if h then
      return not not h(op1, op2)
    else
      h = getbinhandler(op1, op2, "__lt")
      if h then
        return not h(op2, op1)
      else
        error(···)
      end
    end
  end
end
```

Note que, na ausência de um metamétodo "le", Lua tenta o "lt", assumindo que `a <= b` é equivalente a `not (b < a)`.

Como com os outros operadores de comparação, o resultado é sempre um booleano.

- **"index":** A indexação de acesso `table[key]`. Note que o metamétodo é tentado somente quando `key` não está presente em `table`. (Quando `table` não é uma tabela, nunca uma chave está presente, então o metamétodo é sempre tentado.)

```
function gettable_event (table, key)
  local h
```

```
  if type(table) == "table" then
    local v = rawget(table, key)
    -- se a chave está, retorna o valor primitivo
    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`. Note que o metamétodo é tentado somente quando `key` não está presente em `table`.

```
function settable_event (table, key, value)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    -- se a chave está presente, faz a atribuição primitiva
    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 ele
  end
end
```

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

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

2.5 – Coleta de Lixo

Lua realiza gerenciamento automático de memória. Isso significa que você não precisa se preocupar com a alocação de memória para novos objetos nem com a liberação dela quando os objetos não são mais necessários. Lua gerencia memória automaticamente executando um *coletor de lixo* para coletar todos os *objetos mortos* (isto é, objetos que não são mais acessíveis a partir de Lua). Toda memória usada por Lua está sujeita ao gerenciamento automático: cadeias, tabelas, userdatas, funções, fluxos de execução, estruturas internas, etc. Lua implementa um coletor marca-e-varre (*mark-and-sweep*) incremental. Ele usa

dois números para controlar seus ciclos de coleta de lixo: a *pausa do coletor de lixo* e o *multiplicador de passo do coletor de lixo*. Ambos usam pontos percentuais como unidades (e.g., um valor de 100 significa um valor interno de 1).

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

O multiplicador de passo do coletor de lixo 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 tornam o coletor muito lento e podem fazer com que o coletor nunca termine um ciclo. O padrão é 200, o que significa que o coletor executa no "dobro" da velocidade de alocação de memória.

Se você atribuir ao multiplicador de passo um número muito grande (maior do que 10% do número máximo de bytes que o programa pode usar), o coletor se comporta como um coletor pare-o-mundo. Se você então atribuir 200 à pausa, o coletor se comporta como em versões antigas de Lua, fazendo uma coleta completa toda vez que Lua dobra sua memória em uso.

Você pode mudar esses números chamando `lua_gc` em C ou `collectgarbage` em Lua. Você pode também usar essas funções para controlar o coletor diretamente (e.g., pará-lo e reiniciá-lo).

Como uma característica experimental em Lua 5.2, você pode mudar o modo de operação do coletor de incremental para *generacional*. Um *coletor generacional* assume que a maioria dos objetos morre jovem, e portanto ele percorre somente objetos jovens (criados recentemente). Esse comportamento pode reduzir o tempo usado pelo coletor, mas também incrementa o uso de memória (visto que objetos mortos velhos podem se acumular). Para mitigar esse segundo problema, de tempos em tempos o coletor generacional realiza uma coleta completa. Lembre-se que essa é uma característica experimental; você é bem-vindo a experimentá-la, mas verifique seus ganhos.

2.5.1 – Metamétodos de Coleta de Lixo

Você pode especificar metamétodos do coletor de lixo para tabelas e, usando a API C, para userdatas completos (veja §2.4). Esses metamétodos são também chamados *finalizadores*. Finalizadores permitem você coordenar a coleta de lixo de Lua com o gerenciamento de recursos externos (tais como o fechamento de arquivos, conexões de rede ou de banco de dados, ou a liberação de sua própria memória). Para um objeto (tabela ou userdata) ser finalizada quando coletada, você deve *marcá-la* para finalização. Você marca um objeto para finalização quando você especifica sua metatabela e a metatabela possui um campo indexado pela cadeia "`__gc`". Note que se você especificar uma metatabela sem um campo `__gc` e depois criar esse campo na metatabela, o objeto não será marcado para finalização. Contudo, após um objeto ser marcado, você pode livremente mudar o campo `__gc` de sua metatabela.

Quando um objeto marcado torna-se lixo, ele não é coletado imediatamente pelo coletor de lixo. Ao invés disso, Lua coloca-o em uma lista. Após a coleta, Lua faz o equivalente da função a seguir para cada objeto nessa lista:

```
function gc_event (obj)
  local h = metatable(obj).__gc
  if type(h) == "function" then
    h(obj)
  end
end
```

Ao fim de cada ciclo de coleta de lixo, os finalizadores para os objetos são chamados na ordem inversa em que eles foram marcados para coleta, entre aqueles coletados naquele ciclo; isto é, o primeiro finalizador a ser chamado é o associado com o objeto marcado por último no programa. A execução de cada finalizador pode ocorrer em qualquer ponto durante a execução do código regular.

Por causa que os objetos sendo coletados devem ainda ser usados pelo finalizador, ele (e outros objetos acessíveis somente através dele) deve ser *ressuscitado* por Lua. Geralmente, essa ressurreição é passageira, e a memória do objeto é liberada no próximo ciclo de coleta de lixo. Contudo, se o finalizador guarda o objeto em alguma espaço global (e.g., uma variável global), então há uma ressurreição permanente. Em todo o caso, a memória do objeto é liberada somente quando ele se torna completamente inacessível; seu finalizador nunca será chamado duas vezes.

Quando você fecha um estado (veja `lua_close`), Lua chama os finalizadores de todos os objetos marcados para finalização, seguindo a ordem inversa em que eles foram marcados. Se qualquer finalizador marca novos objetos para coleta durante essa fase, esses novos objetos não serão finalizados.

2.5.2 – 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 coletará esse 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 todo o caso, se a chave 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 contendo o caractere '`k`', as chaves na tabela são fracas. Se `__mode` contém '`v`', os valores na tabela são fracos.

Uma tabela com chaves fracas e valores fortes é também chamada de uma *tabela efêmera*. Em uma tabela efêmera, um valor é considerado alcançável somente se sua chave é alcançável. Em particular, se a única referência para uma chave é através desse valor, o par é removido.

Qualquer mudança na fragilidade de uma tabela terá efeito somente no próximo ciclo de coleta. Em particular, se você mudar a fragilidade para um modo mais forte, Lua poderá ainda coletar alguns itens dessa tabela antes da mudança fazer efeito.

Somente objetos que possuem uma construção explícita são removidos de tabelas fracas. Valores, tais como números e funções C leves, não estão sujeitos à coleta de lixo, e por isso não são removidos de tabelas fracas (a menos que seu valor associado seja coletado). Embora cadeias estejam sujeitas à coleta de lixo, elas não possuem uma construção explícita, e por isso não são removidas de tabelas fracas.

Objetos ressuscitados (isto é, objetos sendo finalizados e objetos acessíveis somente através de objetos sendo finalizados) têm um comportamento especial em tabelas fracas. Eles são removidos de valores fracos antes da execução de seus finalizadores, mas são removidos de chaves fracas somente na próxima coleta após a execução de seus finalizadores, quando tais objetos são realmente liberados. Esse comportamento permite o finalizador acessar propriedades associadas com o objeto através de tabelas fracas.

Se uma tabela fraca está entre os objetos ressuscitados em um ciclo de coleta, ela pode não ser apropriadamente limpa até o próximo ciclo.

2.6 – Co-rotinas

Lua oferece suporte a co-rotinas, também chamadas de *fluxos de execução múltiplos 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, contudo, uma co-rotina somente suspende sua execução através de uma chamada explícita a uma função de cessão.

Você cria uma co-rotina chamando `coroutine.create`. 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 co-rotina.

Você executa uma co-rotina chamando `coroutine.resume`. Quando você chama `coroutine.resume` pela primeira vez, passando como seu primeiro argumento um fluxo de execução retornado por `coroutine.create`, a co-rotina inicia sua execução. na primeira linha de sua função principal. Argumentos extras passados para `coroutine.resume` são passados para a função principal da co-rotina. Após a co-rotina começar sua execução, ela executa até que termine ou *ceda*.

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

Uma co-rotina cede chamando `coroutine.yield`. Quando uma co-rotina cede, a `coroutine.resume` correspondente retorna imediatamente, mesmo se a cessão aconteceu dentro de chamadas de função aninhadas (isto é, não na função principal, mas em uma função diretamente ou indiretamente chamada pela função principal). No caso de uma cessão, `coroutine.resume` também retorna **true**, mais quaisquer valores passados para `coroutine.yield`. Da próxima vez que você reiniciar a mesma co-rotina, ela continua sua execução a partir do ponto onde ela cedeu, com a chamada a `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, reinicia a co-rotina. Quaisquer argumentos passados para essa 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). Ao contrário de `coroutine.resume`, `coroutine.wrap` não captura erros; qualquer erro é propagado para o chamador.

Como um exemplo de como co-rotinas funcionam, considere o seguinte 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 o executa, ele produz 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
```

Você pode também criar e manipular co-rotinas através da API C: veja as funções `lua_newthread`, `lua_resume`, e `lua_yield`.

3 – A Linguagem

Esta seção descreve o léxico, a sintaxe, e a semântica de Lua. Em outras palavras, esta seção descreve quais elementos léxicos são válidos, como eles podem ser combinados, e o que suas combinações significam.

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 não-terminal, palavras-chave são mostradas como **kword**, e outros símbolos terminais são mostrados como '**=**'. A sintaxe completa de Lua pode ser encontrada em §9 no fim deste manual.

3.1 – Convenções Léxicas

Lua é uma linguagem de formato livre. Ela ignora espaços (incluindo quebras de linha) e comentários entre elementos léxicos (tokens), exceto como delimitadores entre nomes e palavras-chave.

Nomes (também chamados de *identificadores*) em Lua podem ser qualquer cadeia de letras, dígitos, e sublinhados, que não iniciam com um dígito. Identificadores são usados para nomear variáveis, campos de tabelas, e rótulos.

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

```
and       break     do        else      elseif    end
false     for       function  goto      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 uma convenção, nomes começando com um sublinhado seguido por letras maiúsculas (tais como `_VERSION`) são reservados para variáveis usadas por Lua.

As seguintes cadeias denotam outros elementos léxicos:

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

Cadeias literais podem ser delimitadas por aspas simples ou duplas balanceadas, e podem conter as seguintes sequências de escape similares às de C: '`\a`' (campainha), '`\b`' (retrocesso), '`\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]). Uma barra invertida seguida por uma quebra de linha de verdade resulta em uma quebra de linha na cadeia. A sequência de escape '`\z`' pula a extensão seguinte de caracteres de espaço em branco, incluindo quebras de linha; ela é particularmente útil para quebrar e identar uma cadeia de literais longa em múltiplas linhas sem adicionar as quebras de linha e espaços ao conteúdo da cadeia.

Um byte em uma cadeia literal pode também ser especificado através de seu valor numérico. Isso pode ser feito com a sequência de escape `\x`*XX*, onde *XX* é uma sequência de exatamente dois dígitos hexadecimais, ou com a sequência de escape `\`*ddd*, onde *ddd* é uma sequência de até três dígitos decimais. (Note que se um escape decimal deve ser seguido por um dígito, ele deve ser expresso usando exatamente três dígitos.) Cadeias em Lua podem conter qualquer valor de 8 bits, incluindo zeros dentro delas, os quais podem ser especificados como '`\0`'.

Cadeias literais podem também ser definidas usando um formato longo delimitado por *colchetes longos*. Definimos um *colchete longo de abertura de nível n* como um abre colchete seguido por *n* sinais de igual seguidos por outro abre colchete. Assim, um abre colchete longo de nível 0 é escrito como `[[`, um abre colchete longo de nível 1 é escrito como `[=[`, e assim por diante. Um *colchete longo de fechamento* é definido similarmente; por exemplo, um colchete longo de fechamento de nível 4 é escrito como `]====]`. Um *literal longo* começa com um colchete longo de abertura de qualquer nível e termina no primeiro colchete longo de fechamento do mesmo nível. Ele pode conter qualquer texto exceto um colchete de fechamento do nível apropriado. Literais expressos dessa forma podem se estender por várias linhas, não interpretam nenhuma sequência de escape, e ignoram colchetes longos de qualquer outro nível. Qualquer tipo de sequência de fim de linha (retorno de carro, quebra de linha, retorno de carro seguido por quebra de linha, ou quebra de linha seguida por retorno de carro) é convertida em uma quebra de linha simples.

Qualquer byte em uma cadeia literal que não é afetada explicitamente pelas regras anteriores representa ele mesmo. Contudo, Lua abre arquivos para parsing em modo texto, e as funções de arquivo do sistema podem ter problemas com alguns caracteres de controle. Assim, é mais seguro representar dados não-textuais como um literal entre aspas com sequências de escape explícitas para caracteres não-textuais.

Por conveniência, quando o colchete longo de abertura é imediatamente seguido por uma quebra de linha, a quebra de linha não é incluída na cadeia. Como um exemplo, em um sistema usando ASCII (no qual '`a`' é codificado como 97, a quebra de linha é codificada como 10, e '`1`' é codificado como 49), as cinco cadeias literais a seguir 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 um parte fracionária opcional e um expoente decimal opcional, marcado por uma letra '`e`' ou '`E`'. Lua também aceita constantes hexadecimais, as quais começam com `0x` ou `0X`. Constantes hexadecimais também aceitam uma parte fracionária opcional mais um expoente binário opcional, marcado por uma letra '`p`' ou '`P`'. Exemplos de contantes numéricas válidas são

```
3     3.0     3.1416     314.16e-2     0.31416E1
0xff  0x0.1E  0xA23p-4   0X1.921FB54442D18P+1
```

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

3.2 – Variáveis

Variáveis são lugares que guardam valores. Há três tipos de variáveis em Lua: variáveis globais, variáveis locais, e campos de tabelas.

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

```
var ::= Nome
```

Nome denota identificadores, como definido em §3.1.

Qualquer nome de variável é assumido ser global a menos que explicitamente declarado como um local (veja §3.3.7). Variáveis locais possuem escopo léxico: variáveis locais podem ser acessadas livremente por funções definidas dentro do seu escopo (veja §3.5).

Antes da primeira atribuição a uma variável, seu valor é **nil**.

Colchetes são usados para indexar uma tabela:

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

O significado de acessos aos campos de uma tabela podem ser modificados por metatabelas. Um acesso a um variável indexada `t[i]` é equivalente a uma chamada `gettable_event(t,i)`. (Veja §2.4 para uma descrição completa da função `gettable_event`. Essa função não é definida nem pode ser chamada em Lua. Usamos ela aqui somente para fins didáticos.)

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

```
var ::= expprefixo '.' Nome
```

Um acesso a uma variável global `x` é equivalente a `_ENV.x`. Devido ao modo que um trecho é compilado, `_ENV` nunca é um nome global (veja §2.2).

3.3 – Comandos

Lua oferece suporte a um conjunto quase convencional de comandos, similar a aqueles em Pascal ou C. Esse conjunto inclui atribuições, estruturas de controle, chamadas de função, e declarações de variáveis.

3.3.1 – Blocos

Um bloco é uma lista de comandos, que são executados sequencialmente:

```
bloco ::= {comando}
```

Lua possui *comandos vazios* que permitem você separar comandos com ponto-e-vírgula, começar um bloco com um ponto-e-vírgula ou escrever dois ponto-e-vírgula em sequência:

```
comando ::= ';'
```

Chamadas de função e atribuições podem começar com um abre parêntese. Essa possibilidade leva a uma ambiguidade na gramática de Lua. Considere o seguinte fragmento:

```
a = b + c
(print or io.write)('done')
```

A gramática poderia vê-lo de duas maneiras:

```
a = b + c(print or io.write)('done')

a = b + c; (print or io.write)('done')
```

O parser corrente sempre vê tais construções da primeira maneira, interpretando o abre parêntese como o começo dos argumentos de uma chamada. Para evitar essa ambiguidade, é uma boa prática sempre preceder com um ponto-e-vírgula comandos que começam com um parêntese:

```
;(print or io.write)('done')
```

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 algumas vezes usados para adicionar um comando **return** no meio de outro bloco (veja §3.3.4).

3.3.2 – Trechos

A unidade de compilação de Lua é chamada de um *trecho*. Sintaticamente, um trecho é simplesmente um bloco:

```
trecho ::= bloco
```

Lua trata um trecho como o corpo de uma função anônima com um número variável de argumentos (veja §3.4.10). Dessa forma, trechos podem definir variáveis locais, receber argumentos, e retornar valores. Além disso, tal função anônima é compilada no escopo de uma variável local externa chamada `_ENV` (veja §2.2). A função resultante sempre tem `_ENV` como seu único upvalue, mesmo se ele não usar essa variável.

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

Trechos podem também ser pré-compilados para uma forma binária; veja o programa `luac` para detalhes. Programas na forma de código fonte e na forma compilada são intercambiáveis; Lua detecta automaticamente o tipo do arquivo e age de acordo.

3.3.3 – Atribuição

Lua permite atribuições múltiplas. Por isso, a sintaxe para atribuição define uma lista de variáveis no lado esquerdo e uma lista de expressões no lado direito. Os elementos em ambas as listas são separados por vírgulas:

```
comando ::= listavars '=' listaexps
listavars ::= var {',' var}
listaexps ::= exp {',' exp}
```

Expressões são discutidas em §3.4.

Antes da atribuição, 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 essa chamada entram na lista de valores, antes do ajuste (exceto quando a chamada é delimitada por parênteses; veja §3.4).

O comando de atribuição primeiro avalia todas as suas expressões e somente então as atribuições são realizadas. Assim 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 (como 3) antes de receber 4. Similarmente, 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 modificada por metatabelas. Uma atribuição a uma variável indexada `t[i] = val` é equivalente a `settable_event(t,i,val)`. (Veja §2.4 para uma descrição completa da função `settable_event`. Essa função não é definida nem pode ser chamada em Lua. Usamos ela aqui somente para fins didáticos.)

Uma atribuição a uma variável global `x = val` é equivalente à atribuição `_ENV.x = val` (veja §2.2).

3.3.4 – 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**, em duas variações (veja §3.3.5).

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

No laço **repeat**–**until**, o bloco interno não termina na palavra chave **until**, mas somente após a condição. Assim, a condição pode se referir a variáveis locais declaradas dentro do corpo do laço.

O comando **goto** transfere o controle do programa para um rótulo. Por razões sintáticas, rótulos em Lua são considerados comandos também:

```
comando ::= goto Nome
comando ::= rótulo
rótulo ::= '::' Nome '::'
```

Um rótulo é visível em todo o bloco onde ele é definido, exceto dentro de blocos aninhados onde um rótulo com o mesmo nome é definido e dentro de funções aninhadas. Um goto pode fazer um desvio para qualquer rótulo visível desde que ele não entre no escopo de uma variável local.

Rótulos e comandos vazios são chamados de *comandos nulos*, uma vez que eles não realizam ações.

O comando **break** termina a execução de um laço **while**, **repeat**, ou **for**, fazendo um desvio para o próximo comando após o laço:

```
comando ::= break
```

Um **break** termina o laço mais interno.

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

```
comando ::= return [listaexps] [';']
```

O comando **return** pode somente ser escrito como o último comando de um bloco. Se for realmente necessário um **return** no meio de um bloco, então um bloco interno explícito pode ser usado, como na expressão idiomática `do return end`, porque agora **return** é o último comando de seu bloco (interno).

