LabLua
www.lua.inf.puc-rio.br

Lua

Conceitos Básicos e API C

Agosto de 2008

Sumário

Capítulo 1

Lua

1.1 Introdução

Lua é uma linguagem de programação criada em 1993 no Tecgraf, um laboratório de pesquisa e desenvolvimento da PUC-Rio. Lua é uma linguagem de script dinâmica, semelhante a outras linguagens de script como Python, Ruby e PHP.

Lua destaca-se pela sua simplicidade, portabilidade, rapidez e pela facilidade com que podemos embutir um interpretador Lua em uma aplicação C. Além disso, Lua é a única linguagem criada em um país em desenvolvimento a ganhar relevância global.

Outra característica importante de Lua é o seu tamanho pequeno. O núcleo da linguagem somado às suas bibliotecas padrões ocupa menos de $200k$.

Lua é uma linguagem de propósito geral, que pode ser utilizada tanto para escrever pequenos scripts, com algumas poucas linhas de código, como sistemas complexos, com dezenas de milhares de linhas de código.

Atualmente a linguagem é utilizada principalmente em jogos, como *World of Warcraft*, *The Sims* e *Sim City*. Contudo, Lua não é uma linguagem específica para jogos, sendo utilizada também em muitas outras aplicações, como o software *Adobe Photoshop Lightroom* e o middleware *Ginga* do Sistema Brasileiro de TV Digital.

Um outro uso de Lua é no desenvolvimento de aplicações web, como o *Publique!*, um gerenciador de conteúdo para web desenvolvido pela Fábrica Digital, e *Sputnik*, um wiki extensível disponível como software livre.

1.2 Primeiros Passos

Um pedaço de código Lua é chamado de *trecho*. Um trecho pode ser desde uma simples linha, digitada no modo interativo, como um programa inteiro. Lua compila trechos de código em instruções para uma máquina virtual de maneira muito rápida (Trechos com mais de $1M$ de código Lua podem ser compilados em menos de um segundo).

Para executar um trecho de código Lua, podemos usar um interpretador da linguagem, que chamaremos de `lua` (notem que ao nos referirmos à linguagem, usamos a letra L maiúscula e ao falarmos do interpretador a letra l minúscula).

Caso o programa não tenha sido compilado, o interpretador `lua` irá compilá-lo primeiro para em seguida executá-lo. (Caso você deseje apenas compilar um programa, utilize o compilador `luac`.)

Variáveis em Lua, assim como em C e Java, possuem escopo léxico. Quando uma variável é declarada, o padrão é que ela seja global. Para indicar que uma variável é local ao bloco do programa na qual ela foi declarada, devemos usar a palavra-chave **local**. Além de ser uma boa prática de programação declarar as variáveis como locais, ao fazer isso o programador também pode melhorar o desempenho do programa, uma vez que em Lua o acesso a variáveis locais é bastante mais rápido do que o acesso a variáveis locais.

Vamos apresentar um pouco da sintaxe de Lua através do programa a seguir que lê um número da entrada e retorna verdadeiro se o número é par e falso caso contrário. O exemplo não se propõe a apresentar a melhor solução, mas a apresentar diferentes aspectos da linguagem.

```
01  function odd (n)
02    if n == 0 then
03      return false
04    else
05      return even (n-1)
06    end
07  end
08
09  function even (n)
10    if n == 0 then
11      return true
12    else
13      return odd (n-1)
14    end
15  end
16
17  local n = io.read ("*number")
18  print (even (n))
```

Em Lua, funções são declaradas usando a palavra-chave **function**. A função *odd* retorna *true* se um número é ímpar e *false* caso contrário, ao passo que a função *even* faz o oposto.

Essas duas funções foram escritas de maneira recursiva para enfatizar que Lua implementa chamadas finais próprias. Isso quer dizer que quando o último comando executado por uma função é uma chamada a uma outra função (ou a ela mesma), a pilha de execução não cresce. Ou seja, podemos escrever funções que utilizam chamadas finais sem nos preocuparmos com o tamanho da pilha de execução, que permanecerá constante.

Na linha 17, declaramos uma variável local *n* (notem o uso da palavra-chave **local**), que irá receber um número digitado pelo usuário. Para ler um número da entrada, utilizamos aqui a função *read* da biblioteca padrão de entrada e saída *io*.

Escreva o programa anterior em um arquivo chamado *par.lua* e em seguida tente executá-lo usando o interpretador `lua` através do seguinte comando:

```
lua par.lua
```

Uma solução mais simples para o problema de determinar se um número é par pode ser obtida através do uso do operador módulo de Lua %, que retorna o resto de uma divisão:

```
1  local n = io.read ("*number")
2  print (n % 2 == 0)
```

1.3 Tipos

Em Lua, não declaramos o tipo de uma variável. O tipo da variável é determinado dinamicamente, dependendo do valor que ela está armazenando. Sendo assim, uma variável pode armazenar valores de qualquer um dos tipos básicos de Lua.

Existem oito tipos básicos em Lua: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread* e *table*.

O tipo *nil* possui **nil** como seu único valor, de modo que nenhum outro valor é igual a **nil**. Em geral, usamos esse tipo para indicar a ausência de um valor.

O tipo *boolean* representa os valores booleanos **true** e **false**. Uma expressão será considerada falsa caso seja igual a **nil** ou a **false**, sendo verdadeira caso contrário (inclusive quando ela for 0).

Ao contrário da maioria das linguagens, Lua possui somente um tipo numérico básico, *number*, que é ponto flutuante por padrão. Não há um tipo específico para números inteiros, mas o programador não precisa se preocupar com isso, pois Lua pode representar qualquer número inteiro de 32 bits corretamente e faz isso de maneira transparente.

O tipo *string* representa cadeias de caracteres, as quais podem ser delimitadas por aspas duplas ou simples. Se uma cadeia começa com aspas simples, ela deve terminar com aspas simples. De maneira análoga, se uma cadeia começa com aspas duplas, ela deve terminar com aspas duplas. Para escrever uma cadeia que se estende por várias linhas, deve-se usar a notação [[para abrir a cadeia e]] para fechar a cadeia.

Cadeias de caracteres são imutáveis em Lua, toda vez que você altera uma cadeia, você está na verdade criando uma nova cadeia.

Function é o tipo que representa funções. Em Lua, funções são valores de primeira classe, isso quer dizer que um valor do tipo função não requer nenhum tratamento especial. Dessa forma, a mesma variável que armazena um valor numérico, pode também armazenar uma função. Dado que em Lua funções são valores de primeira classe, temos que a sintaxe:

```
function foo (n)
  ...
end
```

É apenas um açúcar sintático para:

```
foo = function (n)
  ...
end
```

Ou seja, estamos armazenando na variável *foo* um valor do tipo *function*.

O tipo *userdata* é usado para armazenar dados C em variáveis Lua. Valores deste tipo somente podem ser criados ou modificados através da API de Lua com C.

Valores do tipo *thread* representam fluxos de execução independentes. O tipo *thread* (não confundir com processos leves, *threads*, do sistema operacional) dá suporte ao uso de co-rotinas, que serão abordadas mais adiante.

Por fim, o tipo *table* representa uma tabela Lua. Tabelas são o único mecanismo de estruturação de dados de Lua. Com tabelas podemos representar arrays, tabelas de símbolos, conjuntos, grafos, registros, etc. Podemos usar qualquer valor para indexar uma tabela, exceto **nil**.

Usamos colchetes para indexar elementos da tabela. Seja t uma tabela, $t[1]$ representa o elemento na posição 1 da tabela e $t["x"]$ o valor associado com a chave x. Neste último caso, podemos usar também o açúcar sintático $t.x$.

A seguir, ilustramos como uma variável Lua pode armazenar qualquer um dos tipos básicos. Para saber qual é o tipo corrente de uma variável, usamos a função *type*.

```
-- Dois traços seguidos indicam um comentário
local a = 3
print (type (a)) -- imprime "number"
a = "bola"
print (type (a)) -- imprime "string"
a = true
print (type (a)) -- imprime "boolean"
a = print        -- "a" agora é a função "print"
a (type (a))     -- imprime "function"
```

Dentre os tipos básicos de Lua, temos que valores do tipo *function*, *userdata*, *thread* e *table* representam objetos. Dessa forma, uma variável que representa um valor deste tipo não armazena o valor em si, mas apenas uma referência para ele.

1.4 Operadores Relacionais

Lua possui os seguintes operadores relacionais:

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

Os cinco primeiros operadores são bastante comuns em outras linguagens e possuem o significado usual. O operador relacional $\sim=$ significa a negação da igualdade.

Todos os operadores relacionais retornam **true** ou **false**.

Quando comparamos valores que são objetos (valores do tipo *function*, *userdata*, *thread* e *table*), temos que dois valores serão iguais somente se eles representam o mesmo objeto. Como o exemplo abaixo ilustra:

```
local tab1 = {}      -- cria uma tabela
local tab2 = {}

tab1.x = 33          -- associa valor 33 com chave "x"
tab2.x = 33

print (tab1 == tab2) -- imprime "false"

tab2 = tab1
print (tab1 == tab2) -- imprime "true"

tab2.x = 20
print (tab1.x)       -- imprime 20, pois tab1 e tab2 se referem ao mesmo valor
```

1.5 Operadores Lógicos

Lua possui os operadores lógicos **and**, **or** e **not**. Os operadores **and** e **or** usam avaliação de curto circuito, de modo que o segundo operando só é avaliado se necessário.

O resultado do operador **not** é sempre um valor booleano. Ao passo que **or** retorna o primeiro operando que não é **nil** ou **false**. Um idioma comum em Lua é usar operador **or** para atribuir um valor padrão a uma variável, como ilustrado a abaixo:

```
function initx (v)
  x = v or 100
end
```

Seja *initx* uma função que inicializa a variável x com o valor v, a expressão $x = v$ *or* 100 indica que quando v for um valor falso, então a variável x será inicializada com o valor 100.

O operador **and** por sua vez, retorna seu primeiro argumento se ele é falso e o seu segundo argumento caso contrário. A seguir, temos alguns exemplos envolvendo os operadores lógicos:

```
-- Usaremos "-->" para indicar a saída do comando
print (34 or nil)         --> 34
```

```
print (not 34)            --> false
print (true and 0)        --> 0
print (not not 0)         --> true
print (false or "bola")   --> bola
print (n and "33" or "34")
```

É possível definir em Lua um operador ternário, semelhante ao operador `?` de C, usando os operadores lógicos **and** e **or**. Uma expressão do tipo $x\,?\,y\,:\,z$, onde y não é um valor falso (i.e., y é diferente de **nil** e de **false**), pode ser representada em Lua como:

```
x and y or z
```

Usando novamente o exemplo envolvendo números pares, podemos inicializar uma variável s com um determinado valor se n é um número par e com outro valor caso contrário.

```
s = (n % 2 == 0) and "par" or "ímpar"
```

1.6 Variáveis

Para declarar uma variável em Lua, basta escrever o seu nome (que deve começar com uma letra ou um sublinhado). Caso nenhum valor seja atribuído à variável quando ela for declarada, seu valor será **nil**.

Para atribuir um valor a uma variável, use o operador de atribuição `=`, como ilustrado a seguir:

```
x = 1             -- x recebe 1
b, c = "bola", 3  -- b recebe o valor "bola" e c o valor 3
y                 -- o valor de y é nil
print (b, y)      --> bola   nil
```

Ao fazer uma atribuição, todos os valores do lado direito são avaliados em um primeiro momento, e depois estes valores são atribuídos às variáveis do lado esquerdo da atribuição.

É possível termos uma atribuição onde o número de variáveis no lado esquerdo é diferente do número de valores no lado direito. Caso existam mais valores do que variáveis, os valores extra são descartados. Caso contrário, as variáveis extra recebem **nil**. O exemplo a seguir ilustra esses caso:

```
a, b, sobrei = 1, 2
print (a, b, sobrei)  --> 1   2   nil

x, y = "bola", "casa", "sobrei"
print (x, y)          --> bola   casa

x, y = y, x     -- faz a troca de valores (o lado direito é avaliado antes do esquerdo)
print (x, y)    --> casa   bola
```

Como mencionamos anteriormente, variáveis são globais por padrão, de modo que devemos usar a palavra-chave **local** para declarar uma variável como local. O escopo de uma variável local é determinado pelo bloco no qual ela foi declarada.

Uma variável local declarada fora de qualquer bloco é visível em todo o arquivo. Um bloco pode ser criado com as palavras chaves **do**, para abrir um bloco, e **end**, para fechar um bloco, ou através do uso de uma estrutura de controle (*if*, *while*, etc). A seguir, temos um exemplo que ilustra o escopo de variáveis em Lua:

```
local x = 33
local y
print (x, y)    --> 33   nil
if x > 10 then
  local x = 5   -- alterando um "x" local
  y = 9
  print (x, y)  --> 5   9
else
  x = 2         -- alterando o "x" mais externo
  print (x, y)  --> 2   nil
end
print (x, y)    --> 33   9
```

É comum em Lua usar variáveis locais para armazenar valores de algumas variáveis globais, uma vez que o acesso a variáveis locais é mais rápido em Lua. Seja *sqrt* uma função global que calcula a raiz quadrada de um número, um programa Lua que usa esta função teria uma declaração:

```
local sqrt = sqrt
```

A variável local *sqrt* somente é visível após a linha da sua declaração, uma vez que todos os valores do lado direito de uma atribuição são primeiro avaliados e depois atribuídos.

Todas as variáveis globais de Lua, incluindo funções como *print*, estão armazenadas na tabela $\_G$, de modo que quando tentamos acessar uma variável que não é local, o interpretador irá automaticamente procurar pela variável na tabela de globais $\_G$, ou seja, *print* e $\_G.print$ representam a mesma função.

1.7 Criando Tabelas

Existem várias formas de criar uma tabela em Lua usando o operador {}. Para criar uma tabela t vazia, tudo que devemos fazer é $t = \{\}$.

É muito comum, contudo, querermos iniciar uma tabela com alguns valores, ao invés de criarmos uma tabela completamente vazia. O programa a seguir cria uma tabela inicializando-a como um array:

```
-- Cria e inicializa a tabela t
local t = {4, "lua", false}
```

Após a execução do trecho de código acima, $t[1]$ possui o valor 4, $t[2]$ a cadeia *"lua"* e $t[3]$ o valor *false*.

Outra maneira de inicializar uma tabela é como um array associativo, nesse caso devemos fornecer dois valores para cada posição da tabela, um indicando a chave e o outro o valor associado com aquela chave:

```
local t = {x=100, y=200, w=50}
print (t ["y"], t.w)           --> 200   50
```

Após a execução do trecho de código acima, t é uma tabela que possui três chaves: x, com o valor associado 100; y, com o valor associado 200; e w, que possui 50 como valor associado.

Para inserir um novo valor em uma tabela, basta fazer uma atribuição:

```
t[100] = true
t["a"] = "A"
t.curso = "Lua"
```

Outra maneira de inserir valores em uma tabela que está sendo utilizada como array é através da função *table.insert*, que insere um valor em uma determinada posição do array (deslocando os demais elementos) ou no final do array, se nenhuma posição é especificada. O exemplo a seguir ilustra o uso de *table.insert*:

```
local t = {10, 20, 30}
table.insert (t, 2, 15)   -- Insere 15 na posição 2
-- Agora temos t[1] = 10, t[2] = 15, t[3] = 20 e t[4] = 30
table.insert (t, 35)      -- Insere 35 no final da tabela
print (t[1], t[3], t[5])  --> 10   20   35
```

1.8 Tamanho de Tabelas

Se uma tabela estiver sendo utilizada como um array é possível saber o número de elementos do array usando o operador de tamanho #.

O operador # retorna um índice n do array tal que o valor na posição n do array é diferente de **nil** e o valor na posição $n + 1$ é **nil**. Em virtude disto, evite usar o operador # com arrays que possam conter valores **nil** no meio deles. O exemplo a seguir ilustra o funcionamento do operador #.

```
local t = {"a", "b", "c"}
print (#t)                --> 3
table.insert (t, "d")
print (#t)                --> 4
t [6] = "8"
-- Temos um valor nil no meio do array (t[5]). O operador de tamanho não tem um
-- comportamento bem definido neste caso
```

```
print (#t)                 --> ??
t [5] = "e"
-- Agora o array não tem "buracos" e o operador # retorna o valor esperado
print (#t)                 --> 6
```

1.9 Estruturas de Controle

1.9.1 If

A estrutura de controle básica de Lua é o *if*. Caso a condição do *if* seja verdadeira, a parte *then* é executada, caso contrário, a parte *else* é executada. A seguir temos um exemplo da utilização do *if*.

```
local x = 44
if x > 30 then
  print ("maior que 30")
elseif x > 20 then
  print ("maior que 20")
elseif x > 10 then
  print ("maior que 10")
else
  print ("que x pequeno")
end
```

1.9.2 While

O *while* é uma das estruturas de repetição que Lua possui. Caso a condição do laço seja verdadeira, o seu corpo é executado e a condição será então reavaliada, caso contrário o laço termina. O exemplo a seguir ilustra o uso de *while*:

```
local i = 0
while i < 10 do
  print (i)
  i = i + 1
end
```

1.9.3 For numérico

Em Lua, existem dois tipos de *for*: o numérico e o genérico. O *for* numérico possui a seguinte sintaxe:

```
for var=exp1, exp2, exp3 do
  corpo_do_laço
end
```

Onde *exp1* é o valor inicial de *var* e corpo do laço será executado enquanto *var* for menor ou igual *exp2*, no caso de um passo positivo, ou maior ou igual

a *exp2*, no case de um passo negativo. A expressão *exp3* representa o passo do laço. Quando *exp3* não é especificada, assume-se que o seu valor é 1. A seguir, temos um exemplo que usa o *for* numérico:

```
for i=1, 10 do print (i) end

for i=10, 1, -1 do print (i) end
```

A variável *var* é visível somente no corpo do laço e não deve ser alterada. Caso queira sair do laço antes do seu final, você deve usar o comando **break**.

1.9.4 For genérico

O *for* genérico é usado para percorrer os valores retornados por uma função iteradora. Os principais iteradores fornecidos por Lua são: *pairs*, para percorrer as chaves de uma tabela; *ipairs*, para percorrer os índices de um array; e *io.lines*, para percorrer as linhas de um arquivo.

Seja *a* um array, podemos imprimir todos os seus valores com o seguinte *for* genérico:

```
for i, v in ipairs (a) do  -- i guarda o índice, v o valor
  print (v)
end
```

Para imprimir todas as chaves de uma tabela t (em uma ordem qualquer), usamos o seguinte trecho de código:

```
for k, v in pairs (t) do  -- k guarda a chave, v o valor
  print (k, v)
end
```

Exercício 1.9.4

Dado um arquivo de texto onde as linhas ímpares do arquivo possuem o nome de uma pessoa e as linhas pares o sobrenome correspondente ao nome da linha anterior, escreva um programa Lua que lê todas as linhas do arquivo, armazena as informações em uma tabela, onde cada elemento da tabela possui uma chave *nome* e uma chave *sobrenome*, e imprime a tabela resultante.