3.3.5 – Comando for

O comando **for** possui duas formas: um 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 tem 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 a segunda *exp* através de passos da terceira *exp*. Mais precisamente, 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 somente uma vez, antes do laço começar. Elas devem todas resultar em números.
- *var*, *limite*, e *passo* são variáveis invisíveis. Os nomes mostrados aqui são para fins didáticos somente.
- Se a terceira expressão (o passo) está ausente, então um passo de 1 é usado.
- Você pode usar **break** para sair de um laço **for**.
- A variável de laço `v` é local ao laço; você não pode usar o valor dela após o **for** terminar ou ser interrompido. Se você precisa desse valor, atribua-o a outra variável antes de interromper ou sair do laço.

O comando **for** genérico funciona usando funções, chamadas de *iteradores*. A cada iteração, a função iteradora é chamada para produzir um novo valor, parando quando esse novo valor é **nil**. O laço **for** genérico tem a seguinte sintaxe:

```
comando ::= for listanomes in listaexps do bloco end
listanomes ::= Nome {',' Nome}
```

Um comando **for** como

```
for var_1, ···, var_n in listaexps do bloco end
```

é equivalente ao código:

```
do
  local f, s, var = listaexps
  while true do
    local var_1, ···, var_n = f(s, var)
    if var_1 == nil then break end
```

```
        var = var_1
        bloco
      end
    end
```

Note o seguinte:

- *listaexps* é avaliada somente uma vez. 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 estão aqui para fins didáticos somente.
- Você pode usar **break** para sair de um laço **for**.
- As variáveis de laço *var_i* são locais ao laço; você não pode usar o valor delas após o **for** terminar. Se você precisa desses valores, então atribua-os a outras variáveis antes de interromper ou sair do laço.

3.3.6 – Chamadas de Função como Comandos

Para permitir possíveis efeitos colaterais, chamadas de função podem ser executadas como comandos:

```
comando ::= chamadafunção
```

Nesse caso, todos os valores retornados são descartados. Chamadas de função são explicadas em §3.4.9.

3.3.7 – Declarações Locais

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

```
comando ::= local listanomes ['=' listaexps]
```

Se presente, uma atribuição inicial tem a mesma semântica de uma atribuição múltipla (veja §3.3.3). Caso contrário, todas as variáveis são inicializadas com **nil**.

Um trecho é também um bloco (veja §3.3.2), e dessa forma variáveis locais podem ser declaradas em um trecho fora de qualquer bloco explícito.

As regras de visibilidade para variáveis locais são explicadas em §3.5.

3.4 – Expressões

As expressões básicas em Lua são as seguintes:

```
exp ::= expprefixo
exp ::= nil | false | true
exp ::= Número
exp ::= Cadeia
exp ::= deffunção
exp ::= construtortabela
exp ::= '...'
exp ::= exp opbin exp
exp ::= opunário exp
prefixexp ::= var | chamadafunção | '(' exp ')'
```

Números e cadeias de literais são explicados em §3.1; variáveis são explicadas em §3.2; definições de funções são explicadas em §3.4.10; chamadas de funções são explicadas em §3.4.9; construtores de tabelas são explicados em §3.4.8. Expressões *vararg*, denotadas por três pontos (Char{...}), somente podem ser usadas quando diretamente dentro de uma função vararg; elas são explicadas em §3.4.10.

Operadores binários compreendem operadores aritméticos (veja §3.4.1), operadores relacionais (veja §3.4.3), operadores lógicos (veja §3.4.4), e o operador de concatenação (veja §3.4.5). Operadores unários compreendem o menos unário (veja §3.4.1), o **not** unário (veja §3.4.4), e o *operador de comprimento* unário (veja §3.4.6). Tanto chamadas de função como expressões vararg podem resultar em múltiplos

valores. Se uma chamada de função é usada como um comando (veja §3.3.6), então 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 expressão esteja entre parênteses). Em todos os outros contextos, Lua ajusta a lista de resultados para um elemento, ou descartando todos os valores exceto o primeiro ou adicionando um único **nil** se não há nenhum valor.

Aqui estão alguns exemplos:

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

a,b,c = x, f()     -- f() é ajustada para 2 resultados
a,b,c = f()        -- f() é ajustada para 3 resultados
return f()         -- retorna todos os resultados de f()
return ...         -- retorna todos os parâmetros de vararg recebidos
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 vararg
{f(), nil}         -- f() é ajustada para 1 resultado
```

Qualquer expressão entre parênteses sempre resulta em um único valor. Portanto, `(f(x,y,z))` é sempre um único valor, mesmo se `f` retorna vários valores. (O valor de `(f(x,y,z))` é o primeiro valor retornado por `f` ou **nil** se `f` não retorna nenhum valor.)

3.4.1 – Operadores Aritméticos

Lua oferece suporte aos operadores aritméticos usuais: os binários `+` (adição), `-` (subtração), `*` (multiplicação), `/` (divisão), `%` (módulo), e `^` (exponenciação); e o unário `-` (negação matemática). Se os operandos são números, ou cadeias que podem ser convertidas em números (veja §3.4.2), então todas as operações possuem o significado usual. A exponenciação funciona para qualquer expoente. Por exemplo, `x^(-0.5)` computa 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 que arredonda o quociente para menos infinito.

3.4.2 – Coerção

Lua provê conversão automática entre valores string e number em tempo de execução. Qualquer operação aritmética aplicada a uma cadeia tenta converter essa cadeia para um número, seguindo as regras do analisador léxico de Lua. (A cadeia pode ter espaços à esquerda e à direita e um sinal.) Inversamente, sempre que um número é usado onde uma cadeia é esperada, o número é convertido em uma cadeia, em um formato razoável. Para um controle completo sobre como números são convertidos em cadeias, use a função `format` da biblioteca de cadeias (veja `string.format`).

3.4.3 – Operadores Relacionais

Os operadores relacionais em Lua são

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

Esses operadores sempre resultam em **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 são comparados da maneira usual. Tabelas, userdatas, e fluxos de execução são comparados por referência: dois objetos são considerados iguais somente se eles são o mesmo objeto. Toda vez que você cria um novo objeto (uma tabela, um userdata, ou um fluxo de execução), esse novo objeto é diferente de qualquer objeto existente anteriormente. Fechos com a mesma referência são sempre iguais. Fechos com qualquer

diferença detectável (comportamento diferente, definição diferente) são sempre diferentes.

Você pode mudar a maneira que Lua compara tabelas e userdatas usando o metamétodo "eq" (veja §2.4).

As regras de conversão §3.4.2 não se aplicam a comparações de igualdade. Assim, `"0"==0` avalia para **false**, e `t[0]` e `t["0"]` denotam entradas diferentes em uma tabela.

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

Os operadores de ordem funcionam como a seguir. Se ambos os argumentos são números, então eles são comparados como tais. Caso contrário, se ambos os argumentos são cadeias, então os valores delas são comparados de acordo com o idioma (*locale*) atual. Caso contrário, Lua tenta chamar o metamétodo "lt" ou o metamétodo "le" (veja §2.4). Uma comparação `a > b` é traduzida para `b < a` e `a >= b` é traduzida para `b <= a`.

3.4.4 – Operadores Lógicos

Os operadores lógicos em Lua são **and**, **or**, e **not**. Assim como as estruturas de controle (veja §3.3.4), todos os operadores lógicos consideram **false** e **nil** como falso e qualquer coisa diferente como verdeiro.

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 este valor é diferente de **nil** e **false**; caso contrário, **or** retorna seu segundo argumento. Tanto **and** como **or** usam avaliação de curto-circuito; isto é, o segundo operando é avaliado somente se 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.)

3.4.5 – Concatenação

O operador de concatenação de cadeias em Lua é denotado por ponto ponto ('`..`'). Se ambos os operandos são cadeias ou números, então eles são convertidos para cadeias de acordo com as regras mencionadas em §3.4.2. Caso contrário, o metamétodo `__concat` é chamado (veja §2.4).

3.4.6 – O Operador de Comprimento

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

Um programa pode modificar o comportamento do operador de comprimento para qualquer valor exceto cadeias através do metamétodo `__len` (veja §2.4).

A menos que o metamétodo `__len` seja fornecido, o comprimento de uma tabela `t` é definido somente se a tabela é uma *sequência*, isto é, o conjunto de suas chaves numéricas positivas é igual a *{1..n}* para algum inteiro não negativo *n*. Nesse caso, *n* é seu comprimento. Note que uma tabela como

```
{10, 20, nil, 40}
```

não é uma sequência, pois ela tem a chave `4` mas não não tem a chave `3`. (Assim, não existe um *n* tal que o conjunto *{1..n}* seja igual ao conjunto de chaves numéricas positivas dessa tabela.) Note, contudo, que chaves não numéricas não interferem no fato de uma tabela ser uma sequência.

3.4.7 – Precedência

A precedência dos operadores em Lua segue a tabela a seguir, da menor prioridade para a maior:

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

Como 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 outros operadores binários são associativos à esquerda.

3.4.8 – Construtores de Tabelas

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

```
construtortabela ::= '{' [listacampos] '}'
listacampos ::= campo {sepcampos campo} [sepcampos]
campo ::= '[' exp ']' '=' exp | Nome '=' exp | exp
sepcampos ::= ',' | ';'
```

Cada campo da forma `[exp1] = exp2` adiciona à nova tabela uma entrada com chave `exp1` e valor `exp2`. Um campo da forma `nome = exp` é equivalente a `["nome"] = exp`. Finalmente, campos da forma `exp` são equivalentes a `[i] = exp`, onde `i` são números inteiros consecutivos, começando com 1. Campos nos outros formatos não afetam essa 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 tem a forma `exp` e a expressão é uma chamada de função ou uma expressão vararg, então todos os valores retornados por essa expressão entram na lista consecutivamente (veja §3.4.9).

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

3.4.9 – Chamadas de Função

Uma chamada de função em Lua tem a seguinte sintaxe:

```
chamadafunção ::= expprefixo args
```

Em uma chamada de função, primeiro expprefixo e args são avaliados. Se o valor de expprefixo tem tipo *function*, então essa 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 da chamada original (veja §2.4).

A forma

```
chamadafunção ::= expprefixo ':' Nome args
```

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

Argumentos possuem a seguinte sintaxe:

```
args ::= '(' [listaexps] ')'
args ::= construtortabela
args ::= Cadeia
```

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

Uma chamada da forma `return` *`chamadafunção`* 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 chamadora. Logo, não há limite no número de chamadas finais aninhadas que um programa pode executar. Contudo, uma chamada final apaga qualquer informação de depuração a respeito da função chamadora. Note que uma chamada final somente acontece com uma sintaxe particular, onde o **return** possui uma única chamada de função como argumento; essa sintaxe faz a função chamadora retornar exatamente os valores retornados pela função chamada. Assim, nenhum dos exemplos a seguir é uma chamada final:

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

3.4.10 – Definições de Funções

A sintaxe para uma definição de função é

```
deffunção ::= função corpofunção
corpofunção ::= '(' [listapars] ')' bloco end
```

O seguinte açúcar sintático simplifica definições de funções:

```
comando ::= function nomefunção corpofunção
comando ::= local function Nome corpofunção
nomefunção ::= 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
```

não para

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

(Isso somente faz diferença quando o corpo da função contém referências para `f`.)

Uma definição de função é uma expressão executável, cujo valor tem tipo *function*. Quando Lua pré-compila um trecho, todos os corpos de funções do trecho são pré-compilados também. Então, sempre que Lua executa a definição de uma função, a função é *instanciada* (ou *fechada*). Essa instância de função (ou *fecho*) é o valor final da expressão.

Parâmetros agem como variáveis locais que são inicializadas com os valores dos argumentos:

```
listapars ::= listanomes [',' '...'] | '...'
```

Quando uma função é chamada, a lista de argumentos é ajustada para o comprimento da lista de parâmetros, a menos que a função seja uma *função vararg*, a qual é indicada por três pontos ('`...`') no fim de 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 à função através de uma *expressão vararg*, que também é denotada por três pontos. O valor desta expressão é uma lista de todos os argumentos extras de fato, similar a uma função com múltiplos resultados. Se uma expressão vararg é usada dentro de outra expressão ou no meio de uma lista de expressões, então sua lista 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 esteja entre 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
```

Em seguida, temos o seguinte mapeamento de argumentos para parâmetros e para 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** (veja §3.3.4). Se o controle alcança o fim de uma função sem encontrar um comando **return**, então a função retorna sem nenhum resultado.

Há um limite que depende do sistema para o número de valores que uma função pode retornar. Esse limite é garantidamente maior do que 1000.

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

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

é açúcar sintático para

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

3.5 – Regras de Visibilidade

Lua é uma linguagem com escopo léxico. O escopo de uma variável local começa no primeiro comando após a sua declaração e vai até o último comando não nulo do bloco mais interno que inclui a declaração. Considere o seguinte exemplo:

```
x = 10                -- variável global
do                    -- novo bloco
  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 conta das regras de escopo léxico, variáveis locais podem ser livremente acessadas por funções definidas dentro de seu escopo. Uma variável local usada por uma função mais interna é chamada de *upvalue*, ou de *variável local externa*, dentro da função mais interna.

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

```
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 desses fechos usa uma variável `y` diferente, enquanto todos eles compartilham o mesmo `x`.

4 – A Interface de Programação da Aplicação

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

Mesmo quando usamos o termo "função", qualquer facilidade na API pode ser provida como uma macro ao invés. Exceto onde dito de outra maneira, todas essas macros usam cada um de seus argumentos exatamente uma vez (exceto o primeiro argumento, que é sempre um estado Lua), e assim não geram qualquer efeito colateral oculto.

Como na maioria das bibliotecas C, as funções da API Lua não verificam a validade ou a consistência de seus argumentos. Contudo, você pode mudar esse comportamento compilando Lua com a macro `LUA_USE_APICHECK` definida.

4.1 – A Pilha

Lua usa uma *pilha virtual* para passar e receber valores de C. Cada elemento nessa pilha representa um valor Lua (**nil**, número, cadeia, 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 estão ativas. Essa pilha inicialmente contém quaisquer argumentos para a função C e é onde a função C empilha seus resultados para serem retornados para o chamador (veja `lua_CFunction`).

Por conveniência, a maioria das operações de consulta na API não seguem uma disciplina de pilha estrita. Em vez disso, elas podem se referir a qualquer elemento na pilha usando um *índice*: Um índice positivo representa um posição absoluta na pilha (começando de 1); um índice negativo representa uma posição relativa ao topo da pilha. Mais especificamente, se a pilha tem *n* elementos, então o índice 1 representa o primeiro elemento (isto é, o elemento que foi empilhado 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.

4.2 – Tamanho da Pilha

Quando você interage com a API de Lua, você é responsável por assegurar consistência.. Em particular, *você é responsável por controlar o estourou da pilha*. Você pode usar a função `lua_checkstack` para assegurar que a pilha possui espaços extras ao empilhar novos elementos.

Sempre que Lua chama C, ela assegura que a pilha possui pelo menos `LUA_MINSTACK` espaços extras. `LUA_MINSTACK` é definida como 20, assim geralmente você não precisa se preocupar com espaços extras a menos que seu código tenha laços empilhando elementos na pilha.

Quando você chama uma função Lua sem um número fixo de resultados (veja `lua_call`), Lua garante que a pilha tem espaço suficiente para todos os resultados, mas ela não garante qualquer espaço extra. Assim, antes de empilhar qualquer coisa na pilha após uma chamada desse tipo você deve usar `lua_checkstack`.

4.3 – Índices Válidos e Aceitáveis

Qualquer função na API que recebe índices da pilha funciona somente com *índices válidos* ou *índices aceitáveis*.

Um *índice válido* é um índice que se refere a uma posição real dentro da pilha, isto é, sua posição está entre 1 e o topo da pilha (`1 ≤ abs(índice) ≤ topo`). Geralmente, funções que podem modificar os valores em um índice exigem índices válidos.

A menos que dito de outra maneira, qualquer função que aceita índices válidos também aceita *pseudo-índices*, os quais representam alguns valores de Lua que são acessíveis para código C mas que não estão na pilha. Pseudo-índices são usados para acessar o registro e os upvalues de uma função C (veja §4.4).

Funções que não precisam de uma posição na pilha específica, mas somente de um valor na pilha (e.g., funções de consulta), podem ser chamadas com índices aceitáveis. Um *índice aceitável* pode ser qualquer índice válido, incluindo os pseudo-índices, mas também pode ser qualquer inteiro positivo após o topo da pilha dentro do espaço alocado para a pilha, isto é, índices até o tamanho da pilha. (Note que 0 nunca é um índice aceitável.) Exceto quando dito de outra maneira, funções na API funcionam com índices aceitáveis.

Índices aceitáveis servem para evitar testes extras relacionados ao topo da pilha ao consultar a pilha. Por exemplo, um função C pode consultar seu terceiro argumento sem a necessidade de primeiro verificar se há um terceiro argumento, isto é, sem a necessidade de verificar se 3 é um índice válido.

Para funções que podem ser chamadas com índices aceitáveis, qualquer índice não válido é tratado como se ele contivesse um valor de um tipo virtual `LUA_TNONE`, o qual comporta-se como um valor **nil**.

4.4 – Fechos C

Quando uma função C é criada, é possível associar alguns valores a ela, criando assim um *fecho C* (veja `lua_pushcclosure`); esses valores são chamados de *upvalues* e são acessíveis à função sempre que ela é chamada.

Sempre que uma função C é chamada, seus upvalues são posicionados em pseudo-índices específicos. Esses pseudo-índices são produzidos 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 corrente (mas não

maior do que 256), produz um índice aceitável, porém inválido.

4.5 – Registro

Lua provê um *registro*, uma tabela pré-definida que pode ser usada por qualquer código C para armazenar quaisquer valores Lua que ele precise armazenar . A tabela de registro está sempre localizada no pseudo-índice `LUA_REGISTRYINDEX`, que é um índice válido. Qualquer biblioteca C pode armazenar dados nessa tabela, mas ela deve tomar cuidado para escolher chaves que sejam diferentes daquelas usadas por outras bibliotecas, para evitar colisões. Tipicamente, você deve usar como chave uma cadeia contendo o nome de sua biblioteca, ou um userdata leve com o endereço de um objeto C no seu código, ou qualquer objeto Lua criado por seu código. Assim como com nomes globais, chaves que são cadeias começando com um sublinhando seguido por letras maiúsculas são reservadas para Lua.

As chaves inteiras no registro são usadas pelo mecanismo de referência, implementado pela biblioteca auxiliar, e por alguns valores pré-definidos. Portanto, chaves inteiras não devem ser usadas para outros propósitos.

Quando você cria um novo estado Lua, o registro dele vem com alguns valores pré-definidos. Esses valores pré-definidos são indexados com chaves inteiras definidas como constantes em `lua.h`. As seguintes constantes são definidas:

- **`LUA_RIDX_MAINTHREAD`:** Nesse índice o registro tem o fluxo de execução principal do estado. (O fluxo de execução principal é aquele criado junto com o estado.)
- **`LUA_RIDX_GLOBALS`:** Nesse índice o registro tem o ambiente global.