Utilize o *for* genérico e a função iteradora *io.lines (nomedoarquivo)* para ler as linhas do arquivo.

1.10 Saindo de um bloco

Para sair de um bloco de programa, usamos os comando *break* e *return*. O comando *break* é muito utilizado em laços, pois permite terminar a execução do laço.

O comando *return* permite retornar valores de uma função ou simplesmente terminar a execução de uma função. Em Lua, funções podem retornar múltiplos valores, de modo que o comando *return* pode receber múltiplos argumentos.

Ao usar um comando *return* ou *break* existe a restrição de que este comando deve ser o último comando do bloco. O exemplo a seguir é um código Lua que não compila por este motivo:

```
i = 5
while i < 10 do
  break        -- Erro! break deve ser o último comando do bloco
  i = i + 1    -- Atribuição é o último comando do bloco "while"
end
```

Existem duas soluções para este problema. Reordenar seu código, de modo que o comando *break* ou *return* seja o último comando do bloco, ou criar um novo bloco **do-end** ao redor do comando, como ilustrado a seguir:

```
i = 5
while i < 10 do
  do
    break      -- Ok! break agora é o último comando do bloco
  end
  i = i + 1    -- Atribuição é o último comando do bloco "while"
end
```

Capítulo 2

Funções

2.1 Declarando e Chamando uma Função

Uma função pode ser declarada da seguinte forma:

```
function nomedafunção (arg_1, arg_2, ..., arg_n)
  corpoDaFunção
end
```

Onde as variáveis *arg_N* indicam os parâmetros da função. Em Lua, uma função pode ter desde nenhum parâmetro até centenas deles. Da mesma forma, uma função pode não retornar nenhum valor ou retornar vários.

Vamos definir uma função *foo*, que recebe dois parâmetros e retorna a soma e produto deles:

```
function foo (a, b)
  local x = a or 1  -- x recebe o valor padrão 1 quando a é um valor falso
  local y = b or 1  -- y recebe o valor padrão 1 quando b é um valor falso
  return x + y, x * y
end
```

Embora a função *foo* possua dois parâmetros, é possível chamá-la com um número diferente de parâmetros. No caso de chamarmos *foo* com mais parâmetros do a que função espera, os valores extra são descartados. Caso a função seja chamada com um número menor de parâmetros, o valor dos parâmetros que não foram fornecidos é **nil**. O exemplo a seguir ilustra estas situações:

```
s, p = foo (3, 4)     -- a é 3 e b é 4
print (s, p)          --> 7   12

s, p = foo (2)        -- b é nil e y é inicializado com 1
print (s, p)          --> 3   2
```

```
s, p = foo (2, 5, 8)  -- 8 é descartado
print (s, p)          --> 7   10

print (foo ())        --> 2   1
```

2.2 Número variável de parâmetros

Funções com um número variável de parâmetros podem ser declaradas usando três pontos (...). Para acessar então os parâmetros que foram chamados, usamos a notação {...}, para criar um array onde o primeiro parâmetro da função está na posição 1, o segundo na posição 2 e assim por diante. O exemplo a seguir mostra uma função que recebe cadeias de caracteres e imprime aquelas com tamanho maior do que 3.

```
function maior3 (...)
  for i, v in ipairs {...} do  -- percorre a lista de parâmetros
    if #v > 3 then             -- # é o operador de tamanho para cadeias e arrays
      print (v)
    end
  end
end

maior3 ("bola", "sol", "lua", "balao")  -- Imprime "bola" e "balao"
```

2.3 Retornando um único valor

Em Lua, uma função pode não retornar nenhum valor, retornar apenas um valor ou retornar múltiplos valores. Porém, nem sempre iremos obter todos os valores retornados por uma função.

Quando estamos avaliando uma lista de expressões, iremos obter somente o primeiro valor de retorno de cada membro da lista, com exceção da última expressão, que pode retornar múltiplos valores normalmente. O uso abaixo da função *foo* definida anteriormente ilustra este caso:

```
a, b, c, d = foo (1, 2), foo (3, 4), foo (5, 6)
print (a, b, c, d)    -->  3   7   11   30
```

Podemos notar que a chamada $foo(1, 2)$ retornou apenas o valor 3, sendo que o mesmo aconteceu com a chamada $foo(3, 4)$, que retornou apenas o valor 7. A chamada $foo(5, 6)$, como é a última expressão da lista de expressões, retornou dois valores, 11 e 30.

Às vezes nós também queremos que uma chamada de função retorne somente um valor, não importando se a chamada de função faz parte de uma lista de expressões ou não. Nesse caso, devemos colocar parênteses ao redor da chamada de função, isso limita o número de valores de retorno da função a no máximo 1, como mostra o exemplo a seguir:

```
a, b = foo (5, 10)
print (a, b)          --> 15   50
a, b = (foo (5, 10))
print (a, b)          --> 15   nil
a, b, c = foo (1, 2), (foo (3, 4))
print (a, b, c)       -->  3   7   nil
```

2.4 Valores de Primeira Classe

Funções em Lua são valores de primeira classe, o que quer dizer que podemos passar funções como argumentos para outras funções, bem como retornar funções.

Para ilustrar o uso de funções como valores de primeira classe, vamos definir uma função *map*, que recebe uma tabela e uma função e aplica a função sobre todos os elementos da tabela. Abaixo, temos uma definição da função *map*:

```
function map (f, t)
  for k, v in pairs (t) do
     t [k] = f (v)
  end
end
```

Considerando que os valores armazenados na tabela são números, caso se deseje incrementar todos os elementos da tabela em 1 unidade, podemos fazer a seguinte chamada:

```
function inc (v)
  return v + 1
end
map (inc, t)
```

Onde t é uma tabela Lua. Uma maneira mais compacta de fazer a chamada anterior é definindo a função de incremento no momento em que fazemos a chamada à função *map*:

```
map (function (v) return v + 1 end, t)
```

2.5 Fechos

É comum querermos associar variáveis externas com uma função que vai ser definida, criando um *fecho* (*closure*).

Vamos tomar como exemplo a função *generateinc* definida a seguir. Essa função é usada para definirmos uma nova função de incremento, baseada em um valor inicial e um valor de incremento. A função *generateinc* retorna então uma função de incremento, a qual possui referências para as variáveis externas n e s.

```
local i = 3

function generateinc (init, step)
  local n = init or 0
  local s = step or 1
  return function ()
    n = n + s         -- referencia as variáveis externas n e s
    return n
  end
end

local inc = generateinc ()
print (inc ())            --> 1

local inc_ = generateinc (i)
print (inc (), inc_ ())   --> 2   4

i = 10
print (inc (), inc_ ())   --> 3   5

local inc10 = generateinc (5, i)
print (inc (), inc_ (), inc10 ())   --> 4   6   15
```

Quando a função de incremento é retornada, ela possui um estado associado, no qual ela irá guardar o valor das variáveis *n* e *s*. Usando *generateinc*, criamos uma função *inc*, que irá funcionar como um contador de passo 1, começando a partir de 0.

Em seguida, criamos *inc_*, que também possui passo 1, mas que incrementa valores a partir de 3. Por último, criamos a função *inc10*, cujo passo é 10 e cujo valor inicial do contador é 5.

Capítulo 3

Manipulando Cadeias

Em Lua, cadeias de caracteres são imutáveis, isto é, toda vez que tentamos alterar uma cadeia de caracteres, não estamos realmente modificando a cadeia original, mas sim criando uma nova cadeia. Cada cadeia em Lua é única, de modo que a comparação de duas cadeias é muito rápida (uma mera comparação de ponteiros C).

A manipulação de cadeias em Lua é feita através da biblioteca padrão *string*. Ao usar as funções da biblioteca *string*, lembre-se que em Lua o primeiro caractere da cadeia está no índice 1.

Podemos acessar partes de uma cadeia usando a função *string.sub*, como mostrado a seguir:

```
s = "bolada"
print (string.sub (s, 1, 3))   --> bol
print (string.sub (s, 1))      --> bolada
print (string.sub (s, 5, 6))   --> da
print (string.sub (s, 5))      --> da
print (string.sub (s, -4))     --> lada
```

O primeiro parâmetro de *string.sub* é uma cadeia e o segundo parâmetro indica a posição inicial da subcadeia que queremos obter. *string.sub* possui ainda um terceiro parâmetro, que é opcional e indica a posição final da subcadeia.

Caso o primeiro parâmetro de *string.sub* seja negativo, a indexação da cadeia se dará a partir do seu final.

O tamanho de uma cadeia pode ser obtido através do operador # ou da função *string.len*:

```
a = "boi"
b = "formiga"
print (#a, #b)                --> 3   7
print (string.len (a))        --> 3
print (#string.sub (b, 3))    --> 5
```

3.1 Concatenando Cadeias

Para concatenar duas cadeias, podemos usar o operador de concatenação **..**, como ilustrado a seguir:

```
a = "seg" .. "unda"
b = "feira"
print (a .. "-" .. b)   --> segunda-feira
```

Como cada cadeia em Lua é única, devemos tomar um pouco de cuidado ao escrevermos um programa que pode criar milhares de novas cadeias.

Seja t uma tabela Lua que representa um array onde os elementos são cadeias de caracteres. Queremos escrever um programa que cria uma cadeia com todos os elementos de t separados por vírgulas e depois imprime a cadeia resultante. Uma maneira simples de escrever esse programa é mostrada a seguir:

```
local s = ""
for i, v in ipairs (t) do
  s = s .. v .. ","
end
print (string.sub (s, 1, #s - 1))
```

Caso t tenha poucos elementos, o desempenho do programa anterior será satisfatório. Porém, caso t possua alguns milhares de elementos, a execução do programa anterior irá demorar mais do que o esperado. Isso se deve ao fato de que a cada iteração do laço estamos criando uma nova cadeia que será simplesmente descartada na próxima iteração, podendo ser recolhida depois pelo coletor de lixo.

Como em Lua cada cadeia é única, a criação de milhares de novas cadeias (e a interação desse processo com o coletor de lixo) é um pouco custoso, de forma que é bom evitarmos fazer isso quando existem soluções melhores.

Uma maneira mais eficiente de concatenar várias cadeias é através da função *table.concat*. A função *table.concat* recebe uma tabela como primeiro parâmetro e uma cadeia como segundo parâmetro, que será usada para separar os elementos da tabela. O programa abaixo realiza a mesma tarefa que o programa anterior:

```
local s = table.concat (t, ",")
print (s)
```

Mesmo quando as cadeias que desejamos concatenar não estão armazenadas em uma tabela, em geral é mais eficiente inserir os elementos na tabela e depois usar *table.concat* do que criar milhares de cadeias intermediárias.

3.2 Busca e Substituição

Operações de busca e substituição envolvendo cadeias de caracteres são bastante comuns. Em Lua, podemos buscar uma subcadeia usando a função *string.find*. O primeiro parâmetro de *string.find* é a cadeia na qual será feita a busca e o

segundo parâmetro indica um padrão de busca. Não explicaremos aqui como construir um padrão de busca usando características de expressões regulares, iremos usar somente padrões simples. Para maiores detalhes sobre a biblioteca de cadeias de Lua e os padrões de busca/substituição que ela suporta, por favor consulte o manual de referência da linguagem.

O programa a seguir imprime todas as subcadeias que se encontram entre vírgulas. O terceiro parâmetro de *string.find* é opcional e indica a posição inicial da busca. Quando uma busca é bem-sucedida, as posições de início e fim do padrão na cadeia original são retornadas, caso contrário, o valor de retorno é **nil**:

```
local s = "boi,bola,balao"
local i = 1
local j = string.find (s, ",")

while j do
  print (string.sub (s, i, j-1))
  i = j + 1
  j = string.find (s, ",", i)
end

print (string.sub (s, i))
```

Para obter uma nova cadeia a partir da cadeia *s* do exemplo anterior, com todas as vírgulas sendo substituídas por espaços, podemos usar a função *string.gsub*:

```
local s = "boi,bola,balao"
local news = (string.gsub (s, ",", " "))
print (news)          --> boi bola balao"
```

O segundo parâmetro de *string.gsub* é a cadeia a ser substituída e o terceiro indica a cadeia de substituição. A função *string.gsub* retorna também um segundo resultado, o número de substituições que foram realizadas.

O quarto parâmetro de *string.gsub* é opcional e serve para especificar o número máximo de substituições que devem ser feitas. A seguir, usamos novamente *string.gsub*, indicando que somente 1 substituição deve ser realizada:

```
local s = "boi,bola,balao"
local news = (string.gsub (s, ",", " ", 1))
print (news)          --> boi bola,balao"
```

Capítulo 4

Tabelas

Tabelas são o único mecanismo de estruturação de dados em Lua. Tabelas implementam diretamente arrays associativos e arrays numéricos, e podem ser usadas para implementar diversas outras estruturas de dados, como conjuntos e grafos.

Além disso, tabelas são usadas para representar registros, objetos e módulos Lua.

4.1 Manipulando tabelas

A manipulação de tabelas é feita principalmente através da biblioteca padrão de Lua *table*. Para inserir elementos em uma tabela, usamos a função *insert* da biblioteca *table*, como ilustrado a seguir:

```
local t = {}
for i=1, 3 do
  table.insert (t, i)
end
```

Quando passamos somente dois parâmetros para *insert*, Lua insere o segundo parâmetro na posição $n + 1$ da tabela, onde n é o tamanho da tabela de acordo com o operador #.

Caso se queira inserir o elemento 4 na posição 2 da tabela, devemos fazer a seguinte chamada:

```
table.insert (t, 2, 4)
```

Nesse caso, os elementos da tabela nas posições 2 e 3 vão ser deslocados:

```
print (t[1])     --> 1
print (t[2])     --> 4
print (t[3])     --> 2
print (t[4])     --> 3
```

Contudo, não precisamos necessariamente usar *table.insert* para inserir elementos em uma tabela, podemos fazer isso de maneira direta, com uma simples atribuição:

```
local t = {}
for i=1, 3 do
  t [i] = i
end
```

Ou ainda:

```
local t = {}
for i=1, 3 do
  t [#t+1] = i
end
```

Nos exemplos anteriores, fizemos com que o primeiro índice do array esteja na posição 1 da tabela. Em Lua, o primeiro elemento de um array pode estar no índice 0, 1, -33 ou qualquer outro índice. Contudo, ao usarmos funções das bibliotecas padrões de Lua, assume-se que o primeiro elemento do array está no índice 1, de modo que é uma boa prática fazer com que o primeiro elemento do seu array esteja no índice 1.

Da mesma forma que temos uma função *insert*, também existe uma função *remove*, que remove o elemento em um dado índice da tabela ou o último elemento, caso nenhum índice seja fornecido:

```
local t = {10, 20, 30}
table.remove (t)           -- remove o último elemento
print (t[1], t[2], t[3])   --> 10   20   nil
table.insert (t, 40)       -- insere elemento no fim
table.remove (t, 2)        -- remove o segundo elemento, deslocando os elementos seguintes
print (t[1], t[2], t[3])   --> 10   40   nil
```

Mais uma vez, podemos remover os elementos diretamente, atribuindo **nil** a uma determinada posição:

```
local t = {10, 20, 30}
t [3] = nil                -- remove o último elemento
print (t[1], t[2], t[3])   --> 10   20   nil
t [#t+1] = 40              -- insere elemento no fim
t [2] = nil                -- remove o segundo elemento, não desloca os elementos restantes
print (t[1], t[2], t[3])   --> 10   nil   40
```

Ao remover elementos desta forma, devemos tomar cuidado com os *buracos* que ficam na tabela, i.e., valores **nil** que estão no meio da tabela.

Para ver as outras funções de manipulação de tabelas, consulte o manual de referência de Lua, disponível em *http://www.lua.org/manual/5.1/*.

4.2 Descrição de Dados

Desde o princípio, Lua foi usada como uma linguagem de descrição de dados. Para ilustrar este uso de tabelas, vamos criar um registro com informações sobre diversos países:

```
entry {
  name = "Brasil",
  capital = "Brasília",
  netdomain = "br",
  population = 186757608
}

entry {
  name = "Estados Unidos",
  capital = "Washington DC",
  netdomain = "us",
  population = 304385000
}

entry {
  name = "Argentina",
  capital = "Buenos Aires",
  netdomain = "ar",
  population = 40677348
}
```

Em Lua, podemos chamar uma função sem colocar parênteses, dado que o argumento da função é uma tabela ou cadeia de caracteres. Vamos então usar essa característica da linguagem e definir uma função *entry*.

A função *entry* irá receber um argumento e inseri-lo em uma tabela, contendo o registro de todos os países:

```
function entry (e)
  table.insert (t, e)
end
```

Supondo que as informações sobre os países estejam no arquivo `paises.lua`, vamos usar a função *dofile*, que executa o código de um arquivo Lua. O nosso programa principal fica:

```
function entry (e)
  table.insert (t, e)
end

local t = {}
dofile ("paises.lua")   -- executa arquivo "paises.lua"
```

Ao executarmos o programa cima, cada entrada do arquivo `paises.lua` será interpretada como uma chamada de função que simplesmente insere o argumento fornecido em uma tabela.

Podemos, por exemplo, imprimir o nome dos países cuja população é maior do que 100 milhões de habitantes, como ilustrado a seguir, onde temos uma variável chamada _. Em Lua, quando não estamos interessados no valor de uma variável, é comum indicar isso nomeando-a simplesmente como _.

```
for _, v in pairs (t) do
  if v.population > 100000000 then
    print (v.name)
  end
end
```

A definição da função *entry* é um ponto essencial aqui. Essa função não precisa necessariamente inserir o argumento em uma tabela, ela pode realizar qualquer ação, como imprimir o nome da capital de um país.

```
function entry (e)
  print (e.capital)
end

dofile ("paises.lua")   -- executa arquivo "paises.lua", imprimindo o nome das capitais
```

Cuidados ao usar *dofile*

Devemos ter um cuidado especial ao usar a função *dofile*. Em geral, só devemos usar *dofile* quando temos certeza que iremos executar um arquivo Lua que não produzirá erros, caso contrário, todo o nosso programa Lua irá falhar.

Uma abordagem que permite tratar erros quando executando o código de um arquivo Lua é através do uso da função *loadfile*. A função *loadfile* executa em modo protegido, de modo que quando não há erros, o trecho de código Lua é compilado e retornado como uma função. Caso contrário, o valor de retorno é **nil** mais uma mensagem de erro.