4.6 – Tratamento de Erros em C

Internamente, Lua usa a facilidade `longjmp` de C para tratar erros. (Você pode também escolher usar exceções se você compilar Lua como C++; procure por `LUAI_THROW` no código fonte.) Quando Lua enfrenta qualquer erro (como um erro de alocação de memória, erros de tipo, erros sintáticos, e erros de tempo de execução) ela *lança* um erro; isto é, ela faz um desvio longo. Um *ambiente protegido* usa `setjmp` para estabelecer um ponto de recuperação; qualquer erro desvia para o ponto de recuperação ativo mais recente.

Se um erro acontece fora de qualquer ambiente protegido, Lua chama uma *função de pânico* (veja `lua_atpanic`) e então chama `abort`, saindo portanto da aplicação hospedeira. Sua função de pânico pode evitar essa saída nunca retornando (e.g., fazendo um desvio longo para seu próprio ponto de recuperação fora de Lua).

A função de pânico roda como se ela fosse um tratador de mensagens (veja §2.3); em particular, a mensagem de erro está no topo da pilha. Contudo, não há garantias sobre o espaço da pilha. Para empilhar qualquer coisa na pilha, a função de pânico deve primeiro verificar o espaço disponível (veja §4.2).

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

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

4.7 – Tratando Cessões em C

Internamente, Lua usa a facilidade `longjmp` de C para ceder uma co-rotina. Logo, se uma função `foo` chama uma função da API e essa função da API cede (diretamente ou indiretamente através da chamada a outra função que cede), Lua não pode retornar mais para `foo`, pois `longjmp` remove seu *frame* da pilha de C.

Para evitar esse tipo de problema, Lua lança um erro sempre que ela tenta ceder através de uma chamada da API, exceto para três funções: `lua_yieldk`, `lua_callk`, e `lua_pcallk`. Todas essas funções recebem uma *função de continuação* (como um parâmetro chamado `k`) para continuar a execução após uma cessão.

Precisamos estabelecer alguma terminologia para explicar continuações. Temos uma função C chamada a partir de Lua que chamaremos de *função original*. Essa função original então chama uma dessas três funções da API C, as quais chamaremos de *funções chamadas*, que então cedem o fluxo de execução corrente. (Isso pode acontecer quando a função chamada é `lua_yieldk`, ou quando a função chamada é `lua_callk` ou `lua_pcallk` e a função chamada por elas cede.)

Suponha que o fluxo de execução rodando ceda enquanto executa a função chamada. Após o fluxo de execução recomeçar, ele em algum momento terminará executando a função chamada. Contudo, a função chamada não pode retornar para a função original, pois seu frame na pilha de C foi destruído pela cessão. Ao invés disso, Lua chama uma *função de continuação*, que foi fornecida como um argumento para a função chamada. Como o nome indica, a função de continuação deve continuar a tarefa da função original.

Lua trata a função de continuação como se ela fosse a função original. A função de continuação recebe a mesma pilha Lua da função original, no mesmo estado que ela estaria se a função chamada tivesse retornado. (Por exemplo, após um `lua_callk` a função e seus argumentos são removidos da pilha e substituídos pelos resultados da chamada.) Ela também tem os mesmos upvalues. Seja qual for o retorno, ele é tratado por Lua como se fosse o retorno da função original.

A única diferença no estado Lua entre a função original e sua continuação é o resultado de uma chamada a `lua_getctx`.

4.8 – Funções e Tipos

Aqui listamos todas as funções e tipos da API C em ordem alfabética. Cada função possui um indicador como este: [-o, +p, *x*]

O primeiro campo, `o`, é quantos elementos a função desempilha da pilha. O segundo campo, `p`, é quantos elementos a função empilha na pilha. (Qualquer função sempre empilha seus resultados após desempilhar seus argumentos.) Um campo da forma `x|y` significa que a função pode empilhar (ou desempilhar) `x` ou `y` elementos, dependendo da situação; um ponto de interrogação '`?`' significa que não podemos saber quantos elementos a função desempilha/empilha olhando somente seus argumentos (e.g., eles podem depender do que está na pilha). O terceiro campo, `x`, diz se a função pode lançar erros: '`-`' significa que a função nunca lança nenhum erro; '`e`' significa que a função pode lançar erros; '`v`' significa que a função pode lançar um erro de propósito.

lua_absindex

```
int lua_absindex (lua_State *L, int idx);
```
[-0, +0, –]

Converte o índice aceitável `idx` em um índice absoluto (isto é, um que não depende do topo da pilha).

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 por estados Lua. A função alocadora deve prover uma funcionalidade similar a `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 ou algum código sobre o que está sendo alocado; `nsize`, o novo tamanho do bloco.

Quando `ptr` não é `NULL`, `osize` é o tamanho do bloco apontado por `ptr`, isto é, o tamanho fornecido quando ele foi alocado ou realocado.

Quando `ptr` é `NULL`, `osize` codifica o tipo de objeto que Lua está alocando. `osize` é `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, ou `LUA_TTHREAD` quando (e somente quando) Lua está criando um novo objeto desse tipo. Quando `osize` é algum outro valor, Lua está alocando memória para outra coisa.

Lua assume o seguinte comportamento da função alocadora:

Quando `nsize` é zero, o alocador deve comportar-se como `free` e retornar `NULL`.

Quando `nsize` não é zero, o alocador deve comportar-se como `realloc`. O alocador retorna `NULL` se e somente se ele não pode cumprir a requisição. Lua assume que o alocador nunca falha quando `osize >= nsize`.

Aqui está uma implementação simples para a função alocadora. 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);
     }
```

Note que C Padrão assegura que `free(NULL)` não tem efeito e que `realloc(NULL, size)` é equivalente a `malloc(size)`. Esse código assume que `realloc` não falha ao encolher um bloco. (Embora C Padrão não assegure esse comportamento, parece ser uma suposição segura.)

lua_arith

```
void lua_arith (lua_State *L, int op);
```
[-(2|1), +1, *e*]

Realiza uma operação aritmética sobre os dois valores (ou um, no caso de negação) no topo da pilha, com o valor no topo sendo o segundo operando, desempilha esses valores, e empilha o resultado da operação. A função segue a semântica do operador Lua correspondente (isto é, ela pode chamar metamétodos).

O valor de `op` deve ser uma das seguintes constantes:

- **`LUA_OPADD`:** faz adição (`+`)
- **`LUA_OPSUB`:** faz subtração (`-`)
- **`LUA_OPMUL`:** faz multiplicação (`*`)
- **`LUA_OPDIV`:** faz divisão (`/`)
- **`LUA_OPMOD`:** faz módulo (`%`)
- **`LUA_OPPOW`:** faz exponenciação (`^`)
- **`LUA_OPUNM`:** faz negação matemática (`-` unário)

lua_atpanic

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

Estabelece uma nova função de pânico e retorna a antiga (veja §4.6).

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 para a função são empilhados em ordem direta; isto é, o primeiro argumento é empilhado primeiro. Finalmente você chama `lua_call`; `nargs` é o número de argumentos que você empilhou na pilha. Todos os argumentos e o valor da função são retirados da pilha quando a função é chamada. Os resultados da função são colocados na pilha quando a função retorna. O número de resultados é ajustado para `nresults`, a menos que `nresults` seja `LUA_MULTRET`. Nesse caso, todos os resultados da função são empilhados. Lua cuida para que os valores retornados se ajustem no espaço da pilha. Os resultados da função são colocados na pilha em ordem direta (o primeiro resultado é empilhado primeiro), de modo que após a 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 ele em C:

```
lua_getglobal(L, "f");                  /* função a ser chamada  */
lua_pushstring(L, "how");                       /* 1o argumento  */
lua_getglobal(L, "t");                  /* tabela a ser indexada */
lua_getfield(L, -1, "x");   /* empilha 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_setglobal(L, "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_callk

```
void lua_callk (lua_State *L, int nargs, int nresults, int ctx,
                lua_CFunction k);
```
[-(nargs + 1), +nresults, *e*]

Esta função comporta-se exatamente como `lua_call`, mas permite a função chamada ceder (veja §4.7).

lua_CFunction

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

Tipo para funções C.

A fim de se comunicar apropriadamente com Lua, uma função C deve usar o seguinte protocolo, que 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). Assim, quando a função começa, `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 apropriadamente descartado por Lua. Como uma função Lua, uma função C chamada por Lua também pode retornar muitos resultados.

Como um exemplo, a função a seguir 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, –]

Assegura que há no mínimo `extra` espaços de pilha disponíveis na pilha. Retorna falso se não pode cumprir a requisição, pois faria com que a pilha fosse maior do que um tamanho máximo fixo (tipicamente

pelo menos uns poucos milhares de elementos) ou porque não pode alocar memória para o novo tamanho da pilha. Essa função nunca encolhe a pilha; se a pilha já é maior do que o novo tamanho, ela não é modificada.

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 memória dinâmica usada por esse estado. Em várias plataformas, você pode não precisar chamar essa função, pois todos os recursos são naturalmente liberados quando o programa hospedeiro termina. Por outro lado, programas que executam por muito tempo e que criam múltiplos estados, tais como *daemons* ou servidores web, podem precisar fechar estados assim que eles não sejam necessários.

lua_compare

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

Compara dois valores Lua. Retorna 1 se o valor no índice `index1` satisfaz `op` quando comparado com o valor no índice `index2`, seguindo a semântica do operador Lua correspondente (isto é, pode chamar metamétodos). Caso contrário retorna 0. Também retorna 0 se algum dos índices não é válido.

O valor de `op` deve ser uma das seguintes constantes:

- **`LUA_OPEQ`:** comparação de igualdade (==)
- **`LUA_OPLT`:** comparação de menor que (<)
- **`LUA_OPLE`:** comparação de menor ou igual (<=)

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. Se `n` é 1, o resultado é o único valor no topo da pilha (isto é, a função não faz nada); se `n` é 0, o resultado é a cadeia vazia. A concatenação é realizada seguindo a semântica usual de Lua (veja §3.4.5).

lua_copy

```
void lua_copy (lua_State *L, int fromidx, int toidx);
```
[-0, +0, –]

Move o elemento no índice `fromidx` para o índice válido `toidx` sem deslocar nenhum elemento (substituindo portanto o valor naquela posição).

lua_createtable

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

Cria uma nova tabela vazia e a empilha na pilha. O parâmetro `narr` é uma dica de quantos elementos a tabela terá como uma sequência; o parâmetro `nrec` é uma dica de quantos outros elementos a tabela terá. Lua pode usar essas dicas para pré-alocar memória para a nova tabela. Essa pré-alocação é útil para desempenho quando você sabe de antemão quantos elementos a tabela terá. Caso contrário você pode usar a função `lua_newtable`.

lua_dump

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

Descarrega uma função como um trecho binário. Recebe uma função Lua no topo da pilha e produz um trecho binário que, se carregado novamente, resulta em uma função equivalente à que foi descarregada. Como ela produz partes do trecho, `lua_dump` chama a função `writer` (veja `lua_Writer`) com o `data` fornecido para escrevê-lo.

O valor retornado é o código de erro retornado pela última chamada a `writer`; 0 significa que não houve

erros.

Esta função não desempilha a função Lua da pilha.

lua_error

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

Gera um erro Lua. A mensagem de erro (a qual pode efetivamente ser um valor Lua de qualquer tipo) deve estar no topo da pilha. Esta função faz um desvio longo, e portanto nunca retorna (veja `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`:** para o coletor de lixo.
- **`LUA_GCRESTART`:** reinicia o coletor de lixo.
- **`LUA_GCCOLLECT`:** realiza um ciclo de coleta de lixo completo.
- **`LUA_GCCOUNT`:** retorna a quantidade de memória (em Kbytes) correntemente usada por Lua.
- **`LUA_GCCOUNTB`:** retorna o resto da divisão da quantidade de bytes de memória correntemente usada 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 um modo não especificado. Se você quiser controlar o tamanho do passo você deve ajustar experimentalmente o valor de `data`. A função retorna 1 se o passo terminou um ciclo de coleta de lixo.
- **`LUA_GCSETPAUSE`:** estabelece `data` como o novo valor para a *pausa* do coletor (veja §2.5). A função retorna o valor anterior da pausa.
- **`LUA_GCSETSTEPMUL`:** estabelece `data` como o novo valor para o *multiplicador de passo* do coletor (veja §2.5). A função retorna o valor anterior do multiplicador de passo.
- **`LUA_GCISRUNNING`:** retorna um booleano que diz se o coletor está executando (i.e., não parou).
- **`LUA_GCGEN`:** muda o coletor para o modo generacional (veja §2.5).
- **`LUA_GCINC`:** muda o coletor para o modo incremental. Este é o modo padrão.

Para mais detalhes sobre essas opções, veja `collectgarbage`.

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_getctx

```
int lua_getctx (lua_State *L, int *ctx);
```
[-0, +0, –]

Esta função é chamada por uma função de continuação (veja §4.7) para recuperar o estado do fluxo de execução e uma informação de contexto.

Quando chamada na função original, `lua_getctx` sempre retorna `LUA_OK` e não modifica o valor de seu argumento `ctx`. Quando chamada dentro de uma função de continuação, `lua_getctx` retorna `LUA_YIELD` e atribui o valor de `ctx` para a informação de contexto (o valor passado como o argumento `ctx` para a função chamada junto com a função de continuação).

Quando a função chamada é `lua_pcallk`, Lua pode também chamar sua função de continuação para tratar erros durante a chamada. Isto é, em consequência de um erro na função chamada por `lua_pcallk`, Lua pode não retornar para a função original mas ao invés disso pode chamar a função de continuação. Nesse caso, uma chamada a `lua_getctx` retornará o código de erro (o valor que seria retornado por `lua_pcallk`); o valor de `ctx` será atribuído à informação de contexto, como no caso de uma cessão.

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 fornecido. Como em Lua, esta função pode disparar um metamétodo para o evento "índice" (veja §2.4).

lua_getglobal

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

Coloca na pilha o valor da global `name`.

lua_getmetatable

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

Coloca na pilha a metatabela do valor no índice fornecido. 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 fornecido e `k` é o valor no topo da pilha.

Esta função desempilha a chave da pilha (colocando o valor resultante em seu lugar). Como em Lua, esta função pode disparar um metamétodo para o "índice" do evento (veja §2.4).

lua_gettop

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

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

lua_getuservalue

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

Coloca na pilha o valor Lua associado com o userdata no índice fornecido. Esse valor Lua deve ser uma tabela ou **nil**.

lua_insert

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

Move o elemento no topo para índice válido fornecido, deslocando os elementos acima desse índice para abrir espaço. Esta função não pode ser chamada com um pseudo-índice, pois um pseudo-índice não é uma posição na pilha de verdade.

lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

O tipo usado pela API de Lua para representar valores inteiros com sinal.

Por padrão ele 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 fornecido é um booleano, e 0 caso contrário.

lua_iscfunction

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

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

lua_isnone

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

Retorna 1 se o índice fornecido não é válido, e 0 caso contrário.

lua_isnoneornil

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

Retorna 1 se o índice fornecido não é válido ou se o valor nesse 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 fornecido é um número ou uma cadeia 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 fornecido é uma cadeia 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 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 fornecido é um fluxo de execução, e 0 caso contrário.

lua_isuserdata

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

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

lua_len

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

Retorna o "comprimento" do valor no índice fornecido; é equivalente ao operador '`#`' em Lua (veja §3.4.6). O resultado é colocado na pilha.

lua_load

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

Carrega um trecho Lua (sem executá-lo). Se não há erros, `lua_load` coloca o trecho compilado como uma função Lua no topo da pilha. Caso contrário, empilha uma mensagem de erro.

Os valores retornados por `lua_load` são:

- **`LUA_OK`:** sem erros;
- **`LUA_ERRSYNTAX`:** erro de sintaxe durante a pré-compilação;
- **`LUA_ERRMEM`:** erro de alocação de memória;
- **`LUA_ERRGCMM`:** erro ao executar um metamétodo `__gc`. (Este erro não tem relação com o trecho sendo carregado. Ele é gerado pelo coletor de lixo.)

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

O argumento `source` dá um nome para o trecho, o qual é usado para mensagens de erro e em informações de depuração (veja §4.9).

`lua_load` automaticamente detecta se o trecho é texto ou binário e o carrega de acordo (veja o programa `luac`). A cadeia `mode` funciona como na função `load`, com a adição que um valor `NULL` é equivalente à cadeia "`bt`".

`lua_load` usa a pilha internamente, assim a função de leitura deve sempre deixar a pilha inalterada ao retornar.

Se a função resultante tem um upvalue, o valor atribuído a esse upvalue é o ambiente global armazenado no índice `LUA_RIDX_GLOBALS` no registro (veja §4.5). Ao carregar trechos principais, esse upvalue será a variável `_ENV` (veja §2.2).

lua_newstate

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

Cria um novo fluxo de execução rodando em um novo estado, independente. Retorna `NULL` se não pode criar o fluxo ou o estado (devido à falta de memória). O argumento `f` é a função alocadora; Lua faz toda alocação de memória para esse estado através dessa função. O segundo argumento, `ud`, é um ponteiro opaco que Lua passa para o alocador em cada chamada.

lua_newtable

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

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, *e*]

Cria um novo fluxo de execução, coloca-o na pilha, e retorna um ponteiro para um `lua_State` que representa esse novo fluxo. O novo fluxo de execução retornado por essa função compartilha com o fluxo original seu ambiente global, mas possui uma pilha de execução independente.

Não há uma função explícita para fechar ou destruir um fluxo de execução. Fluxos de execução estão sujeitos à coleta de lixo, como qualquer objeto Lua.

lua_newuserdata

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

Esta função aloca um novo bloco de memória com o tamanho fornecido, coloca na pilha um novo userdata completo com o endereço do bloco, e retorna esse endereço. O programa hospedeiro pode usar livremente essa memória.

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 após o índice fornecido). 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'; mantém 'key' para a próxima iteração */
  lua_pop(L, 1);
}
```

Ao percorrer uma tabela, não chame `lua_tolstring` diretamente sobre uma chave, a menos que você saiba que a chave é realmente uma cadeia. Lembre que `lua_tolstring` pode modificar o valor no índice fornecido; isso confunde a próxima chamada a `lua_next`.

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

lua_Number

```
typedef double lua_Number;
```

O tipo de números em Lua. Por padrão, ele é double, mas isso pode ser modificado em `luaconf.h`. Através desse arquivo de configuração você pode mudar Lua para operar com outro tipo para números (e.g., float ou long).

lua_pcall

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

Chama uma função em modo protegido.

Tanto `nargs` quanto `nresults` possuem o mesmo significado que tinham 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 `msgh` é 0, então a mensagem de erro retornada na pilha é exatamente a mensagem de erro original. Caso contrário, `msgh` é o índice na pilha de um *tratador de mensagens*. (Na implementação corrente, esse índice não pode ser um pseudo-índice.) Em caso de erros de tempo de execução, essa função será chamada com a mensagem de erro e seu valor de retorno será a mensagem retornada na pilha por `lua_pcall`.

Tipicamente, o tratador de mensagens é usado para adicionar mais informação de depuração à mensagem de erro, tal como um traço da pilha. Tal informação não pode ser obtida após o retorno de `lua_pcall`, pois nesse ponto a pilha foi desenrolada.

A função `lua_pcall` retorna um dos seguintes códigos (definidos em `lua.h`):

- **`LUA_OK` (0):** sucesso.
- **`LUA_ERRRUN`:** um erro de tempo de execução.
- **`LUA_ERRMEM`:** erro de alocação de memória. Para tais erros, Lua não chama o tratador de mensagens.
- **`LUA_ERRERR`:** erro ao executar o tratador de mensagens.
- **`LUA_ERRGCMM`:** erro ao executar um metamétodo `__gc`. (Este erro tipicamente não tem relação com a função sendo chamada. Ele é gerado pelo coletor de lixo.)

lua_pcallk

```
int lua_pcallk (lua_State *L,                                  [-(nargs + 1), +(nresults|1), –]
                int nargs,
                int nresults,
                int errfunc,
                int ctx,
                lua_CFunction k);
```

Esta função comporta-se exatamente como `lua_pcall`, mas permite a função chamada ceder (veja §4.7).

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, e]
```

Empilha um novo fecho C na pilha.

Quando uma função C é criada, é possível associar alguns valores a ela, criando assim um fecho C (veja §4.4); esses valores são então acessíveis à função sempre que ela é chamada. Para associar valores com uma função C, primeiro esses valores devem ser colocados na pilha (quando há múltiplos valores, o primeiro valor é colocado primeiro). Em seguida `lua_pushcclosure` é chamada para criar e colocar a função C na pilha, com o argumento `n` dizendo quantos valores devem ser associados com a função. `lua_pushcclosure` também retira valores da pilha.

O valor máximo para `n` é 255.

Quando `n` é zero, esta função cria uma *função C leve*, que é apenas um ponteiro para a função C. Nesse caso, ela nunca lança um erro de memória.

lua_pushcfunction

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

Coloca 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 a ser registrada em Lua deve seguir o protocolo correto para receber seus parâmetros e retornar seus resultados (veja `lua_CFunction`).

`lua_pushcfunction` é definida como uma macro:

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

Note que `f` é usado duas vezes.

lua_pushfstring

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

Coloca na pilha uma cadeia formatada e retorna um ponteiro para essa cadeia. É similar a ANSI C function `sprintf`, mas tem algumas diferenças importantes:

- Você não precisa alocar espaço para o resultado: o resultado é uma cadeia Lua 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*, comprimentos, 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 byte).

lua_pushglobaltable

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

Coloca o ambiente global na pilha.

lua_pushinteger

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

Coloca um número com valor `n` na pilha.

lua_pushlightuserdata

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

Colocar um userdata leve na pilha.

Um userdata representa valores C em Lua. Um *userdata leve* representa um ponteiro, um `void*`. Ele é um valor (como um número): você não o cria, ele não possui metatabela individual, e não é coletado (pois nunca foi criado). Um userdata leve é igual a "qualquer" userdata leve com o mesmo endereço C.

lua_pushliteral

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

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

lua_pushlstring

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

Coloca a cadeia apontada por `s` com tamanho `len` na pilha. Lua faz (ou reusa) uma cópia interna da cadeia fornecida, de modo que a memória de `s` pode ser liberada ou reusada imediatamente após a função retornar. A cadeia pode conter qualquer dado binário, incluindo zeros dentro dela.

Retorna um ponteiro para a cópia interna da cadeia.

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

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

Coloca a cadeia terminada por zero apontada por `s` na pilha. Lua faz (ou reusa) uma cópia interna da cadeia fornecida, de modo que a memória apontada por `s` pode ser liberada ou reusada imediatamente após a função retornar.

Retorna um ponteiro para a cópia interna da cadeia.

Se `s` é `NULL`, empilha **nil** e retorna `NULL`.

lua_pushthread

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

Coloca o fluxo de execução representado por `L` na pilha. Retorna 1 se esse fluxo é o fluxo principal de seu estado.

lua_pushunsigned

```
void lua_pushunsigned (lua_State *L, lua_Unsigned n);
```
[-0, +1, –]

Coloca um número com valor `n` na pilha.

lua_pushvalue

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

Coloca uma cópia do elemento no índice fornecido na pilha.

lua_pushvfstring

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

Equivalente a `lua_pushfstring`, exceto que ela 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 `index1` e `index2` são iguais primitivamente (isto é, sem chamar metamétodos). Caso contrário retorna 0. Também retorna 0 se algum dos índices não é 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 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 a tabela no índice fornecido. O acesso é primitivo; isto é, não invoca metamétodos.

lua_rawgetp

```
void lua_rawgetp (lua_State *L, int index, const void *p);
```
[-0, +1, –]

Coloca na pilha o valor `t[k]`, onde `t` é a tabela no índice fornecido e `k` é o ponteiro `p` representado como um userdata leve. O acesso é primitivo; isto é, não invoca metamétodos.

lua_rawlen

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

Retorna o "comprimento" primitivo do valor no índice fornecido: para cadeias, isso é o comprimento da cadeia; para tabelas, isso é o resultado do operador de comprimento ('#') sem metamétodos; para userdatas, isso é o tamanho do bloco de memória alocado para o userdata; para outros valores, é 0.

lua_rawset

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

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

lua_rawseti

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

Faz o equivalente de `t[n] = v`, onde `t` é a tabela no índice 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_rawsetp

```
void lua_rawsetp (lua_State *L, int index, const void *p);
```
[-1, +0, *e*]

Faz o equivalente de `t[k] = v`, onde `t` é a tabela no índice fornecido, `k` é o ponteiro `p` representado como um userdata leve, 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 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, const char *name, lua_CFunction f);
```
[-0, +0, *e*]

Estabelece a função C `f` como o novo valor da global `name`. Ela 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 desse índice para preencher o buraco. Esta função não pode ser chamada com um pseudo-índice, pois um pseudo-índice não é uma posição na pilha de verdade.

lua_replace

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

u

Move o elemento no topo para o índice válido fornecido sem deslocar nenhum elemento (substituindo portanto o valor no topo da pilha), e então desempilha o elemento no topo.

lua_resume

```
int lua_resume (lua_State *L, lua_State *from, int nargs);
```
[-?, +?, –]

Começa e retoma uma co-rotina em um dado fluxo de execução.

Para começar uma co-rotina, você deve colocar na pilha do fluxo de execução a função principal mais quaisquer argumentos; em seguida você chama `lua_resume`, com `nargs` sendo o número de argumentos. Essa chamada retorna quando a co-rotina suspende ou termina sua execução. Quando ela retorna, a pilha contém todos os valores passados para `lua_yield`, ou todos os valores retornados pela função do corpo. `lua_resume` retorna `LUA_YIELD` se a co-rotina cede, `LUA_OK` se a co-rotina termina sua execução sem erros, ou um código de erro em caso de erros (veja `lua_pcall`).

Em caso de erros, a pilha não é desenrolada, de modo que você pode usar a API de depuração sobre ela. A mensagem de erro está no topo da pilha.

Para retomar uma co-rotina, você remove quaisquer resultados da última `lua_yield`, coloca na sua pilha somente os valores a serem passados como resultados de `yield`, e então chama `lua_resume`.

O parâmetro `from` representa a co-rotina que está retomando `L`. Se não há tal co-rotina, esse parâmetro pode ser `NULL`.

lua_setallocf

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

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

lua_setfield

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

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

Esta função retira o valor do topo da pilha. Como em Lua, esta função pode disparar um metamétodo para o evento "newindex" (veja §2.4).

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`.

lua_setmetatable

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

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

lua_settable

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

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

Esta função desempilha tanto a chave quanto o valor da pilha. Como em Lua, esta função pode disparar um metamétodo para o evento "newindex" (veja §2.4).

lua_settop

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

Aceita qualquer índice, ou 0, e estabelece esse 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_setuservalue

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

Desempilha uma tabela ou **nil** da pilha e a estabelece como o novo valor associado ao userdata no índice fornecido.

lua_State

```
typedef struct lua_State lua_State;
```

Uma estrutura opaca que aponta para um fluxo de execução e indiretamente (através do fluxo) para o estado inteiro de uma interpretador Lua. A biblioteca Lua é totalmente reentrante: ela não possui variáveis globais. Toda informação sobre um estado é acessível através desta estrutura.

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

lua_status

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

Retorna o estado do fluxo de execução `L`.

O estado pode ser 0 (`LUA_OK`) para um fluxo normal, um código de erro se o fluxo terminou a execução de um `lua_resume` com um erro, ou `LUA_YIELD` se o fluxo está suspenso.

Você pode chamar funções somente em fluxos de execução com estado `LUA_OK`. Você pode retomar fluxos com estado `LUA_OK` (para começar uma nova co-rotina) ou `LUA_YIELD` (para retomar uma co-rotina).

lua_toboolean

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

Converte um valor Lua no índice fornecido para um valor booleano C (0 ou 1). Como todos os testes em Lua, `lua_toboolean` retorna verdadeiro para qualquer valor Lua diferente de **false** e **nil**; caso contrário ela retorna falso. (Se você quiser aceitar somente valores booleanos de verdade, 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 fornecido para uma função C. Esse valor deve ser uma função C; caso contrário, retorna `NULL`.

lua_tointeger

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

Equivalente a `lua_tointegerx` com `isnum` igual a `NULL`.

lua_tointegerx

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

Converte o valor Lua no índice 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 (veja §3.4.2); caso contrário, `lua_tointegerx` retorna 0.

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

Se `isnum` não é `NULL`, seu referente recebe um valor booleano que indica se a operação foi bem sucedida.

lua_tolstring

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

Converte o valor Lua no índice fornecido para uma cadeia C. Se `len` não é `NULL`, também atribui a `*len` o comprimento da cadeia. O valor Lua deve ser uma cadeia ou um número; caso contrário, a função retorna `NULL`. Se o valor é um número, então `lua_tolstring` também *muda o valor de fato na pilha para uma cadeia*. (Essa 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 dentro do estado Lua. Essa cadeia sempre tem um zero ('`\0`') após seu último caractere (como em C), mas pode conter outros zeros em seu corpo. Como Lua tem 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, –]

Equivalente a `lua_tonumberx` com `isnum` igual a `NULL`.

lua_tonumberx

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

Converte o valor Lua no índice fornecido para o tipo C `lua_Number` (veja `lua_Number`). O valor Lua deve ser um número ou uma cadeia que pode ser convertida para um número (veja §3.4.2); caso contrário, `lua_tonumberx` retorna 0.

Se `isnum` não é `NULL`, seu referente recebe um valor booleano que indica se a operação foi bem sucedida.

lua_topointer

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

Converte o valor no índice fornecido para um ponteiro C genérico (`void*`). O valor pode ser um userdata, uma tabela, um fluxo de execução, ou uma função; caso contrário, `lua_topointer` retorna `NULL`. Objetos

diferentes irão fornecer ponteiros diferentes. Não há maneira de converter o ponteiro de volta para seu valor original.

Tipicamente esta função é usada somente para informação de depuração.

lua_tostring

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

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

lua_tothread

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

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

lua_tounsigned

```
lua_Unsigned lua_tounsigned (lua_State *L, int index);
```
[-0, +0, –]

Equivalente a `lua_tounsignedx` com `isnum` igual a `NULL`.

lua_tounsignedx

```
lua_Unsigned lua_tounsignedx (lua_State *L, int index, int *isnum);
```
[-0, +0, –]

Converte o valor Lua no índice fornecido para o tipo inteiro sem sinal `lua_Unsigned`. O valor Lua deve ser um número ou uma cadeia que pode ser convertida para um número. (veja §3.4.2); caso contrário, `lua_tounsignedx` retorna 0.

Se o número não é um inteiro, ele é truncado de alguma maneira não especificada. Se o número está fora do intervalo de valores representáveis, ele é normalizado para o resto de sua divisão por um a mais do que o valor representável máximo.

Se `isnum` não é `NULL`, seu referente recebe um valor booleano que indica se a operação foi bem sucedida.

lua_touserdata

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

Se o valor no índice fornecido é um userdata completo, retorna o endereço de seu bloco. Se o valor é um userdata leve, 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 válido fornecido, ou `LUA_TNONE` para um índice não-válido (porém aceitável). 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`, o qual deve ser um dos valores retornados por `lua_type`.

lua_Unsigned

```
typedef unsigned long lua_Unsigned;
```

O tipo usado pela API de Lua para representar valores inteiros sem sinal. Ele deve ter no mínimo 32 bits.

Por padrão ele é um `unsigned int` ou um `unsigned long`, que podem ambos guardar valores de 32 bits.

lua_upvalueindex

```
int lua_upvalueindex (int i);
```
[-0, +0, –]

Retorna o pseudo-índice que representa o `i`-ésimo upvalue da função que está executando (veja §4.4).

lua_version

```
const lua_Number *lua_version (lua_State *L);
```
[-0, +0, *v*]

Retorna o endereço do número da versão armazenado no núcleo de Lua. Quando chamada com um `lua_State` válido, retorna o endereço da versão usada para criar aquele estado. Quando chamada com `NULL`, retorna o endereço da versão executando a chamada.

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 sem erros; qualquer outro valor significa um erro e impede `lua_dump` de chamar a função de escrita novamente.

lua_xmove

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

Troca valores entre diferentes fluxos de execução do mesmo estado.

Esta função retira `n` valores da pilha `from`, e os coloca na pilha `to`.

lua_yield

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

Esta função é equivalente a `lua_yieldk`, mas ela não possui uma continuação (veja §4.7). Assim, quando o fluxo de execução recomeça, ela retorna para a função que chamou a função chamando `lua_yield`.

lua_yieldk

```
int lua_yieldk (lua_State *L, int nresults, int ctx, lua_CFunction k);
```
[-?, +?, –]

Cede uma co-rotina.

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

```
return lua_yieldk (L, n, i, k);
```

Quando uma função C chama `lua_yieldk` dessa maneira, a co-rotina que está executando suspende sua execução, e a chamada a `lua_resume` que iniciou essa co-rotina retorna. O parâmetro `nresults` é o número de valores da pilha que são passados como resultados para `lua_resume`.

Quando a co-rotina é retomada novamente, Lua chama a função de continuação fornecida `k` para continuar a execução da função C que cedeu (veja §4.7). Essa função de continuação recebe a mesma pilha da função anterior, com os resultados removidos e substituídos pelos argumentos passados para `lua_resume`. Além disso, a função de continuação pode acessar o valor `ctx` chamando `lua_getctx`.

4.9 – A Interface de Depuração

Lua não possui facilidades de depuração pré-definidas. Ao invés disso, ela oferece uma interface especial por meio de funções e *ganchos*. Essa interface permite a construção de diferentes tipos de depuradores, analisadores dinâmicos de programas, e outras ferramentas que precisam 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 linedefined;            /* (S) */
  int lastlinedefined;        /* (S) */
  unsigned char nups;         /* (u) número de upvalues */
  unsigned char nparams;      /* (u) número de parâmetros */
  char isvararg;              /* (u) */
  char istailcall;            /* (t) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* parte privada */
  outros campos
} lua_Debug;
```

Uma estrutura usada para guardar pedaços diferentes de informação sobre uma função ou um registro de ativação. `lua_getstack` preenche somente a parte privada dessa 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`:** a fonte do trecho que criou a função. Se `source` começa com um '`@`', significa que a função foi definida em um arquivo onde o nome do arquivo vem depois do '`@`'. Se `source` começa com um '`=`', o resto de seu conteúdo descreve a fonte de uma maneira que depende do usuário. Caso contrário, a função foi definida em uma cadeia onde `source` é essa cadeia.
- **`short_src`:** uma versão "imprimível" 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.
- **`currentline`:** a linha corrente onde a função dada está executando. Quando nenhuma informação sobre a linha está disponível, `currentline` recebe -1.
- **`name`:** um nome razoável para a função dada. Como 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 ela não consegue encontrar um nome, então `name` recebe `NULL`.
- **`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 nenhum outra opção parece se aplicar.)
- **`istailcall`:** verdadeiro se esta invocação de função foi chamada por uma chamada final. Nesse caso, o chamador deste nível não está na pilha.
- **`nups`:** o número de upvalues da função.
- **`nparams`:** o número de parâmetros fixo da função (sempre 0 para funções C).
- **`isvararg`:** verdadeiro se a função é uma função vararg. (sempre verdadeiro para funções C).

lua_gethook

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

Retorna a função de gancho corrente.

lua_gethookcount

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

Retorna a contagem do gancho corrente.

lua_gethookmask

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

Retorna a máscara do gancho corrente.

lua_getinfo

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

Obtém informação sobre uma função ou 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 fornecido como argumento para um gancho (veja `lua_Hook`).

Para obter informação sobre uma função você a coloca na pilha e inicia a cadeia `what` com o caractere '>'. (Nesse caso, `lua_getinfo` desempilha a função do topo da pilha.) Por exemplo, para saber em qual linha uma função `f` foi definida, você pode escrever o código a seguir:

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

Cada caractere na cadeia `what` seleciona alguns campos da estrutura `ar` a serem preenchidos ou um valor a ser colocado 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`;
- **'`t`':** preenche o campo `istailcall`;
- **'`u`':** preenche os campos `nups`, `nparams`, e `isvararg`;
- **'`f`':** coloca na pilha a função que está executando no nível fornecido;
- **'`L`':** coloca na pilha uma tabela cujos índices são os números 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, um 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 dado registro de ativação ou de uma dada função.

No primeiro caso, o parâmetro `ar` dever ser um registro de ativação válido que foi preenchido por uma chamada anterior a `lua_getstack` ou fornecido como um argumento para um gancho (veja `lua_Hook`). O índice `n` seleciona qual variável local inspecionar; veja `debug.getlocal` para detalhes sobre índices e nomes de variáveis.

`lua_getlocal` coloca o valor da variável na pilha e retorna o nome dela.

No segundo caso, `ar` deve ser `NULL` e a função a ser inspecionada deve estar no topo da pilha. Nesse caso, somente parâmetros de funções Lua são visíveis (pois não há informação sobre quais variáveis estão ativas) e nenhum valor é colocado na pilha.

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, enquanto que o nível *n+1* é a função que chamou o nível *n* (exceto para chamadas finais, que não contam com a pilha). Quando não há erros, `lua_getstack` retorna 1; quando chamada com um nível maior do que a profundidade da pilha, 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 as variáveis locais externas que a função usa, e que são consequentemente incluídas em seu fecho.) `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 particular, pois eles estão ativos durante a função inteira. Assim, 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, o campo `event` de seu argumento `ar` recebe o evento específico que disparou o gancho. Lua identifica esses eventos com as seguintes constantes: `LUA_HOOKCALL`, `LUA_HOOKRET`, `LUA_HOOKTAILCALL`, `LUA_HOOKLINE`, e `LUA_HOOKCOUNT`. Além disso, para eventos de linha, o campo `currentline` também é determinado. Para obter o valor de qualquer outro campo em `ar`, o gancho deve chamar `lua_getinfo`.

Para eventos de chamada, `event` pode ser `LUA_HOOKCALL`, o valor normal, ou `LUA_HOOKTAILCALL`, para uma recursão final; neste caso, não haverá nenhum evento de retorno correspondente.

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

Funções de gancho não podem ter continuações, isto é, elas não podem chamar `lua_yieldk`, `lua_pcallk`, ou `lua_callk` com um `k` não nulo.

Funções de gancho podem ceder sob as seguintes condições: Somente eventos de contagem e de linha podem ceder e eles não podem produzir nenhum valor; para ceder uma função de gancho deve terminar sua execução chamando `lua_yield` com `nresults` igual a zero.