Caso deseje utilizar *loadfile*, o trecho de código do exemplo anterior que usa *dofile* seria substituído por:

```
local f, s = loadfile ("paises.lua")  -- compila o arquivo "paises.lua"
if not f then
  print ("Erro ao compilar 'paises.lua'\n", s)
  return
end

f ()   -- Executa função retornada
```

4.3 Meta-métodos

Cada tabela pode ter associada a ela uma meta-tabela. Meta-tabelas permitem modificar o comportamento de um tabela, indicando qual ação deve ser tomada quando uma operação envolvendo uma tabela é realizada.

Para saber se uma tabela possui uma meta-tabela associada podemos chamar a função *getmetatable*:

```
local t = {}
print (getmetatable (t))  --> nil
```

Inicialmente, uma tabela não possui nenhuma meta-tabela associada. Para fazer isso, podemos usar a função *setmetatable*:

```
local meta = {}
local t = {}
setmetatable (t, meta)
print (meta, getmetatable (t))   --> table: 0x806f118   table: 0x806f118
```

Temos agora que *meta* é a meta-tabela de *t*.

É possível fazer algumas operações envolvendo tabelas, tais como $+$, $-$ e $<$, cujo significado é dado através de meta-tabelas. Considere o seguinte trecho de código Lua:

```
local t1, t2 = {}, {}
t1.x = 20  t1.y = 30
t2.x = 10  t2.y = 5
t1 = t1 + t2
```

Ao tentarmos executá-lo, iremos obter um erro, pois as tabelas *t1* e *t2* não possuem uma meta-tabela associada indicando qual é o comportamento da tabela quando a operação $+$ é realizada. Como as tabelas não possuem uma operação de soma específica, Lua tenta realizar a operação de soma usual, a qual irá falhar.

Vamos então definir uma meta-tabela e definir o seu campo *__add*, de modo a realizar corretamente a operação $+$ envolvendo tabelas:

```
local t1, t2 = {}, {}
local meta = {}

setmetatable (t1, meta)
setmetatable (t1, meta)

meta.__add = function (a, b)
  local c = {}
  setmetatable (c, getmetatable (a))
  c.x = a.x + b.x
  c.y = a.y + b.y
  return c
```

```
end

t1.x = 20  t1.y = 30
t2.x = 10  t2.y = 5
t1 = t1 + t2
print (t1.x, t1.y)    --> 30   35
```

Agora, quando a operação + é realizada, Lua verifica se a tabela *t1* possui uma meta-tabela associada. Em caso afirmativo, Lua procurará por um campo *__add* nessa meta-tabela, chamando-o, caso ele exista, para realizar a operação +.

Outra operação que pode ter seu comportamento modificado com o uso de meta-métodos é *print*, através da definição do campo *__tostring* da meta-tabela. No exemplo anterior, poderíamos definir esse campo da seguinte forma, onde .. é o operador de Lua que concatena cadeias de caracteres:

```
meta.__tostring = function (a)
  return "x = " .. a.x .. ", y = " .. a.y
end

t1.x = 20  t1.y = 30
t2.x = 10  t2.y = 5
t1 = t1 + t2
print (t1)          --> x = 30, y = 35
```

Para tratar elementos não existentes em tabelas podemos usar os meta-métodos *index* e *newindex*. O meta-método *index* é chamado sempre que tentamos acessar um campo ausente de uma tabela, ao passo que o meta-método *newindex* é chamado sempre que atribuímos um valor a um campo ausente de uma tabela.

Tomando como base o exemplo anterior, vamos definir um meta-método *index*:

```
meta.__index = function (t, k)    -- t é a tabela que estamos acessando
  print ("Acessando elemento não inicializado " .. k)
  return 0
end

t1 = {x=20, y=5}
t2 = {x=30}       -- não definimos um valor para o campo "y" de t2

setmetatable (t1, meta)
setmetatable (t2, meta)

t1 = t1 + t2      --> Acessando elemento não inicializado y
                  -- Quando tentamos acessar t2.y, o meta-método index retorna 0
print (t1)        --> x = 50, y = 5
```

Note que sempre que criamos uma nova tabela precisamos dizer quem é a sua meta-tabela, caso contrário a tabela não terá nenhuma meta-tabela associada. Se no meta-método *__add*, por exemplo, não tivéssemos atribuído a *c* a mesma meta-tabela de *a*, a função *print* não usaria o meta-método *__tostring* definido anteriormente.

4.4 Programação OO em Lua

Lua não é uma linguagem orientada a objetos, mas provê mecanismos que tornam possível a programação orientada a objetos (POO) (há, inclusive, mais de uma maneira de programar orientado a objetos em Lua).

A idéia central da POO em Lua é o uso de protótipos. Podemos entender uma meta-tabela como um protótipo de uma classe. Toda vez que fazemos uma chamada a um objeto, caso o objeto não implemente aquele método, o campo *__index* da sua meta-tabela será avaliado. Em geral, iremos utilizar um modelo OO onde o campo *__index* da meta-tabela será a própria meta-tabela. Quando o campo *__index* é uma tabela, ao invés de uma função, Lua tenta acessar o chave ausente nessa tabela. Dessa forma, quando um método não for implementado por um objeto, iremos procurar pela implementação dele na sua meta-tabela.

Para mostrar como podemos fazer isso, vamos definir uma classe *Rectangle*, para representar retângulos. Podemos definir uma estrutura básica para a nossa classe e um construtor da seguinte forma:

```
Rectangle = {
  width = 0,
  height = 0
}

function Rectangle.new (self, o)
  o = o or {}
  setmetatable (o, self)
  self.__index = self
  return o
end
```

Primeiro, definimos que *Rectangle* possui dois campos: *width* e *height*, ambos inicializados com 0; em seguida, definimos o método *new*, que deverá criar novos retângulos. Este método possui dois parâmetros: o primeiro parâmetro, *self*, deve ser um objeto *Rectangle*; ao passo que o segundo parâmetro contém a definição de um retângulo (os valores dos seus campos *width* e *height*). Caso o segundo parâmetro não seja fornecido, um objeto vazio é criado.

Em seguida, estabelecemos *self* como a meta-tabela de *o* e atribuímos *self* a *self.__index*, de modo que quando tentamos acessar um campo ausente no objeto *o*, iremos procurar pelo campo em *self*. No caso de não fornecemos o parâmetro *o* para o método *new*, ao tentarmos acessar o campo *width* do novo objeto, iremos na verdade acessar o campo *width* da sua meta-tabela, que no caso é 0.

Podemos evitar o uso explícito do parâmetro *self* usando a notação *Rectangle:new*, ao invés de *Rectangle.new*. Vamos definir mais um método, que calcula a área de um retângulo, já fazendo uso da nova notação.

```
function Rectangle:area ()
  return self.width * self.height
end
```

Ao usarmos a notação de dois pontos, Lua automaticamente cria um variável *self* que referencia o objeto que realizou a chamada ao método.

A seguir, temos um exemplo de uso de objetos *Rectangle*. Note que as notações de ponto e de dois pontos também se aplicam para a chamada de métodos e podem ser usadas livremente, métodos declarados usando *:* podem ser chamados com a outra notação e vice-versa.

```
local ret1 = Rectangle:new {height=6, width=4}
local ret2 = Rectangle.new (Rectangle)

print (ret1:area ())      --> 24
print (ret1.area (ret1))  --> 24
print (ret2:area ())      --> 0
```

4.5 Herança

Com o modelo de objetos que usamos anteriormente, é muito fácil estender/modificar a classe *Rectangle*. Suponha que queremos definir uma nova classe, *Cuboid*, para representar um paralelepípedo retangular. Nesse caso, cada objeto terá agora um campo *depth*, representando a profundidade do objeto, e teremos também que redefinir o método que calcula a área e adicionar um método para calcular o volume do cubóide. Temos a seguir a definição de *Cuboid*:

```
Cuboid = Rectangle:new ({depth=0})

function Cuboid:area ()
  return 2 * (self.width * self.height + self.width * self.depth +
              self.depth * self.height)
end

function Cuboid:volume ()
  return self.width * self.height * self.depth
end

local c1 = Cuboid:new ()
print (c1:area ())     --> 0
local c2 = Cuboid:new {width=1, height=2, depth=5}
print (c2:area ())     --> 34
print (c2:volume ())  --> 10
```

Ao criarmos *c1*, usaremos todos os valores padrões que definimos para *width*, *height* e *depth*. Ao tentarmos acessar o campo *depth* do objeto *c1*, estamos na verdade acessando o campo *depth* de *Cuboid*, já que *c1* não definiu um campo *depth*. Ao tentarmos acessar os campos *height* e *width* de *c1*, estamos na verdade acessando os campos *width* e *height* de *Rectangle*, já que nem *c1* nem *Cuboid* definiram estes campos.

O objeto *c2*, por sua vez, define valores para os campos *width*, *height* e *depth*. Devemos notar que quando chamamos *Cuboid:new*, na verdade estamos chamando *Rectangle:new*, uma vez que *Cuboid* não definiu um método *new*.

Tanto *c1* como *c2* estão usando o método *area* da classe *Cuboid* para calcular a sua área.

Capítulo 5

Módulos

O sistema de módulos de Lua foi padronizado com o surgimento da versão 5.1, em 2006. Embora módulos já existissem e fossem usados há algum tempo pelos desenvolvedores, não existia ainda uma política bem definida para módulos.

Segundo a definição da segunda edição de *Programming in Lua*, um módulo é uma biblioteca que pode ser carregada usando-se **require** e que define um único nome global que armazena uma tabela. Todas as funções e variáveis do módulo estão contidas nesta tabela, de modo que podemos dizer que um módulo Lua nada mais é do que uma tabela. Sendo assim, módulos em Lua são valores de primeira classe também e nenhum mecanismo especial precisa ser adicionado à linguagem para que ela dê suporte a módulos.