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 a função de gancho. `mask` especifica sobre quais eventos o gancho será chamado: ele é formado por uma conjunção bit a bit das contantes `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:** 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:** é chamado quando o interpretador retorna de uma função. O gancho é chamado logo antes de Lua sair da função. Não há uma maneira padrão de acessar os valores retornados pela função.
- **O gancho de linha:** é chamado quando o interpretador está para começar a execução de uma nova linha de código, ou quando ele faz um desvio para trás no código (mesmo que seja para a mesma linha). (Este evento somente acontece quando Lua está executando uma função Lua.)
- **O gancho de contagem:** é chamado após o interpretador executar cada uma das instruções `count`. (Este evento somente acontece 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 dado registro de ativação. Os parâmetros `ar` e `n` são como em `lua_getlocal` (veja `lua_getlocal`). `lua_setlocal` atribui o valor no topo da pilha à variável e retorna o nome dela. 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. Atribui o valor no topo da pilha ao upvalue e retorna o nome dele. Também desempilha o valor da pilha. Os parâmetros `funcindex` e `n` são como em `lua_getupvalue` (veja `lua_getupvalue`).

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

lua_upvalueid

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

Retorna um identificador único para o upvalue de número `n` do fecho no índice `funcindex`. Os parâmetros `funcindex` e `n` são como em `lua_getupvalue` (veja `lua_getupvalue`) (mas `n` não pode ser maior do que o número de upvalues).

Esses identificadores únicos permitem ao programa verificar se fechos diferentes compartilham upvalues. Fechos Lua que compartilham um upvalue (isto é, que acessam a mesma variável local externa) retornarão identificadores idênticos para esses índices de upvalue.

lua_upvaluejoin

```
void lua_upvaluejoin (lua_State *L, int funcindex1, int n1,         [-0, +0, –]
                                    int funcindex2, int n2);
```

Faz o `n1`-ésimo upvalue do fecho Lua no índice `funcindex1` se referir ao `n2`-ésimo upvalue do fecho Lua no índice `funcindex2`.

5 – A Biblioteca Auxiliar

A *biblioteca auxiliar* oferece várias funções convenientes para interfacear C com Lua. Enquanto a API básica oferece funções primitivas para todas as interações entre C e Lua, a biblioteca auxiliar oferece

funções de mais alto nível para algumas tarefas comuns.

Todas as funções e tipos da biblioteca auxiliar estão definidos no arquivo de cabeçalho `lauxlib.h` e possuem um prefixo `luaL_`.

Todas as funções na biblioteca auxiliar são construídas em cima da API básica, e portanto elas não oferecem nada que não possa ser feito com essa API. Apesar disso, o uso da biblioteca auxiliar garante mais consistência para seu código.

Várias funções na biblioteca auxiliar usam internamente alguns espaços extras de pilha. Quando uma função na biblioteca auxiliar usa menos do que cinco espaços, ela não verifica o tamanho da pilha; ela simplesmente assume que há espaços suficientes.

Várias funções na biblioteca auxiliar são usadas para verificar argumentos de funções C. Como a mensagem de erro é formatada para argumentos (e.g., "`argumento ruim #1`"), você não deve usar essas funções para outros valores da pilha.

Funções chamadas `luaL_check*` sempre lançam um erro se a verificação não é satisfeita.

5.1 – Funções e Tipos

Aqui listamos todas as funções e tipos da biblioteca auxiliar em ordem alfabética.

luaL_addchar

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

Adiciona o byte `c` ao buffer `B` (veja `luaL_Buffer`).

luaL_addlstring

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

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

luaL_addsize

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

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

luaL_addstring

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

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

luaL_addvalue

```
void luaL_addvalue (luaL_Buffer *B);
```
[-1, +?, *e*]

Adiciona o valor no topo da pilha ao buffer `B` (veja `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, o qual é o valor a ser adicionado ao buffer.

luaL_argcheck

```
void luaL_argcheck (lua_State *L,
                    int cond,
```
[-0, +0, *v*]

```
                    int arg,
                    const char *extramsg);
```

Verifica se `cond` é verdadeira. Se não, lança um erro com uma mensagem padrão.

luaL_argerror

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

Lança um erro com uma mensagem de erro padrão que inclui `extramsg` como um comentário.

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*.

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

- Primeiro declare uma variável `b` do tipo `luaL_Buffer`.
- Em seguida inicialize-a com uma chamada `luaL_buffinit(L, &b)`.
- Depois adicione pedaços da cadeia ao buffer chamando qualquer uma das funções `luaL_add*`.
- Finalmente chame `luaL_pushresult(&b)`. Essa chamada deixa a cadeia final no topo da pilha.

Se você sabe de antemão o tamanho total da cadeia resultante, você pode usar o buffer assim:

- Primeiro declare uma variável `b` do tipo `luaL_Buffer`.
- Em seguida inicialize-a e pré-aloque um espaço de tamanho `sz` com uma chamada `luaL_buffinitsize(L, &b, sz)`.
- Depois copie a cadeia para esse espaço.
- Termine chamando `luaL_pushresultsize(&b, sz)`, onde `sz` é o tamanho total da cadeia resultante copiada para esse espaço.

Durante sua operação normal, um buffer de cadeia usa um número variável de espaços de pilha. Assim, ao usar um buffer, você não pode assumir que você sabe onde o topo da pilha está. Você pode usar a pilha entre chamadas sucessivas às operações de buffer desde que esse uso seja balanceado; isto é, quando você chama uma operação de buffer, a chamada está no mesmo nível 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 volta ao nível que ela estava 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, –]

Inicializa um buffer `B`. Esta função não alocar nenhum espaço; o buffer deve ser declarado como uma variável (veja `luaL_Buffer`).

luaL_buffinitsize

```
char *luaL_buffinitsize (lua_State *L, luaL_Buffer *B, size_t sz);
```
[-?, +?, *e*]

Equivalente à sequência `luaL_buffinit`, `luaL_prepbuffsize`.

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` tem uma metatabela e essa metatabela tem um campo `e`, esta função chama

esse campo passando o objeto como seu único argumento. Nesse caso esta função retorna verdadeiro e coloca na pilha o valor retornado pela chamada. Se não há metatabela ou metamétodo, esta função retorna falso (sem colocar nenhum valor na pilha).

luaL_checkany

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

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

luaL_checkint

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

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

luaL_checkinteger

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

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

luaL_checklong

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

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

luaL_checklstring

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

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

Esta função usa `lua_tolstring` para obter seu resultado, assim todas as conversões e cuidados dessa função se aplicam aqui.

luaL_checknumber

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

Verifica se o argumento `arg` da função é um número e retorna esse número.

luaL_checkoption

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

Verifica se o argumento `arg` da função é uma cadeia e procura por essa cadeia no array `lst` (que deve ser terminado por NULL). Retorna o índice no array onde a cadeia foi encontrada. Lança um erro se o argumento não é uma cadeia ou se a cadeia não pode ser encontrada.

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

Esta á 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, lançando um erro se a pilha não pode aumentar para esse tamanho. `msg` é um texto adicional a ser colocado na mensagem de erro (ou `NULL` para nenhum texto adicional).

luaL_checkstring

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

Verifica se o argumento `arg` da função é uma cadeia e retorna essa cadeia.

Esta função usa `lua_tolstring` para obter seu resultado, assim todas as conversões e cuidados relacionados a essa função se aplicam aqui.

luaL_checktype

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

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

luaL_checkudata

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

Verifica se o argumento `arg` da função é um userdata do tipo `tname` (veja `luaL_newmetatable`) e retorna o endereço do userdata (veja `lua_touserdata`).

luaL_checkunsigned

```
lua_Unsigned luaL_checkunsigned (lua_State *L, int arg);
```
[-0, +0, *v*]

Verifica se o argumento `arg` da função é um número e retorna esse número convertido para um `lua_Unsigned`.

luaL_checkversion

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

Verifica se o núcleo executando a chamada, o núcleo que criou o estado Lua, e o código fazendo a chamada estão todos usando a mesma versão de Lua. Também verifica se o núcleo executando a chamada e o núcleo que criou o estado Lua estão usando o mesmo espaço de endereços.

luaL_dofile

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

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

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

Retorna falso se não há erros ou verdadeiro em caso de erros.

luaL_dostring

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

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

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

Retorna false se não há erros ou verdadeiro em caso de erros.

luaL_error

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

Lança 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 começo da mensagem de erro o nome do arquivo e o número da linha onde o erro ocorreu, se essa informação está disponível.

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

luaL_execresult

```
int luaL_execresult (lua_State *L, int stat);
```
[-0, +3, *e*]

Esta função produz os valores retornados por funções relacionadas a processos na biblioteca padrão (`os.execute` e `io.close`).

luaL_fileresult

```
int luaL_fileresult (lua_State *L, int stat, const char *fname);
```
[-0, +(1|3), *e*]

Esta função produz os valores retornados por funções relacionadas a arquivos na biblioteca padrão (`io.open`, `os.rename`, `file:seek`, etc.).

luaL_getmetafield

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

Coloca na pilha o campo `e` da metatabela do objeto no índice `obj`. Se o objeto não tem uma metatabela, ou se a metatabela não tem esse campo, retorna falso 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 (veja `luaL_newmetatable`).

luaL_getsubtable

```
int luaL_getsubtable (lua_State *L, int idx, const char *fname);
```
[-0, +1, *e*]

Garante que o valor `t[fname]`, onde `t` é o valor no índice `idx`, é uma tabela, e coloca essa tabela na pilha. Retorna verdadeiro se encontra uma tabela anterior lá e falso se cria uma nova tabela.

luaL_gsub

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

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

luaL_len

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

Retorna o "comprimento" do valor no índice fornecido como um número; é equivalente ao operador '`#`' em Lua (veja §3.4.6). Lança um erro se o resultado da operação não é um número. (Esse caso somente pode acontecer através de metamétodos.)

luaL_loadbuffer

```
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
```
[-0, +1, –]

```
                      size_t sz,
                      const char *name);
```

Equivalente a `luaL_loadbufferx` com `mode` igual a `NULL`.

luaL_loadbufferx

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

Carrega um buffer como um trecho 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 informação de depuração e mensagens de erro. A cadeia `mode` funciona como na função `lua_load`.

luaL_loadfile

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

Equivalente a `luaL_loadfilex` com `mode` igual a `NULL`.

luaL_loadfilex

```
int luaL_loadfilex (lua_State *L, const char *filename,                [-0, +1, e]
                                  const char *mode);
```

Carrega um arquivo como um trecho 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 um `#`.

A cadeia `mode` funciona como na função `lua_load`.

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 ou o arquivo tem um modo errado.

Como `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, –]
```

Carrega uma cadeia como um trecho 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`.

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

luaL_newlib

```
void luaL_newlib (lua_State *L, const luaL_Reg *l);                    [-0, +1, e]
```

Cria uma nova tabela e registra lá as funções na lista `l`. Ela é implementada como a seguinte macro:

```
(luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))
```

luaL_newlibtable

```
void luaL_newlibtable (lua_State *L, const luaL_Reg l[]);              [-0, +1, e]
```

Cria uma nova tabela com um tamanho otimizado para armazenar todas as entradas no array `l` (mas não

as armazena de fato). É planejada para ser usada em conjunção com `luaL_setfuncs` (veja `luaL_newlib`).

É implementada como uma macro. O array `l` deve ser o array de fato, não um ponteiro para ele.

luaL_newmetatable

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

Se o registro já tem a chave `tname`, retorna 0. Caso contrário, cria uma nova tabela para ser usada como uma metatabela para userdata, adiciona-a 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 um alocador baseado na função `realloc` de C padrão e então estabelece uma função de pânico (veja §4.6) que imprime uma mensagem de erro para a saída de erro padrão em caso de erros fatais.

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

luaL_openlibs

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

Abre todas as bibliotecas Lua padrão no estado fornecido.

luaL_optint

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

Se o argumento `arg` da função é um número, retorna esse número convertido para um `int`. Se esse argumento está ausente ou é **nil**, retorna `d`. Caso contrário, lança um erro.

luaL_optinteger

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

Se o argumento `arg` da função é um número, retorna esse número convertido para um `lua_Integer`. Se esse argumento está ausente ou é **nil**, retorna `d`. Caso contrário, lança um erro.

luaL_optlong

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

Se o argumento `arg` da função é um número, retorna esse número convertido para um `long`. Se esse argumento está ausente ou é **nil**, retorna `d`. Caso contrário, lança um erro.

luaL_optlstring

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

Se o argumento `arg` da função é uma cadeia, retorna essa cadeia. Se esse argumento está ausente ou é **nil**, retorna `d`. Caso contrário, lança 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 arg, lua_Number d);
```
[-0, +0, *v*]

Se o argumento `arg` da função é um número, retorna esse número. Se esse argumento está ausente ou é **nil**, retorna `d`. Caso contrário, lança um erro.

luaL_optstring

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

Se o argumento `arg` da função é uma cadeia, retorna essa cadeia. Se esse argumento está ausente ou é **nil**, retorna `d`. Caso contrário, lança um erro.

luaL_optunsigned

```
lua_Unsigned luaL_optunsigned (lua_State *L,
                               int arg,
                               lua_Unsigned u);
```
[-0, +0, *v*]

Se o argumento `arg` da função é um número, retorna esse número convertido para um `lua_Unsigned`. Se esse argumento está ausente ou é **nil**, retorna `u`. Caso contrário, lança um erro.

luaL_prepbuffer

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

Equivalente a `luaL_prepbuffsize` com o tamanho pré-definido `LUAL_BUFFERSIZE`.

luaL_prepbuffsize

```
char *luaL_prepbuffsize (luaL_Buffer *B, size_t sz);
```
[-?, +?, *e*]

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

luaL_pushresult

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

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

luaL_pushresultsize

```
void luaL_pushresultsize (luaL_Buffer *B, size_t sz);
```
[-?, +1, *e*]

Equivalente à sequência `luaL_addsize`, `luaL_pushresult`.

luaL_ref

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

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. Contanto 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 pela 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 registradas por `luaL_setfuncs`. `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_requiref

```
void luaL_requiref (lua_State *L, const char *modname,                [-0, +1, e]
                    lua_CFunction openf, int glb);
```

Chama a função `openf` com a cadeia `modname` como um argumento e atribui a `package.loaded[modname]` o resultado da chamada, como se essa função tivesse sido chamada através de `require`.

Se `glb` é verdadeiro, também armazena o resultado na global `modname`.

Deixa uma cópia desse resultado na pilha.

luaL_setfuncs

```
void luaL_setfuncs (lua_State *L, const luaL_Reg *l, int nup);       [-nup, +0, e]
```

Registra todas as funções no array `l` (veja `luaL_Reg`) na tabela no topo da pilha (abaixo de upvalues opcionais, veja a seguir).

Quando `nup` não é zero, todas as funções são criadas compartilhando `nup` upvalues, os quais devem ser colocados previamente no topo da pilha da tabela da biblioteca. Esses valores são retirados da pilha após o registro.

luaL_setmetatable

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

Estabelece a metatabela do objeto no topo da pilha como a metatabela associada com o nome `tname` no registro (veja `luaL_newmetatable`).

luaL_testudata

```
void *luaL_testudata (lua_State *L, int arg, const char *tname);      [-0, +0, e]
```

Esta função funciona como `luaL_checkudata`, exceto pelo fato de que, quando o teste falha, ela retorna `NULL` ao invés de lançar um erro.

luaL_tolstring

```
const char *luaL_tolstring (lua_State *L, int idx, size_t *len);      [-0, +1, e]
```

Converte qualquer valor Lua no índice fornecido para uma cadeia C em um formato razoável. A cadeia resultante é colocada na pilha e também retornada pela função. Se `len` não é `NULL`, a função também atribui a `*len` o comprimento da cadeia.

Se o valor possui uma metatabela com um campo `"__tostring"`, então `luaL_tolstring` chama o metamétodo correspondente com o valor como argumento, e usa o resultado da chamada como resultado dela.

luaL_traceback

```
void luaL_traceback (lua_State *L, lua_State *L1, const char *msg,   [-0, +1, e]
```

```
                        int level);
```

Cria e coloca um traço na pilha `L1`. Se `msg` não é `NULL` ela é acrescentada ao início do traço. O parâmetro `level` diz em qual nível começar o traço.

luaL_typename

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

Retorna o nome do tipo do valor no índice fornecido.

luaL_unref

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

Libera a referência `ref` da tabela no índice `t` (veja `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, *e*]

Coloca na pilha uma cadeia identificando a posição corrente do controle no nível `lvl` na pilha de chamadas. Tipicamente essa cadeia possui o seguinte formato:

nometrecho`:`*linhacorrente*`:`

O Nível 0 é a função executando, o Nível 1 é a função que chamou a função executando, etc.

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

6 – 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., I/O); e outras poderiam ser implementadas em Lua mesmo, mas são bastante úteis ou possuem exigências de desempenho críticas que merecem uma implementação em C (e.g., `table.sort`).

Todas as bibliotecas são implementadas através da API C oficial e são oferecidas como módulos C separados. Atualmente, Lua possui as seguintes bibliotecas padrão:

- biblioteca básica (§6.1);
- biblioteca de co-rotinas (§6.2);
- biblioteca de pacotes (§6.3);
- manipulação de cadeias (§6.4);
- manipulação de tabelas (§6.5);
- funções matemáticas (§6.6) (sin, log, etc.);
- operações bit a bit (§6.7);
- entrada e saída (§6.8);
- facilidades do sistema operacional (§6.9);
- facilidades de depuração (§6.10).

Exceto pelas bibliotecas básica e de pacotes, cada biblioteca oferece 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. Alternativamente, o programa hospedeiro pode abri-las individualmente usando `luaL_requiref` para chamar `luaopen_base` (para a biblioteca básica),

`luaopen_package` (para a biblioteca de pacotes), `luaopen_coroutine` (para a biblioteca de co-rotinas), `luaopen_string` (para a biblioteca de cadeias), `luaopen_table` (para a biblioteca de tabelas), `luaopen_math` (para a biblioteca matemática), `luaopen_bit32` (para a biblioteca de bits), `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`.

6.1 – Funções Básicas

A biblioteca básica oferece funções essenciais para Lua. Se você não incluir esta biblioteca em sua aplicação, você deve verificar cuidadosamente se você precisa oferecer 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 interface genérica para o coletor de lixo. Ela realiza funções diferentes de acordo com o seu primeiro argumento, `opt`:

- **`"collect"`:** realiza um ciclo de coleta de lixo completo. Esta é a opção padrão.
- **`"stop"`:** para a execução automática do coletor de lixo. O coletor executará somente quando explicitamente invocado, até uma chamada para reiniciá-lo.
- **`"restart"`:** reinicia a execução automática do coletor de lixo.
- **`"count"`:** retorna a memória total em uso por Lua (em Kbytes) e uma segundo valor com a memória total em bytes módulo 1024. O primeiro valor tem uma parte fracionária, assim a seguinte igualdade é sempre verdadeira:

  ```
  k, b = collectgarbage("count")
  assert(k*1024 == math.floor(k)*1024 + b)
  ```

  (O segundo resultado é útil quando Lua é compilada com um tipo diferente de ponto flutuante para números.)

- **`"step"`:** realiza um passo de coleta de lixo. O passo "size" é controlado por `arg` (valores maiores significam passos maiores) de maneira não especificada. Se você quiser controlar o tamanho do passo você deve experimentalmente ajustar o valor de `arg`. Retorna **true** se o passo terminou um ciclo de coleta.
- **`"setpause"`:** estabelece `arg` como o novo valor da *pausa* do coletor (veja §2.5). Retorna o valor anterior da *pausa*.
- **`"setstepmul"`:** estabelece `arg` como o novo valor do *multiplicador de passo* do coletor (veja §2.5). Retorna o valor anterior do *passo*.
- **`"isrunning"`:** retorna um booleano que diz se o coletor está executando (i.e., não parado).
- **`"generational"`:** muda o coletor para o modo generacional. Esta é uma característica experimental (veja §2.5).
- **`"incremental"`:** muda o coletor para o modo incremental. Este é o modo padrão.

dofile ([filename])

Abre o arquivo indicado e executa seu conteúdo como um trecho 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 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 ao começo da mensagem, se a mensagem é uma cadeia. O argumento `level` especifica como obter a posição do erro. Quando level é 1 (o padrão), a posição do erro é onde a função `error` foi chamada. O level 2 aponta o erro para onde a função que chamou `error` foi chamada; e assim por diante. Passar um level 0 evita a adição de informação da posição do erro à mensagem.

_G

Uma variável global (não uma função) que guarda o ambiente global (veja §2.2). Lua em si não usa esta variável; mudar o valor dela não afeta nenhum ambiente, e vice-versa.

getmetatable (object)

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

ipairs (t)

Se `t` tem um metamétodo `__ipairs`, chama-o com `t` como argumento e retorna os primeiros três resultados da chamada.

Caso contrário, 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 (ld [, source [, mode [, env]]])

Carrega um trecho.

Se `ld` é uma cadeia, o trecho é essa cadeia. Se `ld` é uma função, `load` a chama repetidamente para obter os pedaços do trecho. Cada chamada a `ld` deve retornar uma cadeia que concatena com resultados anteriores. Um retorno de uma cadeia vazia, **nil**, ou nenhum valor sinaliza o fim do trecho.

Se não há erros sintáticos, retorna o trecho compilado como uma função; caso contrário, retorna **nil** mais a mensagem de erro.

Se a função resultante tem upvalues, o primeiro upvalue recebe o valor de `env`, se esse parâmetro é fornecido, ou o valor do ambiente global. (Quando você carrega um trecho principal, a função resultante sempre terá exatamente um upvalue, a variável `_ENV` (veja §2.2). Quando você carrega um trecho binário criado a partir de uma função (veja `string.dump`), a função resultante pode ter upvalues arbitrários.)

`source` é usada como a fonte do trecho para mensagens de erro e informação de depuração (veja §4.9). Quando ausente, o padrão é `ld`, se `ld` é uma cadeia, ou "`=(load)`" caso contrário.

A cadeia `mode` controla se o trecho pode ser texto ou binário (isto é, um trecho pré-compilado). Ela pode ser a cadeia "`b`" (somente trechos binários), "`t`" (somente trechos textuais), ou "`bt`" (tanto binário como texto). O padrão é "`bt`".

loadfile ([filename [, mode [, env]]])

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

next (table [, index])

Permite um programa percorrer todos os campos de um tabela. Seu primeiro argumento é uma tabela e seu segundo argumento é um índice nessa 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 uma tabela é vazia.

A ordem na qual os índices são enumerados não é especificada, *mesmo para índices numéricos*. (Para percorrer uma tabela em ordem numérica, use um **for** numérico.)

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)

Se `t` tem um metamétodo `__pairs`, chama-o com `t` como argumento e retorna os primeiros três resultados da chamada.

Caso contrário, 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 seu percorrimento.

pcall (f [, arg1, ···])

Chama a função `f` com os argumentos dados em *modo protegido*. Isso significa que qualquer erro dentro de `f` não é propagado; ao invés disso, `pcall` captura o erro e retorna um código de estado. Seu primeiro resultado é o código de estado (um booleano), o qual é verdadeiro se a chamada aconteceu sem erros. Em tal caso, `pcall` também retorna todos os resultados da chamada, após esse primeiro resultado. Em caso de qualquer erro, `pcall` retorna **false** mais a mensagem de erro.

print (···)

Recebe qualquer número de argumentos e imprime seus valores para `stdout`, usando a função `tostring` para converter cada argumento para uma cadeia. `print` não é projetada para saída formatada, mas somente como uma maneira rápida de mostrar um valor, por exemplo para depuração. Para um controle completo sobre a saída, use `string.format` e `io.write`.

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 um tabela; `index` pode ser qualquer valor.

rawlen (v)

Retorna o comprimento do objeto `v`, o qual deve ser uma tabela ou uma cadeia, sem invocar qualquer metamétodo. Retorna um número inteiro.

rawset (table, index, value)

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

Esta função retorna `table`.

select (index, ···)

Se `index` é um número, retorna todos os argumentos após o argumento número `index`; um número negativo indexa a partir do fim (-1 é o último argumento). Caso contrário, `index` deve ser a cadeia `"#"`, e `select` retorna o número total de argumentos extras que ela recebeu.

setmetatable (table, metatable)

Estabelece a metatabela para a tabela fornecida. (Você não pode modificar 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`", lança um erro.

Esta função retorna `table`.

tonumber (e [, base])

Quando chamada sem `base`, `tonumber` tenta converter seu argumento para um número. Se o argumento já é um número ou uma cadeia que pode ser convertida para um número (veja §3.4.2), então `tonumber` retorna esse número; caso contrário, retorna **nil**.

Quando chamada com `base`, então `e` deve ser uma cadeia a ser interpretada como um número inteiro nessa base. 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. Se a cadeia `e` não é um número válido na base fornecida, a função retorna **nil**.

tostring (v)

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

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

type (v)

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

_VERSION

Uma variável global (não uma função) que guarda uma cadeia contendo a versão do interpretador corrente. O conteúdo corrente desta variável é "`Lua 5.2`".

xpcall (f, msgh [, arg1, ···])

Esta função é similar a `pcall`, exceto pelo fato de que ela estabelece um novo tratador de mensagens `msgh`.

6.2 – Manipulação de Co-rotinas

As operações relacionadas a co-rotinas compreendem uma sub-biblioteca da biblioteca básica e vêm dentro da tabela `coroutine`. Veja §2.6 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 essa nova co-rotina, um objeto com tipo "`thread`".

coroutine.resume (co [, val1, ···])

Começa ou continua a execução da co-rotina `co`. Da primeira vez que você retoma uma co-rotina, ela começa executando o corpo dela. Os valores `val1`, ... são passados como os argumentos para a função do corpo. Se a co-rotina cedeu, `resume` a recomeça; 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 pela função do corpo (se a co-rotina termina). Se há qualquer erro, `resume` retorna **false** mais a mensagem de erro.

coroutine.running ()

Retorna a co-rotina executando mais um booleano, verdadeiro quando a co-rotina executando é a principal.

coroutine.status (co)

Retorna o estado da co-rotina `co`, como uma cadeia: `"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 executar ainda; `"normal"` se a co-rotina está ativa mas não está executando (isto é, ela retomou outra co-rotina); e `"dead"` se a co-rotina finalizou a função do corpo dela, 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 retoma a co-rotina cada vez que ela é chamada. Quaisquer argumentos extras 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. Quaisquer argumentos para `yield` são passados como resultados extras para `resume`.

6.3 – Módulos

A biblioteca de pacotes oferece facilidades básicas para carregar módulos em Lua. Ela exporta uma função diretamente no ambiente global: `require`. Todo o resto é exportado em um tabela `package`.

require (modname)

Carrega o módulo fornecido. A função começa investigando a tabela `package.loaded` para determinar se `modname` já está carregado. Se está, então `require` retorna o valor armazenado em `package.loaded[modname]`. Caso contrário, tenta encontrar um *carregador* para o módulo.

Para encontrar um carregador, `require` é guiada pela sequência de `package.searchers`. Modificando essa sequência, podemos modificar como `require` procura por um módulo. A explicação a seguir é baseada na configuração padrão de `package.searchers`.

Primeiro `require` consulta `package.preload[modname]`. Se ela tem um valor, esse valor (que deve ser uma função) é o carregador. Caso contrário `require` procura por um carregador Lua usando o caminho armazenado em `package.path`. Se isso também falha, ela procura por um carregador C usando o caminho armazenado em `package.cpath`. Se isso também falha, ela tenta um carregador *tudo-em-um* (veja `package.searchers`).

Uma vez que um carregador é encontrado, `require` chama o carregador com dois argumentos: `modname` e um valor extra dependente de como ela obteve o carregador. (Se o carregador veio de um arquivo, esse valor extra é o nome do arquivo.) Se o carregador retorna qualquer valor diferente de nil, `require` atribui o valor retornado a `package.loaded[modname]`. Se o carregador não retorna um valor diferente de nil e não atribuiu nenhum valor a `package.loaded[modname]`, então `require` atribui **true** a essa entrada. Em todo caso, `require` retorna o valor final de `package.loaded[modname]`.

Se há qualquer erro ao carregar ou executar o módulo, ou se ela não encontrou nenhum carregador para o módulo, então `require` lança um erro.

package.config

Uma cadeia descrevendo algumas configurações de tempo de compilação para pacotes. Esta cadeia é

uma sequência de linhas:

- A primeira linha é a cadeia separadora de diretórios. O padrão é '`\`' para Windows e '`/`' para todos os outros sistemas.
- A segunda linha é o caractere que separa modelos em um caminho. O padrão é '`;`'.
- A terceira linha é a cadeia que marca os pontos de substituição em um modelo. O padrão é '`?`'.
- A quarta linha é uma cadeia que, em um caminho no Windows, é substituída pelo diretório do executável. O padrão é '`!`'.
- A quinta linha é uma marca para ignorar todo o texto antes dela ao construir o nome da função `luaopen_`. O padrão é '`-`'.

package.cpath

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

Lua inicializa o caminho C `package.cpath` da mesma maneira que inicializa o caminho Lua `package.path`, usando a variável de ambiente `LUA_CPATH_5_2` ou 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 estão carregados. Quando você requisita um módulo `modname` e `package.loaded[modname]` não é falso, `require` simplesmente retorna o valor armazenado lá.

Essa variável é somente uma referência para a tabela real; atribuições a essa variável não modificam a tabela usada por `require`.

package.loadlib (libname, funcname)

Dinamicamente liga o programa hospedeiro com a biblioteca C `libname`.

Se a função `funcname` é "`*`", então somente liga com a biblioteca, tornando os símbolos exportados pela biblioteca disponíveis para outras bibliotecas ligadas dinamicamente. Caso contrário, procura por uma função `funcname` dentro da biblioteca e retorna essa função como uma função C. Assim, `funcname` deve seguir o protótipo `lua_CFunction` (veja `lua_CFunction`).

Essa é uma função de baixo nível. Ela ignora completamente o sistema de pacotes e módulos. Ao contrário de `require`, ela não realiza nenhuma busca de caminho e não adiciona extensões automaticamente. `libname` deve ser o nome completo do arquivo da biblioteca C, incluindo se necessário um caminho e uma extensão. `funcname` deve ser o nome exato exportado pela biblioteca C (o qual pode depender do compilador e do ligador C usados).

Esta função não é suportada por C Padrão. Dessa forma, ela está disponível somente em algumas plataformas (Windows, Linux, Mac OS X, Solaris, BSD, além de outros sistemas Unix que suportam o padrão `dlfcn`).

package.path

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

Ao iniciar, Lua inicializa esta variável com o valor da variável de ambiente `LUA_PATH_5_2` ou da variável de ambiente `LUA_PATH` ou com um valor padrão definido em `luaconf.h`, se essas variáveis de ambiente não estão definidas. Qualquer "`;;`" no valor da variável de ambiente é substituído pelo caminho padrão.

package.preload

Uma tabela para guardar carregadores para módulos específicos (veja `require`).

Esta variável é somente uma referência para a tabela real; atribuições a esta variável não modificam a tabela usada por `require`.

package.searchers

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

Cada entrada nesta tabela é uma *função buscadora*. Ao procurar por um módulo, `require` chama cada uma dessas buscadoras em ordem ascendente, com o nome do módulo (o argumento fornecido para `require`) como seu único argumento. A função pode retornar outra função (o *carregador* do módulo) mais um valor extra que será passado para esse carregador, ou uma cadeia explicando por que ela não encontrou esse módulo (ou **nil** se ela não tem nada a dizer). Lua inicializa esta tabela com quatro funções buscadoras.

A primeira buscadora simplesmente procura por um carregador na tabela `package.preload`.

A segunda buscadora procura por um carregador como uma biblioteca Lua, usando o caminho armazenado em `package.path`. A busca é feita como descrito na função `package.searchpath`.

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

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

a 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, essa buscadora primeiro usa uma facilidade de ligação dinâmica para ligar a aplicação com a biblioteca. Em seguida ela tenta encontrar uma função C dentro da biblioteca a ser usada como carregador. O nome dessa 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 tem 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 buscadora tenta um *carregador tudo-em-um*. Ela busca o caminho C para uma biblioteca pela raiz do nome do módulo fornecido. Por exemplo, ao requisitar `a.b.c`, ela procurará por uma biblioteca C para `a`. Se encontrar, busca dentro dela por uma função de abertura para o submódulo; no nosso exemplo, essa seria `luaopen_a_b_c`. Com essa facilidade, um pacote pode empacotar vários submódulos C dentro de uma única biblioteca, com cada submódulo mantendo sua função de abertura original.

Todas as buscadoras exceto a primeira (preload) retorna como valor extra o nome do arquivo onde o módulo foi encontrado, como retornado por `package.searchpath`. A primeira buscadora não retorna valor extra.

package.searchpath (name, path [, sep [, rep]])

Procura pelo `name` fornecido no `path` fornecido.

Um caminho é uma cadeia contendo uma sequência de *modelos* separados por ponto-e-vírgula. Para cada modelo, a função substitui cada ponto de interrogação (se houver) no modelo por uma cópia de `name` onde todas as ocorrências de `sep` (um ponto, por padrão) foram substituídas por `rep` (o separador de diretórios do sistema, por padrão), e em seguida tenta abrir o nome do arquivo resultante.

Por exemplo, se o caminho é a cadeia

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

a busca pelo nome `foo.a` tentará abrir os arquivos `./foo/a.lua`, `./foo/a.lc`, e `/usr/local/foo/a/init.lua`, nessa ordem.

Retorna o nome resultante do primeiro arquivo que ela conseguiu abrir em modo de leitura (após fechar o arquivo), ou **nil** mais uma mensagem de erro se nada foi bem sucedido. (Essa mensagem de erro lista todos os nomes de arquivo que ela tentou abrir.)

6.4 – Manipulação de Cadeias

Esta biblioteca oferece funções genéricas para manipulação de cadeias, 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 0, como em C). Índices podem ser negativos e são interpretados como uma indexação de trás pra frente, a partir do fim da cadeia. Dessa forma, o último caractere está na posição -1, e assim por diante.

A biblioteca de cadeias oferece todas as suas funções dentro da tabela `string`. Ela também estabelece uma metatabela para cadeias onde o campo `__index` aponta para a tabela `string`. Logo, 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 cadeias assume codificações de caracteres de um byte.

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

Retorna os códigos numéricos internos dos caracteres `s[i]`, `s[i+1]`, ..., `s[j]`. O valor padrão para `i` é 1; o valor padrão para `j` é `i`. Esses índices são corrigidos seguindo as mesmas regras da função `string.sub`.

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 tem um código numérico interno igual a seu argumento correspondente.

Códigos numéricos não são necessariamente portáveis entre plataformas.

string.dump (function)

Retorna uma cadeia contendo uma representação binária da função fornecida, de modo que um `load` posterior sobre essa cadeia retorna uma cópia da função (mas com novos upvalues).

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

Procura pelo primeiro casamento de `pattern` na cadeia `s`. Se encontra um casamento, então `find` retorna os índices de `s` onde essa ocorrência começou e terminou; caso contrário, retorna **nil**. Um terceiro argumento numérico opcional `init` especifica onde começar a busca; o valor padrão dele é 1 e pode ser negativo. Um valor **true** para o quarto argumento opcional `plain` desabilita as facilidades de casamento de padrão, de modo que a função faz uma operação de "busca de subcadeia" simples, sem que os caracteres de `pattern` sejam considerados mágicos. No 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 também são retornados, após os dois índices.

string.format (formatstring, ···)

Retorna uma versão formatada de seu número variável de argumentos seguindo a descrição dada em seu primeiro argumento (que deve ser uma cadeia). A cadeia de formatação segue as mesmas regras de ANSI C function `sprintf`. As únicas diferenças são que as opções/modificadores `*`, `h`, `L`, `l`, `n`, e `p` não são suportadas e que há uma opção extra, `q`. A opção `q` formata uma cadeia entre aspas duplas, usando sequências de escape quando necessário para garantir que ela possa ser lida de volta de modo seguro pelo interpretador Lua. Por exemplo, a chamada

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

pode produzir a cadeia:

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

As opções `A` e `a` (quando disponíveis), `E`, `e`, `f`, `G`, e `g` esperam todas um número como argumento. As opções `c`, `d`, `i`, `o`, `u`, `X`, e `x` também esperam um número, mas o intervalo desse número pode ser limitado pela implementação C subjacente. Para as opções `o`, `u`, `X`, e `x`, o número não pode ser negativo. A opção `q` espera uma cadeia; a opção `s` espera uma cadeia sem zeros dentro dela. Se o argumento para a opção

`s` não é uma cadeia, ele é convertido para uma seguindo as mesmas regras de `tostring`.

string.gmatch (s, pattern)

Retorna uma função iteradora que, cada vez que é chamada, retorna as próximas capturas de `pattern` na cadeia `s`. Se `pattern` não especifica capturas, então o casamento inteiro é produzido a cada chamada.

Como um exemplo, o seguinte laço irá iterar sobre todas as palavras da cadeia `s`, imprimindo uma por linha:

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

O próximo exemplo coleta todos os pares `key=value` da cadeia fornecida dentro de uma tabela:

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

Para esta função, um circunflexo '^' no início do padrão não funciona como uma âncora, pois isso impediria a iteração.

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

Retorna uma cópia de `s` na qual todas (ou as primeiras `n`, se fornecido) ocorrências de `pattern` foram 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 casamentos que ocorreram. O nome `gsub` vem de *Global SUBstitution*.

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

Se `repl` é uma tabela, então a tabela é consultada a cada casamento, usando a primeira captura como a chave.

Se `repl` é uma função, então essa função é chamada toda vez que um casamento ocorre, com todas as subcadeias capturadas passadas como argumentos, em ordem.

Em todo caso, se o padrão não especifica capturas, então ela comporta-se como se o padrão inteiro estivesse dentro de uma captura.