Ao fazer **require** de um módulo, também espera-se que o módulo retorne a tabela com as suas funções e variáveis exportadas. Dessa forma o seguinte trecho de código carrega um módulo e guarda uma referência para a tabela correspondente em uma variável local:

```
local math = require 'math'
```

Podemos também armazenar as funções que um módulo exporta em variáveis locais:

```
require 'math'  --carrega módulo math
local sin = math.sin
local sqrt = math.sqrt
```

Os seguintes módulos fazem parte da biblioteca padrão de Lua:

- **coroutine**: possui as operações relacionadas com co-rotinas;
- **string**: contém funções que manipulam cadeias de caracteres;
- **table**: manipulação de tabelas;
- **math**: módulo com as funções matemáticas;
- **io**: biblioteca de entrada e saída (E/S);

- **package**: biblioteca de módulos. Duas funções dessa biblioteca **require** e **module** são exportadas diretamente no ambiente global de Lua;
- **os**: implementa facilidades do sistema operacional;
- **debug**: biblioteca de depuração.

O interpretador `lua` importa automaticamente todos os módulos da biblioteca padrão, de modo que não é preciso fazer *require 'math'*, por exemplo, para poder usar a biblioteca matemática.

5.1 Definindo um Módulo

Assim como **require** é uma função indispensável para quem deseja usar um módulo, a função **module** foi criada para facilitar o trabalho dos desenvolvedores de módulos. Embora seja possível definir módulos sem usar **module**, é preferível que se use a função **module**, pois ela simplifica a definição de um módulo e é uma solução padrão fornecida pela linguagem.

A maneira mais simples de declarar um módulo Lua (e também a mais usual) é colocando a seguinte declaração no começo do seu módulo:

```
module (..., package.seeall)
```

O primeiro parâmetro de **module** é o nome do módulo. Podemos então escrever explicitamente o nome do módulo ou simplesmente colocar ..., indicando que o módulo terá o mesmo nome do arquivo no qual ele está sendo definido. Nesse caso, se temos um arquivo `bola.lua`, o nome do nosso módulo será *bola*.

Ao criarmos um módulo usando **module**, estamos definindo um novo ambiente para as variáveis do módulo e suas funções. Esse ambiente não contém nenhuma biblioteca padrão ou função de Lua. Dessa forma, se quisermos usar uma função de Lua como *print*, por exemplo, devemos declarar uma variável local com o valor dessa função antes de chamar **module**.

Uma maneira fácil de importar todas as funções e bibliotecas padrões de Lua é fornecendo *package.seeall* como segundo parâmetro para **module**.

Na figura 5.1 temos a definição de um módulo simples, que representa uma fração armazenando o seu numerador e o seu denominador. Supondo que a definição foi salva em um arquivo `fraction.lua`, o nome do nosso módulo será *fraction*. Temos a seguir um trecho de código Lua que importa o módulo *fraction* e usa as suas funcionalidades (o arquivo com o código abaixo deve estar no mesmo diretório onde está o arquivo `fraction.lua`):

```
require 'fraction'

local x = fraction.new (1, 2)
local y = fraction.new (5, 3)

fraction.print (x)                    --> 1 / 2
```

```
module (..., package.seeall)

function new (n, d)
  return {n=n, d=d}
end

-- Usamos "local" para declarar uma função que não é exportada pelo módulo
-- Calcula o maior divisor comum
local function gcd (a, b)
  if b == 0 then
    return a
  else
    return gcd (b, a % b)
  end
end

-- Calcula o menor múltiplo comum
local function lcd (a, b)
  return (a * b) / gcd (a, b)
end

function add (f1, f2)
  local d = lcd (f1.d, f2.d)
  local n = (f1.n * d / f1.d) + (f2.n * d / f2.d)
  return new (n, d)
end

function sub (f1, f2)
  return add (f1, new (-f2.n, f2.d))
end

function mul (f1, f2)
  return new (f1.n * f2.n, f1.d * f2.d)
end

function div (f1, f2)
  return mul (f1, new (f2.d, f2.n))
end

function print (f)
  -- Temos que referenciar a função "print" global explicitamente, caso
  -- contrário estaríamos acessando recursivamente a função "print" do módulo
  _G.print (f.n .. ' / ' .. f.d)
end
```

Figura 5.1: Definição do módulo *fraction*

```
fraction.print (y)                    --> 5 / 3

fraction.print (fraction.add (x, y))  --> 13 / 6

local z = fraction.sub (y, x)

fraction.print (z)                    -->  7 / 6

fraction.print (fraction.mul (y, z))  --> 35 / 18
```

5.2 Entendendo require

Quando chamada, a função **require** segue alguns passos para tentar encontrar o módulo desejado.

Primeiro, **require** tenta achar um módulo Lua com o nome fornecido, caso não encontre, tentará então achar um módulo C correspondente.

Quando procura por um módulo Lua, **require** usa a variável de ambiente *LUA_PATH* ou um caminho pré-definido durante a compilação de Lua. Em sistemas Unix, uma seqüência padrão usada por **require** quando importando um módulo é primeiro procurá-lo no diretório atual e depois nos diretórios `/usr/local/share/lua/5.1/` e `/usr/local/share/lua/5.1/nomedomodulo/init.lua`. Usando a variável *LUA_PATH*, poderíamos especificar esta seqüência de busca da seguinte forma, onde os pontos de interrogação são trocados pelo nome do módulo que se está procurando

```
?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua
```

Caso a função **require** não encontre um módulo Lua, ela irá procurar por um arquivo *.so* (ou *.dll* no caso do Windows) no diretório atual e no diretório `/usr/local/lib/lua/5.1`. Neste caso, a variável usada por **require** é *LUA_CPATH*, a qual poderia ser configurada da seguinte forma:

```
?.so;/usr/local/lib/lua/5.1/?.so
```

Quando um módulo é carregado, uma entrada correspondente é atualizada na tabela *package.loaded*, de modo que ao tentarmos carregar um módulo de novamente, a entrada armazenada em *package.loaded* é retornada. Para forçar que um módulo seja carregado novamente, podemos fazer:

```
package.loaded['meuModulo'] = nil
require 'meuModulo'
```

5.3 Instalando Módulos

Uma maneira básica de instalar um módulo é simplesmente copiando o arquivo `.lua`, `.so` ou `.dll` para um diretório onde o interpretador de Lua poderá encontrá-lo.

Outra abordagem é modificar as variáveis *LUA_PATH* e *LUA_CPATH* de modo que elas apontem para o diretório onde o seu módulo se encontra.

Uma abordagem mais recente e que facilita a vida de usuários e desenvolvedores de módulos Lua é utilizar LuaRocks[1]. LuaRocks é um sistema de instalação e gerenciamento de módulos Lua.

LuaRocks é baseado no conceito de *rocks*. Podemos entender um *rock* como um módulo Lua. Associado a um rock temos o seu *rockspec*, que é um arquivo que contém informações sobre a instalação do módulo, documentação e dependências do mesmo.

Para instalar um módulo usando LuaRocks, tudo que temos que fazer é:

```
luarocks install nomedomodulo
```

Em seguida, para utilizar o módulo instalado dessa forma em um programa Lua, devemos usar *luarocks.require*:

```
require 'luarocks.require'
```

E depois utilizar *require* normalmente.

5.4 Entendendo um rockspec

Para um usuário de módulos, um arquivo rockspec pode ser visto como uma caixa preta. Em geral, um usuário de módulos não precisará entender ou modificar um rockspec. Quando for necessário fornecer o valor de alguma variável de configuração, de forma que ela reflita o local correto no qual uma determinada biblioteca foi instalada no seu sistema, é preferível que se faça esta operação via linha de comando (tente *luarocks help* e veja as opções disponíveis), do que através da modificação de um rockspec.

Para um desenvolvedor de módulos, por outro lado, é fundamental que ele entenda um rockspec, de forma a poder escrever a especificação do seu módulo, facilitando a vida de pessoas que desejam utilizá-lo.

Na figura 5.2, podemos ver o exemplo do rockspec de um módulo Lua. O nome do pacote é *Shake* e a versão é 1.0.1 (o sufixo -1 indica apenas a versão do rockspec, não do pacote em si). Em seguida, temos a indicação de onde o código fonte do módulo está disponível. LuaRocks irá então procurar pelo módulo no endereço fornecido, fazer o download, compilar os arquivos e instalar o pacote.

A variável *description* possui um descrição do módulo, indicando também qual é a sua licença. A seguir, temos a variável *dependencies*, que lista as dependências de um módulo. Shake possui três dependências: a versão 5.1 de Lua (ou alguma versão mais nova); a versão 0.1.2 de *leg*; e a versão 1.3.0 de *luafilesystem*. Tirando a dependência de Lua 5.1, todas as outras dependências, que possuem rocks no repositório de LuaRocks, serão instaladas por LuaRocks, que irá, se necessário, instalar também as dependências de *leg* e *luafilesystem*.