Se o valor retornado pela consulta à tabela ou pela chamada de função é uma cadeia ou um número, então ele 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 load(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.2"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.2.tar.gz"
```

string.len (s)

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

string.lower (s)

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

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

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

string.rep (s, n [, sep])

Retorna uma cadeia que é a concatenação de `n` cópias da cadeia `s` separadas pela cadeia `sep`. O valor padrão para `sep` é a cadeia vazia (isto é, nenhum separador).

string.reverse (s)

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

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

Retorna a subcadeia de `s` que começa 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 vazia). Em particular, a chamada `string.sub(s,1,j)` retorna um prefixo `s` com comprimento `j`, e `string.sub(s, -i)` retorna um sufixo de `s` com comprimento `i`.

Se, após a tradução de índices negativos, `i` é menor do que 1, ele é corrigido para 1. Se `j` é maior do que o comprimento da cadeia, ele é corrigido para esse comprimento. Se, após essas correções, `i` é maior do que `j`, a função retorna a cadeia vazia.

string.upper (s)

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

6.4.1 – Padrões

Classes de Caracteres:

Uma *classe de caracteres* é usada para representar um conjunto de caracteres. As seguintes combinações são permitidas ao descrever 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.
- **`%g`:** representa todos os caracteres que podem ser impressos exceto o espaço.
- **`%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.
- **`%`*`x`*:** (onde *x* é qualquer caractere não-alfanumérico) representa o caractere *x*. Esta é a maneira padrão de escapar caracteres mágicos. Qualquer caractere de pontuação (mesmo os não mágicos) podem ser precedidos 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*. Um intervalo de caracteres pode ser especificado separando os caracteres das extremidades do intervalo, em ordem ascendente, 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 intervalos e classes não é definida. Assim, 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 letras simples (`%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 depende do idioma 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 caracteres simples, que casa qualquer caractere simples na classe;
- uma classe de caracteres simples seguida por '`*`', que casa 0 ou mais repetições de caracteres na classe. Esses itens de repetição sempre casarão a maior sequência possível;
- uma classe de caracteres simples seguida por '`+`', que casa 1 ou mais repetições de caracteres na classe. Esses itens de repetição sempre casarão a maior sequência possível;
- uma classe de caracteres simples seguida por '`-`', que também casa 0 ou mais repetições de caracteres na classe. Ao contrário de '`*`', esses itens de repetição sempre casarão a menor sequência possível;
- uma classe de caracteres simples seguida por '`?`', que casa 0 ou 1 ocorrência de um caractere na 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 os *x* e *y* são *balanceados*. Isso significa que, se você ler a cadeia da esquerda para a direita, contando *+1* para um *x* e *-1* para um *y*, o último *y* é o primeiro *y* onde o contador alcança 0. Por exemplo, o item `%b()` casa expressões com parênteses balanceados.
- `%f[`*set*`]`, um *padrão de fronteira*; tal item casa uma cadeia vazia em qualquer posição desde que o próximo caractere pertença a *set* e o caractere anterior não pertença a *set*. O conjunto *set* é interpretado como descrito anteriormente. O início e o fim da cadeia principal são tratados como se eles fossem o caractere '`\0`'.

Padrão:

Um *padrão* é uma sequência de itens de padrão. Um circunflexo '`^`' no início de um padrão ancora o casamento no início do texto principal. Um '`$`' no fim de um padrão ancora o casamento no fim do texto principal. Em outras posições, '`^`' e '`$`' não possuem significado especial e representam eles mesmos.

Capturas:

Um padrão pode conter subpadrões delimitados por parênteses; eles descrevem *capturas*. Quando um casamento é bem sucedido, as subcadeias da cadeia principal que casam capturas são armazenadas (*capturadas*) para uso futuro. Capturas são numeradas de acordo com seu parêntese esquerdo. Por exemplo, no padrão "`(a*(.)%w(%s*))`", a parte da cadeia casando "`a*(.)%w(%s*)`" é armazenada como a primeira captura (e por isso tem o número 1); o caractere casando "`.`" é capturado com o número 2, e a parte casando "`%s*`" tem o número 3.

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

6.5 – Manipulação de Tabelas

Esta biblioteca oferece funções genéricas para manipulação de tabelas. Ela oferece todas as suas funções dentro da tabela `table`.

Lembre-se que, sempre que uma operação precisa do comprimento de uma tabela, a tabela deve ser uma sequência de fato ou ter um metamétodo `__len` (veja §3.4.6). Todas as funções ignoram chaves não numéricas em tabelas fornecidas como argumentos.

Por razões de desempenho, todas os acessos (*get/set*) a tabelas realizados por estas funções são primitivos.

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

Dada uma lista onde todos os elementos são cadeias ou números, retorna a cadeia `list[i]..sep..list[i+1] ··· sep..list[j]`. O valor padrão para `sep` é a cadeia vazia, o padrão para `i` é 1, e o padrão para `j` é `#list`. Se `i` é maior do que `j`, retorna a cadeia vazia.

table.insert (list, [pos,] value)

Insere o elemento `value` na posição `pos` de `list`, deslocando os elementos `list[pos], list[pos+1], ···, list[#list]`. O valor padrão para `pos` é `#list+1`, assim uma chamada `table.insert(t,x)` insere `x` no fim da lista `t`.

table.pack (···)

Retorna uma nova tabela com todos os parâmetros armazenados nas chaves 1, 2, etc. e com um campo "`n`" com o número total de parâmetros. Note que a tabela resultante pode não ser uma sequência.

table.remove (list [, pos])

Remove de `list` o elemento na posição `pos`, retornando o valor do elemento removido. Quando `pos` é um inteiro entre 1 e `#list`, desloca os elementos `list[pos+1], list[pos+2], ···, list[#list]` e apaga o elemento `list[#list]`; O índice `pos` pode também ser 0 quando `#list` é 0, ou `#list + 1`; nesses casos, a função apaga o elemento `list[pos]`.

O valor padrão para `pos` é `#list`, assim uma chamada `table.remove(l)` remove o último elemento da lista `l`.

table.sort (list [, comp])

Ordena os elementos da lista em uma dada ordem, *in-place*, de `list[1]` até `list[#list]`. Se `comp` é fornecido, então ela deve ser uma função que recebe dois elementos da lista e retorna verdadeiro quando o primeiro elemento deve vir antes do segundo na ordem final (de modo que `not comp(list[i+1],list[i])` será verdadeiro após a ordenação). Se `comp` não é fornecido, então o operador Lua padrão < é usado ao invés.

O algoritmo de ordenação não é estável; isto é, elementos considerados iguais pela ordem fornecida podem ter suas posições relativas alteradas após a ordenação.

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

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

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

Por padrão, `i` é 1 e `j` é `#list`.

6.6 – Funções Matemáticas

Esta biblioteca é uma interface para a biblioteca matemática de C padrão. Ela oferece todas as suas funções dentro da 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 os sinais de ambos os parâmetros para encontrar o quadrante do resultado. (Também trata corretamente o caso quando `x` é zero.)

math.ceil (x)

Retorna o menor inteiro maior ou igual a `x`.

math.cos (x)

Retorna o co-seno de `x` (que se assume estar 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 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 é zero quando `x` é zero).

math.huge

O valor `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 [, base])

Retorna o logaritmo de `x` na base dada. O valor padrão para `base` é *e* (de modo que a função retorna o logaritmo natural de `x`).

math.max (x, ···)

Retorna o valor máximo entre seus argumentos.

math.min (x, ···)

Retorna o valor mínimo entre seus argumentos.

math.modf (x)

Retorna dois números, a parte integral de `x` e a parte fracionária de `x`.

math.pi

O valor de π.

math.pow (x, y)

Retorna x^y. (Você também pode usar a expressão `x^y` para computar esse valor.)

math.rad (x)

Retorna o ângulo `x` (dado em graus) em radianos.

math.random ([m [, n]])

Esta função é um interface para a função geradora pseudo-randômica `rand` simples oferecida por C Padrão. (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 sequências iguais de números.

math.sin (x)

Retorna o seno de `x` (que se assume estar 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 esse valor.)

math.tan (x)

Retorna a tangente de `x` (que se assume estar em radianos).

math.tanh (x)

Retorna a tangente hiperbólica de `x`.

6.7 – Operações Bit a Bit

Esta biblioteca oferece operações bit a bit. Ela oferece todas as suas funções dentro da tabela `bit32`.

A menos que dito de outra maneira, todas as funções aceitam argumentos numéricos no intervalo $(-2^{51},+2^{51})$; cada argumento é normalizado para o resto dessa divisão por 2^{32} e truncado para um inteiro (de algum modo não especificado), de modo que seu valor final cabe no intervalo $[0,2^{32} - 1]$. De maneira similar, todos os resultados estão no intervalo $[0,2^{32} - 1]$. Note que `bit32.bnot(0)` é `0xFFFFFFFF`, o que é diferente de `-1`.

bit32.arshift (x, disp)

Retorna o número `x` deslocado `disp` bits para a direita. O número `disp` pode ser qualquer inteiro representável. Deslocamentos negativos deslocam para a esquerda.

Esta operação de deslocamento é o que é chamado de deslocamento aritmético. Bits vagos à esquerda são preenchidos com cópias do bit mais significativo de `x`; bits vagos à direita são preenchidos com zeros. Em particular, deslocamentos com valores absolutos maiores do que 31 resultam em zero ou `0xFFFFFFFF` (todos os bits originais são deslocados para fora).

bit32.band (···)

Retorna o *and* bit a bit de seus operandos.

bit32.bnot (x)

Retorna a negação bit a bit de `x`. Para qualquer inteiro `x`, a seguinte identidade vale:

```
assert(bit32.bnot(x) == (-1 - x) % 2^32)
```

bit32.bor (···)

Retorna o *or* bit a bit de seus operandos.

bit32.btest (···)

Retorna um booleano sinalizando se o *and* bit a bit de seus operandos é diferente de zero.

bit32.bxor (···)

Retorna o *ou exclusivo* de seus operandos.

bit32.extract (n, field [, width])

Retorna o número sem sinal formado pelos bits `field` a `field + width - 1` de `n`. Bits são numerados de 0 (menos significativo) a 31 (mais significativo). Todos os bits acessados deve estar no intervalo *[0, 31]*.

O padrão para `width` é 1.

bit32.replace (n, v, field [, width])

Retorna uma cópia de `n` com os bits de `field` a `field + width - 1` substituídos pelo valor `v`. Veja `bit32.extract` para detalhes sobre `field` e `width`.

bit32.lrotate (x, disp)

Retorna o número `x` rotacionado `disp` bits para a esquerda. O número `disp` pode ser qualquer inteiro representável.

Para qualquer deslocamento válido, a seguinte identidade vale:

```
assert(bit32.lrotate(x, disp) == bit32.lrotate(x, disp % 32))
```

Em particular, deslocamentos negativos rotacionam para a direita.

bit32.lshift (x, disp)

Retorna o número `x` deslocado `disp` bits para a esquerda. O número `disp` pode ser qualquer inteiro representável. Deslocamentos negativos deslocam para a direita. Em qualquer direção, bits vagos são preenchidos com zeros. Em particular, deslocamentos com valores absolutos maiores do que 31 resultam em zero (todos os bits são deslocados para fora).

Para deslocamentos positivos, a seguinte igualdade vale:

```
assert(bit32.lshift(b, disp) == (b * 2^disp) % 2^32)
```

bit32.rrotate (x, disp)

Retorna o número `x` rotacionado `disp` bits para a direita. O número `disp` pode ser qualquer inteiro representável.

Para qualquer deslocamento válido, a seguinte identidade vale:

```
assert(bit32.rrotate(x, disp) == bit32.rrotate(x, disp % 32))
```

Em particular, deslocamentos negativos rotacionam para a esquerda.

bit32.rshift (x, disp)

Retorna o número `x` deslocado `disp` bits para a direita. O número `disp` pode ser qualquer inteiro representável. Deslocamentos negativos deslocam para a esquerda. Em qualquer direção, bits vagos são preenchidos com zeros. Em particular, deslocamentos com valores absolutos maiores do que 31 resultam em zero (todos os bits são deslocados para fora).

Para deslocamentos positivos, a seguinte igualdade vale:

```
assert(bit32.rshift(b, disp) == math.floor(b % 2^32 / 2^disp))
```

Esta operação de deslocamento é o que é chamado de deslocamento lógico.

6.8 – Facilidades de Entrada e Saída

A biblioteca de E/S oferece dois estilos diferentes para manipulação de arquivos. O primeiro usa descritores de arquivos 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 sobre esses arquivos padrão. O segundo estilo usa descritores de arquivos explícitos.

Ao usar descritores de arquivos implícitos, todas as operações são fornecidas pela tabela `io`. Ao usar descritores de arquivos explícitos, a operação `io.open` retorna um descritor de arquivo e então todas as operações são fornecidas como métodos do descritor de arquivo.

A tabela `io` também oferece três descritores de arquivos pré-definidos com seus significados usuais de C: `io.stdin`, `io.stdout`, e `io.stderr`. A biblioteca de E/S nunca fecha esses arquivos.

A menos que dito de outra maneira, todas as funções de E/S retornam **nil** em caso de falha (mais uma mensagem de erro como um segundo resultado e um código de erro dependente do sistema como um terceiro resultado) e algum valor diferente de **nil** em caso de sucesso. Em sistemas não Posix, a computação da mensagem de erro e do código de erro em caso de erros pode não ser segura se há múltiplos fluxos de execução, pois ela depende da variável C global `errno`.

io.close ([file])

Equivalente a `file:close()`. Sem um `file`, fecha o arquivo de saída padrão.

io.flush ()

Equivalente a `io.output():flush()`.

io.input ([file])

Quando chamada com um nome de arquivo, abre o arquivo nomeado (em modo texto), e estabelece seu manipulador como o arquivo de saída padrão. Quando chamada com um manipulador de arquivo, simplesmente estabelece esse 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 lança o erro, ao invés de retornar um código de erro.

io.lines ([filename ···])

Abre o nome do arquivo fornecido em modo de leitura e retorna uma função iteradora que funciona como `file:lines(···)` sobre o arquivo aberto. Quando a função iteradora detecta o fim do arquivo, retorna **nil** (para finalizar o laço) e automaticamente fecha o arquivo.

A chamada `io.lines()` (sem nome de arquivo) é equivalente a `io.input():lines()`; isto é, ela itera sobre as linhas do arquivo de entrada padrão. Nesse caso ela não fecha o arquivo quando o laço termina.

Em caso de erros esta função lança o erro, ao invés de retornar um código de erro.

io.open (filename [, mode])

Esta função abre um arquivo, no modo especificado na cadeia `mode`. Retorna um novo manipulador de arquivo, ou, em caso de erros, **nil** mais uma mensagem de erro.

A cadeia `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 são apagados;
- **"a+":** modo de atualização de adição, todos os 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.

io.output ([file])

Similar a `io.input`, mas opera sobre o arquivo de saída padrão.

io.popen (prog [, mode])

Esta função é dependente do sistema e não está disponível em todas as plataformas.

Começa o programa `prog` em um processo separado e retorna um manipulador de arquivo que você pode usar para ler dados desse programa (se `mode` é `"r"`, o padrão) ou para escrever dados para esse programa (se `mode` é `"w"`).

io.read (···)

Equivalente a `io.input():read(···)`.

io.tmpfile ()

Retorna um manipulador para um arquivo temporário. Esse 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, `"closed 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 isso leva uma quantidade imprevisível de tempo para acontecer.

Ao fechar um manipulador de arquivo criado com `io.popen`, `file:close` retorna os mesmos valores retornados por `os.execute`.

file:flush ()

Salva qualquer dado escrito para `file`.

file:lines (···)

Retorna uma função iteradora que, cada vez que é chamada, lê o arquivo de acordo com os formatos fornecidos. Quando nenhum formato é fornecido, usa "*l" como um padrão. Como um exemplo, a construção

```
for c in file:lines(1) do corpo end
```

irá iterar sobre todos os caracteres do arquivo, começando na posição corrente. Diferente de `io.lines`, esta função não fecha o arquivo quando o laço termina.

Em caso de erros esta função lança o erro, ao invés de retornar um código de erro.

file:read (···)

Lê o arquivo `file`, de acordo com os formatos fornecidos, os quais especificam o que ler. Para cada formato, a função retorna uma cadeia (ou um número) com os caracteres lidos, ou **nil** se ela não conseguiu ler dados com o formato especificado. Quando chamada sem formatos, usa o formato padrão que lê a próxima linha (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, começando na posição corrente. no fim do arquivo, retorna a cadeia vazia.
- **`"*l"`:** lê a próxima linha pulando o fim de linha, retornando **nil** no fim do arquivo. Este é o formato padrão.
- **`"*L"`:** lê a próxima linha mantendo o fim de linha (se presente), retornando **nil** no fim do arquivo.
- ***number*:** lê uma cadeia com até esse número de bytes, retornando **nil** no fim do arquivo. Se `number` é zero, não lê nada e retorna uma cadeia vazia, ou **nil** 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 segue:

- **`"set"`:** base é a posição 0 (início do arquivo);
- **`"cur"`:** base é a posição corrente;
- **`"end"`:** base é o fim do arquivo;

Em caso de sucesso, `seek` retorna a posição final do arquivo, medida em bytes a partir do início do arquivo. Se `seek` falha, retorna **nil**, mais uma cadeia descrevendo o erro.

O valor padrão para `whence` é `"cur"`, e para `offset` é 0. Por isso, a chamada `file:seek()` retorna a posição corrente do arquivo, sem modificá-la; a chamada `file:seek("set")` ajusta a posição para o início do arquivo (e retorna 0); e a chamada `file:seek("end")` ajusta a posição para o fim do arquivo, e retorna seu tamanho.

file:setvbuf (mode [, size])

Estabelece o modo de bufferização para um arquivo de saída. Há três modos disponíveis:

- **`"no"`:** sem bufferização; o resultado de qualquer operação de saída aparece imediatamente.
- **`"full"`:** bufferização completa; a operação é realizada somente quando o buffer está cheio ou quando você explicitamente `descarrega` o arquivo (veja `io.flush`).
- **`"line"`:** bufferização de linha; a saída é bufferizada até que uma quebra de linha seja produzida ou haja qualquer entrada de alguns arquivos especiais (tal como um dispositivo 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 ou números.

Em caso de sucesso, esta função retorna `file`. Caso contrário retorna **nil** mais uma cadeia descrevendo o erro.

6.9 – Facilidades do Sistema Operacional

Esta biblioteca é implementada através da tabela `os`.

os.clock ()

Retorna uma aproximação da quantidade em segundos de tempo de CPU usada por um 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, essa é a hora a ser formatada (veja a função `os.time` para uma descrição desse valor). Caso contrário, `date` formata a hora corrente.

Se `format` começa com '`!`', então a data está 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), and `isdst` (flag do horário de verão, um booleano). Esse último campo pode estar ausente se a informação não está disponível.

Se `format` não é "`*t`", então `date` retorna a data como uma cadeia, formatada de acordo com as mesmas regras de ANSI C function `strftime`.

Quando chamada sem argumentos, `date` retorna uma representação razoável de data e hora que depende do sistema hospedeiro e do idioma corrente (isto é, `os.date()` é equivalente a `os.date("%c")`).

Em sistemas não Posix, esta função pode não ser segura se há múltiplos fluxos de execução por causa de sua dependência de C function `gmtime` e C function `localtime`.

os.difftime (t2, t1)

Retorna o número de segundos da hora `t1` para a hora `t2`. Em POSIX, Windows, e alguns outros sistemas, esse valor é exatamente `t2-t1`.

os.execute ([command])

Esta função é equivalente a ANSI C function `system`. Ela passa `command` para ser executado por um interpretador de comandos de sistema operacional. Seu primeiro resultado é **true** se o comando terminou com sucesso, ou **nil** caso contrário. Após esse primeiro resultado a função retorna uma cadeia e um número, como segue:

- **`"exit"`:** o comando terminou normalmente; o número seguinte é o estado de saída do comando.
- **`"signal"`:** o comando foi terminado por um sinal; o número seguinte é o sinal que terminou o comando.

Quando chamada sem um `command`, `os.execute` retorna um booleano que é verdadeiro se um interpretador de comandos está disponível.

os.exit ([code [, close])

Chama ANSI C function `exit` para terminar o programa hospedeiro. Se `code` é **true**, o estado retornado é `EXIT_SUCCESS`; se `code` é **false**, o código retornado é `EXIT_FAILURE`; se `code` é um número, o estado retornado é esse número. O valor padrão para `code` é **true**.

Se o segundo argumento opcional `close` é verdadeiro, fecha o estado Lua antes de sair.

os.getenv (varname)

Retorna o valor da variável de ambiente do processo `varname`, ou **nil** se a variável não está definida.

os.remove (filename)

Apaga o arquivo (ou diretório vazio, em sistemas POSIX) com o nome fornecido. Se esta função falha, ela retorna **nil**, mais uma cadeia descrevendo o erro e o código do erro.

os.rename (oldname, newname)

Renomeia o arquivo ou diretório chamado `oldname` para `newname`. Se esta função falha, ela retorna **nil**, mais uma cadeia descrevendo o erro e o código do erro.

os.setlocale (locale [, category])

Estabelece o idioma (*locale*) corrente do programa. `locale` é uma cadeia dependente do sistema especificando um idioma: `category` é uma cadeia opcional descrevendo qual categoria mudar: "`all`",

"collate", "ctype", "monetary", "numeric", ou "time"; a categoria padrão é "all". A função retorna o nome do novo idioma, ou **nil** se a requisição não pode ser honrada.

Se locale é a cadeia vazia, o idioma corrente é definido como uma idioma nativo definido pela implementação. Se locale é a cadeia "C", o idioma corrente é definido como o idioma de C padrão.

Quando chamada com **nil** como primeiro argumento, esta função somente retorna o nome do idioma corrente para a categoria fornecida.

Esta função pode não ser segura se há múltiplos fluxos de execução por causa de sua dependência de C function setlocale.

os.time ([table])

Retorna o tempo corrente quando chamada sem argumentos, ou um tempo representando a data e a hora especificados pela tabela dada. Esta tabela deve ter campos year, month, e day, e pode ter campos hour (o padrão é 12), min (o padrão é 0), sec (o padrão é 0), e isdst (o padrão é **nil**). Para uma descrição desses campos, veja a função os.date.

O valor retornado é um número, cujo significado depende de seu sistema. Em POSIX, Windows, e alguns outros sistemas, este número conta o número de segundos desde algum dado tempo de início (a "época"). Em outros sistemas, o significado não é especificado, e o número retornado por time por ser usado somente como um argumento para os.date e os.difftime.

os.tmpname ()

Retorna uma cadeia com um nome de arquivo que pode ser usado para um arquivo temporário. O arquivo deve ser explicitamente aberto antes de seu uso e explicitamente removido quando não mais necessário.

Em sistemas POSIX, esta função também cria um arquivo com esse nome, para evitar riscos de segurança. (Alguma outra pessoa poderia criar o arquivo com permissões erradas no tempo entre obter o nome e criar o arquivo.) Você ainda tem que abrir o arquivo para usá-lo e para removê-lo (mesmo se você não usá-lo).

Quando possível, você pode preferir usar io.tmpfile, que automaticamente remove o arquivo quando o programa termina.

6.10 – A Biblioteca de Depuração

Esta biblioteca oferece a funcionalidade da interface de depuração (§4.9) para programas Lua. Você deve ter cuidado ao usar esta biblioteca. Várias de suas funções violam suposições básicas a respeito de código Lua (e.g., que variáveis locais a uma função não podem ser acessadas de fora; que metatabelas de userdatas não podem ser modificadas por código Lua; que programas Lua não quebram) e por isso podem comprometer código que de outro modo seria seguro. Além disso, algumas funções desta biblioteca podem ser lentas.

Todas as funções desta biblioteca são oferecidas dentro da tabela debug. Todas as funções que operam sobre um fluxo de execução possuem um primeiro argumento opcional que é o fluxo sobre o qual operar. O padrão é sempre o fluxo corrente.

debug.debug ()

Entra em um modo interativo com o usuário, executando cada cadeia que o usuário entra. Usando comandos simples e outras facilidades de depuração, o usuário pode inspecionar variáveis globais e locais, modificar seus valores, avaliar expressões, e assim por diante. Uma linha contendo somente a palavra cont finaliza esta função, de modo que a chamadora continua sua execução.

Note que comandos para debug.debug não estão lexicamente aninhados dentro de nenhuma função e assim não possuem acesso direto a variáveis locais.

debug.gethook ([thread])

Retorna as configurações de gancho correntes do fluxo, como três valores: a função de gancho corrente, a máscara de gancho corrente, e o contador de gancho corrente (como definido pela função `debug.sethook`).

debug.getinfo ([thread,] f [, 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 `f`, o qual significa a função executando no nível `f` da pilha de chamadas do fluxo fornecido: o nível 0 é a função corrente (a própria `getinfo`); o nível 1 é a função que chamou `getinfo` (exceto para chamadas finais, que não contam na pilha); e assim por diante. Se `f` 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 preencher. O padrão para `what` é obter toda informação disponível, 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 puder ser encontrado. e a expressão `debug.getinfo(print)` retorna uma tabela com toda informação disponível sobre a função `print`.

debug.getlocal ([thread,] f, local)

Esta função retorna o nome e o valor da variável local com índice `local` da função no nível `f` da pilha. Esta função acessa não somente variáveis locais explícitas, mas também parâmetros, temporários, etc.

O primeiro parâmetro ou variável local possui índice 1, e assim por diante, até a última variável ativa. Índices negativos se referem a parâmetros vararg; -1 é o primeiro parâmetro vararg. A função retorna **nil** se não há nenhuma variável com o índice fornecido, e lança um erro quando chamada com um nível fora do intervalo. (Você pode chamar `debug.getinfo` para verificar se o nível é válido.)

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

O parâmetro `f` também pode ser uma função. Nesse caso, `getlocal` retorna somente o nome dos parâmetros da função.

debug.getmetatable (value)

Retorna a metatabela do `value` fornecido ou **nil** se ele não possui uma metatabela.

debug.getregistry ()

Retorna a tabela de registro (veja §4.5).

debug.getupvalue (f, up)

Esta função retorna o nome e o valor do upvalue com índice `up` da função `f`. A função retorna **nil** se não há upvalue com o índice fornecido.

debug.getuservalue (u)

Retorna o valor Lua associado a `u`. Se `u` não é um userdata, retorna **nil**.

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 qualquer combinação dos seguintes caracteres, com o significado dado:

- **'`c`':** o gancho é chamado toda vez que Lua chama uma função;
- **'`r`':** o gancho é chamado toda vez que retorna de uma função;

- **`'l'`:** o gancho é chamado toda vez que Lua entra uma nova linha de código.

Além disso, com um `count` diferente de zero, o gancho é chamado também após cada `count` instruções.

Quando chamada sem argumentos, `debug.sethook` desabilita o gancho.

Quando o gancho é chamado, seu primeiro parâmetro é uma cadeia descrevendo o evento que disparou sua chamada: `"call"` (ou `"tail call"`), `"return"`, `"line"`, e `"count"`. Para eventos de linha, o gancho também recebe o novo número de linha como seu segundo parâmetro. Dentro de um gancho, você pode chamar `getinfo` com nível 2 para obter mais informação sobre a função executando. (o nível 0 é a função `getinfo`, e o nível 1 é a função de gancho).

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 nenhuma variável local com o índice fornecido, e lança um erro quando chamada com um `level` fora do intervalo. (Você pode chamar `getinfo` para verificar se o nível é válido.) Caso contrário, retorna o nome da variável local.

Veja `debug.getlocal` para mais informações sobre índices e nomes de variáveis.

debug.setmetatable (value, table)

Estabelece a `table` fornecida (que pode ser **nil**) como a metatabela para o `value` fornecido. Retorna `value`.

debug.setupvalue (f, up, value)

Esta função atribui o valor `value` ao upvalue com índice `up` da função `f`. A função retorna **nil** se não há nenhum upvalue com o índice fornecido. Caso contrário, retorna o nome do upvalue.

debug.setuservalue (udata, value)

Estabelece o `value` fornecido como o valor Lua associado ao `udata` dado. `value` deve ser uma tabela ou **nil**; `udata` deve ser um userdata completo.

Retorna `udata`.

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

Se `message` está presente mas não é uma cadeia nem **nil**, esta função retorna `message` sem processamento adicional. Caso contrário, retorna uma cadeia 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 começar o traço (o padrão é 1, a função chamando `traceback`).

debug.upvalueid (f, n)

Retorna um identificador único (como um userdata leve) para o upvalue com número `n` da função fornecida.

Esses identificadores únicos permitem um programa verificar se diferentes fechos compartilham upvalues. Fechos Lua que compartilham um upvalue (isto é, que acessam uma mesma variável local externa) retornarão identificadores idênticos para esses índices de upvalues.

debug.upvaluejoin (f1, n1, f2, n2)

Faz o `n1`-ésimo upvalue do fecho Lua `f1` se referir ao `n2`-ésimo upvalue do fecho Lua `f2`.

7 – O Interpretador de Linha de Comando Lua

Embora Lua tenha sido projetada como uma linguagem de extensão, para ser embarcada em um programa

C hospedeiro, ela também é frequentemente usada como uma linguagem auto-suficiente. Um interpretador para Lua como uma linguagem auto-suficiente, chamado simplesmente de `lua`, é fornecido com a distribuição padrão. O interpretador de linha de comando inclui todas as bibliotecas padrão, incluindo a biblioteca de depuração. Seu uso é:

```
lua [options] [script [args]]
```

As opções são:

- **`-e` *`comando`*:** executa a cadeia *comando*;
- **`-l` *`mod`*:** "requisita" *mod*;
- **`-i`:** entra em modo interativo após executar *script*;
- **`-v`:** imprime informações da versão;
- **`-E`:** ignora variáveis de ambiente;
- **`--`:** para de tratar opções;
- **`-`:** executa `stdin` como um arquivo e para de tratar opções.

Após tratar suas opções, `lua` executa o *script* fornecido, passando pra ele os *args* fornecidos como argumentos do tipo cadeia. Quando chamado sem argumentos, `lua` comporta-se como `lua -v -i` quando a entrada padrão (`stdin`) é um terminal, e como `lua -` caso contrário.

Quando chamado sem a opção `-E`, o interpretador verifica se há uma variável de ambiente `LUA_INIT_5_2` (ou `LUA_INIT` se ela não está definida) antes de executar ser argumento. Se o conteúdo da variável possui o formato `@`*`nomearquivo`*, então `lua` executa o arquivo. Caso contrário, `lua` executa a própria cadeia.

Quando chamado com a opção `-E`, além de ignorar `LUA_INIT`, Lua também ignora os valores de `LUA_PATH` e `LUA_CPATH`, estabelecendo os valores de `package.path` e `package.cpath` com os caminhos padrão definidos em `luaconf.h`.

Todas as opções são tratadas em ordem, exceto `-i` e `-E`. Por exemplo, uma invocação como

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

irá primeiro atribuir 1 a `a`, em seguida imprimir o valor de `a`, e finalmente executar o arquivo `script.lua` sem argumentos. (Aqui `$` é o *prompt* do interpretador de comandos. Seu prompt pode ser diferente.)

Antes de começar a executar o script, `lua` junta todos os argumentos 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 vai para o índice 1, e assim por diante. Quaisquer argumentos antes do nome do script (isto é, o nome do interpretador mais as opções) vão para índices negativos. Por exemplo, na chamada

```
$ lua -la b.lua t1 t2
```

o interpretador primeiro executa o arquivo `a.lua`, em seguida cria uma 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 esses argumentos com a expressão vararg '`...`'.

Em modo interativo, se você escrever um comando incompleto, o interpretador espera que ele seja completado mostrando um prompt diferente.

Em caso de erros não protegidos no script, o interpretador reporta o erro para o fluxo de saída padrão. Se o objeto de erro é uma cadeia, o interpretador adiciona um traço de pilha a ela. Caso contrário, se o objeto de erro possui um metamétodo `__tostring`, o interpretador chama esse metamétodo para produzir a mensagem final. Finalmente, se o objeto de erro é **nil**, o interpretador não reporta o erro.

Ao terminar normalmente, o interpretador fecha seu estado Lua principal (veja `lua_close`). O script pode evitar esse passo chamando `os.exit` para terminar.

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 se ela começa com `#`. Assim, scripts Lua podem virar

programas executáveis usando `chmod +x` e a forma `#!`, como em

```
#!/usr/local/bin/lua
```

(Obviamente, a localização do interpretador Lua pode ser diferente em sua máquina. Se `lua` está no seu `PATH`, então

```
#!/usr/bin/env lua
```

é uma solução mais portável.)

8 – Incompatibilidades com a Versão Anterior

Aqui listamos as incompatibilidades que você pode encontrar ao migrar um programa de Lua 5.1 para Lua 5.2. Você pode evitar algumas incompatibilidades compilando Lua com opções apropriadas (veja o arquivo `luaconf.h`). Contudo, todas essas opções de compatibilidade serão removidas na próxima versão de Lua. De modo similar, todas as características marcadas como obsoletas em Lua 5.1 foram removidas em Lua 5.2.

8.1 – Mudanças na Linguagem

- O conceito de *ambiente* mudou. Somente funções Lua possuem ambientes. Para estabelecer o ambiente de uma função Lua, use a variável `_ENV` ou a função `load`.

  Funções C não possuem mais ambientes. Use um upvalue com uma tabela compartilhada se você precisa manter estado compartilhado entre várias funções C. (Você pode usar `luaL_setfuncs` para abrir uma biblioteca C com todas as funções compartilhando um upvalue comum.)

  Para manipular o "ambiente" de um userdata (o qual é agora chamado de valor do usuário), use as novas funções `lua_getuservalue` and `lua_setuservalue`.

- Identificadores Lua não podem usar letras dependentes do idioma.
- Fazer um passo ou uma coleta completa do coletor de lixo não reinicia o coletor se ele tiver sido parado.
- Tabelas fracas com chaves fracas agora agem como *tabelas efêmeras*.
- O evento de *retorno final* em ganchos de depuração foi removido. Ao invés disso, chamadas finais geram um evento novo especial, *chamada final*, de modo que o depurador pode saber que não haverá um evento de retorno correspondente.
- A igualdade entre valores do tipo função foi modificada. Agora, uma definição de função pode não criar um novo valor; ela pode reusar algum valor anterior se não há uma diferença observável para a nova função.

8.2 – Mudanças nas Bibliotecas

- A função `module` está obsoleta. É fácil montar um módulo com código Lua normal. Não se espera que módulos sejam variáveis globais.
- As funções `setfenv` e `getfenv` foram removidas, por causa das mudanças em ambientes.
- A função `math.log10` está obsoleta. Use `math.log` com 10 como seu segundo argumento, ao invés.
- A função `loadstring` está obsoleta. Use `load` ao invés; ela agora aceita cadeias como argumentos e é exatamente equivalente a `loadstring`.
- A função `table.maxn` está obsoleta. Escreva-a em Lua se você realmente precisa dela.
- A função `os.execute` agora retorna **true** quando o comando termina com sucesso e **nil** mais informação de erro caso contrário.
- A função `unpack` foi movida para a biblioteca de tabelas e desse modo deve ser chamada como `table.unpack`.
- A classe de caracteres `%z` em padrões está obsoleta, pois agora padrões podem conter '`\0`' como um caractere normal.

- A tabela `package.loaders` foi renomeada para `package.searchers`.
- Lua não tem mais verificação de bytecode. Assim, todas as funções que carregam código (`load` e `loadfile`) são potencialmente inseguras ao carregar dados binários não confiáveis. (Na verdade, essas funções já eram inseguras por causa de falhas no algoritmo de verificação.) Quando em dúvida, use o argumento `mode` dessas funções para restringi-las a carregar trechos textuais.
- Os caminhos padrão na distribuição oficial podem mudar entre versões.

8.3 – Mudanças na API

- O pseudo-índice `LUA_GLOBALSINDEX` foi removido. Você deve obter o ambiente global do registro (veja §4.5).
- O pseudo-índice `LUA_ENVIRONINDEX` e as funções `lua_getfenv`/`lua_setfenv` foram removidas, pois funções C não possuem mais ambientes.
- A função `luaL_register` está obsoleta. Use `luaL_setfuncs` de modo que seu módulo não crie globais. (Não se espera mais que módulos estabeleçam variáveis globais.)
- O argumento `osize` da função de alocação pode não ser zero ao criar um novo bloco, isto é, quando `ptr` é `NULL` (veja `lua_Alloc`). Use somente o teste `ptr == NULL` para verificar se o bloco é novo.
- Os finalizadores (metamétodos `__gc`) para userdatas são chamados na ordem reversa em que eles foram marcados para finalização, não na que eles foram criados (veja §2.5.1). (A maioria dos userdatas são marcados imediatamente após eles serem criados.) Além disso, se a metatabela não possui um campo `__gc` quando definida, o finalizador não será chamado, mesmo se ele for definido depois.
- `luaL_typerror` foi removida. Escreva sua própria versão se você precisar.
- A função `lua_cpcall` está obsoleta. Você pode simplesmente empilhar a função com `lua_pushcfunction` e chamá-la com `lua_pcall`.
- As funções `lua_equal` e `lua_lessthan` estão obsoletas. Use a nova `lua_compare` com opções apropriadas ao invés.
- A função `lua_objlen` foi renomeada para `lua_rawlen`.
- A função `lua_load` tem um parâmetro extra, `mode`. Passe `NULL` para simular o comportamento antigo.
- A função `lua_resume` tem um parâmetro extra, `from`. Passe `NULL` ou o fluxo fazendo a chamada.

9 – A Sintaxe Completa de Lua

Aqui está a sintaxe completa de Lua em BNF estendido. (Ela não descreve as precedências dos operadores.)

```
trecho ::= bloco

bloco ::= {comando} [comandoret]

comando ::=  ';' |
         listavars '=' listaexps |
         chamadafunção |
         rótulo |
         break |
         goto Nome |
         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 listanomes in listaexps do bloco end |
         function nomefunção corpofunção |
         local função Nome corpofunção |
         local listanomes ['=' listaexps]
```

```
comandoret ::= return [listaexps] [';']

rótulo ::= '::' Nome '::'

nomefunção ::= Nome {'.' Nome} [':' Nome]

listavars ::= var {',' var}

var ::=  Nome | prefixexp '[' exp ']' | prefixexp '.' Nome

listanomes ::= Nome {',' Nome}

listaexps ::= exp {',' exp}

exp ::=  nil | false | true | Número | Cadeia | '...' | deffunção |
         expprefixo | construtortabela | exp opbin exp | opunária exp

expprefixo ::= var | chamadafunção | '(' exp ')'

chamadafunção ::=  expprefixo args | expprefixo ':' Nome args

args ::=  '(' [listaexps] ')' | construtortabela | Cadeia

deffunção ::= function corpofunção

corpofunção ::= '(' [listapars] ')' bloco end

listapars ::= listanomes [',' '...'] | '...'

construtortabela ::= '{' [listacampos] '}'

listacampos ::= campo {sepcampos campo} [sepcampos]

campo ::= '[' exp ']' '=' exp | Nome '=' exp | exp

sepcampos ::= ',' | ';'

opbin ::= '+' | '-' | '*' | '/' | '^' | '%' | '..' |
          '<' | '<=' | '>' | '>=' | '==' | '~=' |
          and | or

opunária ::= '-' | not | '#'
```

Last update: Tue Jan 28 13:47:43 BRST 2014