[1]Para uma referência mais completa sobre LuaRocks, por favor consulte a página da ferramenta em http://www.luarocks.org/

```
package = "Shake"
version = "1.0.1-1"
source = {
  url = "http://luaforge.net/frs/download.php/3009/shake-1.0.1.tar.gz"
}
description = {
  summary = "A Simple Lua Test Engine",
  detailed = [[
      Shake is a simple and transparent test engine for Lua that assumes
      that tests only use standard assert and print calls.
      Shake uses Leg and LPeg to preprocess test files and extract a lot
      more information than what is usually available when tests use
      standard Lua assertions.
  ]],
  license = "MIT/X11",
  homepage = "http://shake.luaforge.net/"
}
dependencies = {
  "lua >= 5.1",
  "leg >= 0.1.2",
  "luafilesystem >= 1.3.0",
}
build = {
  type = "make",
  variables = {
      LUA_DIR = "$(LUADIR)",
      SYS_BINDIR = "$(BINDIR)"
  }
}
```

Figura 5.2: Rockspec do módulo Shake

Caso o módulo que você queira instalar seja uma ligação (*binding*) para uma biblioteca C externa, você deve primeiro instalar a biblioteca C antes de usar o LuaRocks para instalar o módulo que depende dela.

A última parte do rockspec contém a descrição da variável *build*, que irá dizer ao LuaRocks como o módulo deve ser instalado. O campo *type* indica que ferramenta devemos usar para construir o módulo: *make*, *cmake* ou *module* (no caso de *module*, LuaRocks usará seu próprio sistema para instalação de módulos).

Caso deseje obter uma explicação mais detalhada sobre os demais campos do rockspec, por favor consulte o site do LuaRocks.

Capítulo 6

API C

Atualmente, é muito comum desenvolvermos partes de uma aplicação utilizando uma linguagem de script e outra parte utilizando uma linguagem de mais baixo nível que possua um melhor desempenho.

Em 90% dos casos, a linguagem de mais baixo nível escolhida é C, de modo que várias linguagens de script como Python, Perl e Ruby possuem alguma forma de se comunicar com C.

Lua não é diferente e disponibiliza uma API para que o código Lua possa se comunicar com código C e vice-versa.

Lua é uma linguagem extensível pois podemos definir funções em C e usá-las em programas Lua. Ao mesmo tempo, Lua também é uma linguagem de extensão, uma vez que é possível embutir Lua em uma aplicação C.

O modelo para se comunicar com código C oferecido por Lua é baseado em uma pilha virtual, onde tanto o código C como o código Lua podem colocar e retirar valores na pilha.

As funções que manipulam a pilha virtual possuem como parâmetro uma estrutura *lua_State*, que armazena informações sobre o estado do programa Lua. Nenhuma informação sobre o estado é armazenada em variáveis globais, de modo que podemos ter vários *lua_State* independentes.

6.1 Primeiros Passos

Para termos acesso ao conjunto de funções que fazem a comunicação entre Lua e C devemos incluir alguns arquivos de cabeçalho em nosso programa C. O arquivo `lua.h` é onde estão as funções básicas fornecidas por Lua para manipular a pilha virtual. O nome das funções relacionadas com esse arquivo começa com *lua_*.

O arquivo `lauxlib.h` define algumas funções de mais alto nível usando a API básica fornecida por `lua.h`. As funções disponíveis em `lauxlib.h` não acessam nenhum valor interno de Lua, elas são implementadas usando somente as funcionalidades providas pela API básica de comunicação com C. O nome das funções relacionadas com `lauxlib.h` possuem o prefixo *luaL_*.

```
01 #include <stdio.h>
02 #include <string.h>
03 #include "lua.h"
04 #include "lauxlib.h"
05 #include "lualib.h"
06
07 int main () {
08   char buff [256];
09   int error;
10   lua_State *L = luaL_newstate ();  /* cria um novo estado Lua */
11   luaL_openlibs (L);    /* dá acesso a todas as bibliotecas padrões */
12
13   while (fgets (buff, sizeof (buff), stdin) != NULL) {
14     error = luaL_loadbuffer (L, buff, strlen (buff), "line") ||
15             lua_pcall (L, 0, 0, 0);
16     if (error) {
17        fprintf (stderr, "%s", lua_tostring (L, -1));
18        lua_pop (L, 1);   /* retira a mensagem de erro da pilha */
19     }
20  }
21
22  lua_close (L);
23  return 0;
24 }
```

Figura 6.1: Embarcando o interpretador Lua em um programa C

O arquivo `lualib.h`, por sua vez, dá acesso às bibliotecas padrões de Lua (*math*, *table*, *io*, etc) e deve ser incluído caso se queira ter acesso a elas.

O exemplo apresentado na figura 6.1 aparece no livro *Programming in Lua* e ilustra como é possível embarcar o interpretador Lua em uma aplicação C.

Na linha 10, um novo estado Lua é criado através da função *luaL_newstate*. Essa é a única função da API C de Lua que não recebe como parâmetro um *lua_State*.

Em seguida, na linha 11, inicializamos todas as bibliotecas padrões. É possível também inicializar cada biblioteca individualmente através de funções específicas, como *luaopen_math* e *luaopen_table*, por exemplo.

O laço das linhas $13 - 20$ lê um texto da entrada padrão e o passa para o interpretador Lua. A função *luaL_loadbuffer*, usada na linha 14, recebe um estado Lua e um buffer de dados e tenta compilar o buffer de dados como um trecho de código Lua (assim como a função *loadfile* vista anteriormente). O terceiro parâmetro de *luaL_loadbuffer* é o tamanho do buffer e o quarto é o nome que queremos dar ao trecho de código, de modo que esse nome irá aparecer em mensagens de erro e será útil também para propósitos de depuração.

Caso nenhum erro ocorra durante a execução de *luaL_loadbuffer*, o trecho de código Lua compilado será colocado no topo da pilha e o valor 0 será retornado. Caso contrário, um valor diferente de zero será retornado, junto com uma mensagem de erro.

Supondo que nenhum erro ocorreu, a função *lua_pcall*, na linha 15, será então chamada. *lua_pcall* executa uma função em modo protegido. De acordo com o protocolo da pilha virtual de Lua, devemos primeiro colocar na pilha a função que queremos executar e em seguida os seus argumentos (o primeiro argumento é colocado primeiro, depois o segundo e assim por diante).

O segundo parâmetro de *lua_pcall* serve então para indicar quantos argumentos a função possui. No nosso caso, temos um trecho de código Lua que foi retornado como uma função por *luaL_loadbuffer*, sendo que a função retornada não possui nenhum parâmetro.

O terceiro parâmetro de *lua_pcall* indica o número de valores de retorno da função. Os valores de retorno da função serão deixados na pilha. Como a nossa função não retorna nada, fornecemos um valor 0 também para o terceiro parâmetro de *lua_pcall*.

O quarto e último parâmetro é o índice na pilha virtual de uma função que possa tratar um possível erro durante a execução da função. Ao fornecermos um valor 0 para este parâmetro, faremos com que uma possível mensagem de erro seja deixada na pilha.

Na linha 17, caso tenha ocorrido um erro, iremos imprimir a mensagem de erro que foi deixada na pilha. Existem duas maneiras de se acessar a pilha, a partir do topo ou a partir da sua base. Para acessarmos a pilha a partir do topo, usamos índices negativos: -1 é o topo da pilha, -2 é o índice do elemento logo abaixo e assim por diante. Para acessarmos a base da pilha, usamos índices positivos: 1 é o elemento mais em baixo na pilha, 2 é o índice do elemento logo acima e assim por diante.

A função *lua_tostring*, usada na linha 17, está acessando o elemento no topo da pilha (a mensagem de erro) e convertendo-o para uma cadeia de caracteres. *lua_tostring* espera que o valor no topo da pilha seja uma cadeia de caracteres ou um número. Essa função retorna um ponteiro para uma cadeia dentro do estado Lua. É importante tomar cuidado com possíveis usos desse ponteiro, pois como Lua realiza coleta de lixo automática, não temos garantias de que o ponteiro retornado ainda será válido após tirarmos o valor da pilha. Em virtude disso, sempre é bom copiarmos a cadeia para a qual o ponteiro retornado por Lua aponta, ao invés de utilizar o ponteiro diretamente.

A linha 18 apenas retira a mensagem de erro da pilha. O segundo parâmetro de *lua_pop* indica quantos valores queremos retirar da pilha.

Por fim, na linha 22, fechamos o nosso estado Lua, liberando toda a memória dinâmica usada pelo estado. Às vezes não é necessário chamar essa função, uma vez que algumas plataformas liberam a memória alocada dinamicamente por um processo quando ele termina.

Tente compilar o programa acima e fornecer algumas linhas de código Lua para o interpretador. Em um ambiente Unix, o programa acima pode ser compilado da seguinte forma:

```
gcc interpretador.c -llua -lm -ldl
```

Onde *-lm* é necessário para fazer a ligação das funções da biblioteca matemática de C e *-ldl* para possibilitar o carregamento dinâmico de módulos.

O exemplo acima assume que o seu compilador sabe onde procurar os arquivos cabeçalho de Lua e a sua biblioteca dinâmica. Caso este não seja o caso, você deve indicar o caminho explicitamente, como mostrado a seguir:

```
gcc -I/home/user/lua/include/ interpretador.c -L/home/user/lua/lib/ -llua -lm -ldl
```

6.2 Funções Básicas

Como a API C de Lua consiste basicamente na manipulação de uma pilha virtual, estaremos todo o tempo realizando operações como colocar elementos na pilha, desempilhar elementos e verificar o tipo de um determinado elemento.

Para a tarefa básica de empilhar elementos na pilha, Lua oferece um conjunto de funções, cada uma especializada em empilhar um valor de um determinado tipo na pilha. A seguir, temos uma lista dessas funções:

```
void lua_pushnil      (lua_State *L);
void lua_pushboolean  (lua_State *L, int bool);
void lua_pushinteger  (lua_State *L, lua_Integer n);
void lua_pushnumber   (lua_State *L, lua_Number n);
void lua_pushlstring  (lua_State *L, const char *s, size_t len);
void lua_pushstring   (lua_State *L, const char *s);
```

A diferença entre as funções *lua_pushlstring* e *lua_pushstring* é que a primeira espera que o tamanho da cadeia seja indicado, pois a cadeia pode conter \0s dentro dela ou não ser terminada por \0. Já a função *lua_pushstring*, espera que a cadeia seja terminada por \0.

Quando colocamos uma cadeia na pilha, Lua faz uma cópia da cadeia. Dessa forma, podemos modificar a cadeia ou liberar a memória para a qual a variável *s* aponta, sem alterar com isso a cadeia que colocamos na pilha.

Quando queremos saber o tipo de um elemento da pilha, podemos usar as funções *lua_is**, que possuem a seguinte assinatura:

```
int lua_is* (lua_State *L, int index);
```

Onde *index* indica o índice do elemento na pilha. Essas funções sempre retornam 0 ou 1. As funções *lua_isnumber* e *lua_isstring* merecem uma nota especial. A primeira retorna 1 caso o elemento no índice *index* seja uma cadeia que pode ser convertida para um número, ao passo que a segunda sempre retorna 1 caso o elemento correspondente seja um número, uma vez que um número sempre pode ser convertido para uma cadeia.

É possível também converter elementos da pilha, através das funções *lua_to**. A seguir, listamos algumas delas:

```
int lua_toboolean (lua_State *L, int index);
lua_Integer lua_tointeger (lua_State *L, int index);
lua_Number lua_tonumber (lua_State *L, int index);
const char *lua_tolstring (lua_State *L, int index, size_t *len);
const char *lua_tostring (lua_State *L, int index);
```

A função *lua_toboolean* sempre retorna 0 ou 1, sendo que 0 é retornado somente quando o valor Lua correspondente é **false** ou **nil**.

Devemos tomar cuidado ao usar a função *lua_tointeger* quando o número não for inteiro. Nesse caso, o número será truncado de alguma maneira não especificada. Tanto *lua_tointeger* como *lua_tonumber* retornam 0 quando o elemento na posição *index* da pilha não pode ser convertido para um número.

As funções *lua_tolstring* e *lua_tostring* são bastante similares, sendo que a primeira também indica na variável *len*, caso ela não seja **NULL**, o tamanho da cadeia retornada. Sendo assim, *lua_tostring* é apenas uma forma abreviada de usarmos *lua_tolstring*, passando sempre **NULL** como o terceiro parâmetro. Caso o elemento no índice *index* não possa ser convertido para uma cadeia, o valor **NULL** será retornado.

Devemos lembrar que as funções de conversão apenas retornam um valor, elas não retiram o valor da pilha. Para isso, devemos fazer uma chamada à função *lua_pop*.

Uma variação das funções *lua_to** são as funções *luaL_check**. A diferença entre elas é que as funções *luaL_check** também verificam se um dado valor pode ser convertido. Podemos entender as funções *luaL_check** como uma junção de *lua_is** e *lua_to**. Seu uso, porém, é mais restrito, pois as funções *luaL_check** são usadas apenas na verificação dos argumentos de uma função, não operando, portanto, sobre índices da pilha.

6.3 Manipulando a Pilha

A API C de Lua também disponibiliza funções que manipulam a pilha e permitem realizar operações como copiar elementos, saber quantos elementos a pilha possui e modificar o valor de um dado elemento da pilha. Um conjunto dessas funções é apresentado a seguir:

```
int  lua_gettop (lua_State *L);
void lua_pop (lua_State *L, int n);
void lua_insert (lua_State *L, int index);
void lua_pushvalue (lua_State *L, int index);
void lua_remove (lua_State *L, int index);
void lua_replace (lua_State *L, int index);
void lua_settop (lua_State *L, int index);
```

A primeira função, *lua_gettop*, retorna o número de elementos na pilha, sendo que um valor de retorno 0 indica que a pilha está vazia.

A função *lua_pop*, como já vimos, desempilha n elementos da pilha.

Podemos mover um elemento na pilha através da função *lua_insert*, que recebe um índice na pilha e move o elemento no topo da pilha para esse índice, deslocando outros elementos para cima se necessário.

Já a função *lua_pushvalue*, copia o elemento em um dado índice para o topo da pilha, aumentando portanto o tamanho da pilha.

Para remover elementos da pilha, podemos usar a função *lua_remove*, que remove o elemento no índice fornecido, deslocando para baixo os elementos que se encontram acima. A função *lua_replace*, por sua vez, move o elemento no topo da pilha (desempilhando-o) para um índice específico, substituindo o valor que existia no índice anteriormente.

A função *lua_settop* pode tanto aumentar como diminuir o tamanho da pilha. Quando usamos essa função, estabelecemos o elemento no índice fornecido como o novo topo da pilha. Se o novo topo for maior do que o anterior, os novos elementos serão inicializados com **nil**. Caso o índice fornecido seja 0, todos os elementos da pilha serão removidos.

Na figura 6.2, podemos ver o exemplo de um programa que usa algumas das funções apresentadas anteriormente. A principal finalidade do programa é exercitar o uso das funções da API C de Lua.

Nas linhas $5-15$, temos a definição da função *printtypes*, que irá imprimir o tipo de cada elemento da pilha. Essa função recebe como único parâmetro um estado Lua.

Usando a função *lua_gettop* (linha 7), obtemos o número de elementos na pilha. Lembre-se que o elemento na base da pilha está no índice 1 e o elemento no topo no índice -1.

Em seguida (linhas $10-15$), temos um laço que obtém o tipo de cada elemento da pilha e o imprime. Para obter o tipo de um elemento na pilha usamos a função *lua_type*, que retorna um valor inteiro. Os valores de retorno para cada tipo estão definidos no arquivo `lua.h`. Quando o elemento é uma cadeia de caracteres, por exemplo, o valor retornado é *LUA_TSTRING*.

A função *lua_typename*, por sua vez, recebe um valor inteiro e retorna uma cadeia que corresponde ao nome de um tipo.

O nosso programa principal (linhas $17-38$) começa então definindo um novo estado Lua e empilhando três valores. Em seguida, na linha 24, chamamos a função *printtypes* pela primeira vez. O tipo dos elementos da pilha é listado a partir do elemento mais embaixo, de forma que a cadeia mais à direita (*string*) corresponde ao tipo do elemento no topo da pilha.

```
01 #include <stdio.h>
02 #include "lua.h"
03 #include "lauxlib.h"
04
05 static void printtypes (lua_State *L) {
06   int n = lua_gettop (L);
07   int i;
08   int type;
09
10   for (i=1; i<=n; ++i) {
11     type = lua_type (L, i);
12     printf ("%s  ", lua_typename (L, type));
13   }
14   printf ("\n");
15 }
16
17 int main () {
18   lua_State *L = luaL_newstate ();
19
20   lua_pushinteger (L, 51);   /* colocando uma boa idéia na pilha */
21   lua_pushnumber (L, 3.14);
22   lua_pushstring (L, "lua");
23
24   printtypes (L);   /*  number  number  string */
25
26   lua_settop (L, 5);
27   printtypes (L);   /* number  number  string  nil  nil */
28
29   lua_replace (L, 2);
30   printtypes (L);  /* number  nil  string  nil */
31
32   lua_pushboolean (L, 1);
33   lua_insert (L, -4);
34   printtypes (L);  /* number  boolean  nil  string  nil */
35
36   lua_close (L);
37   return 0;
38 }
```

Figura 6.2: Exemplo de uso das funções da API C de Lua

Capítulo 7

Usando Lua em uma Aplicação C

Como vimos, Lua tanto pode estender uma aplicação (linguagem de extensão), como pode pode ser estendida por funções definidas em C (linguagem extensível). Vamos começar usando Lua para estender uma aplicação C.

Quando estendendo uma aplicação C, é bastante comum usarmos Lua para escrever um arquivo de configuração.

O uso de Lua oferece várias vantagens. A primeira vantagem é que não precisaremos termos que nos preocupar em ler e entender o arquivo de configuração, Lua fará isso por nós. Outra vantagem é que, como já vimos, é muito fácil embarcar Lua em uma aplicação. Além disso, Lua ocupa pouquíssimo espaço, de forma que o tamanho da sua aplicação não irá crescer muito.

Devemos lembrar também que desde o princípio Lua foi pensada como uma linguagem de descrição de dados, de forma que não precisamos ter nenhuma preocupação excessiva com a sintaxe do nosso arquivo de configuração.

A seguir, apresentaremos alguns usos da API C de Lua, introduzindo também novas funções. Porém, não iremos apresentar todas as funções disponíveis da API. Caso você deseje saber mais detalhes sobre as funções disponíveis na API C de Lua, consulte o manual de referência de Lua.

7.1 Um arquivo de configuração simples

O primeiro arquivo de configuração que iremos utilizar é bastante simples. O nosso intuito é utilizar o arquivo de configuração para representar algumas características de um objeto.

Supondo que temos em C uma estrutura *Car*, representando um carro, podemos usar um arquivo de configuração para fornecer valores para algumas propriedades do carro, como velocidade, combustível e cor. O nosso primeiro arquivo de configuração é mostrado a seguir:

```
fuel = 43
speed = 100
color = "blue"
```

Agora, nosso programa C precisa carregar o arquivo de configuração e ler os valores que ele deseja. Para fazer isso, iremos definir a função *loadconfig*, que é mostrada na figura 7.1.

A função *loadconfig* recebe três parâmetros: um estado Lua, o nome do arquivo de configuração e um ponteiro para uma estrutura *Car*.

Na linhas 5 tentamos compilar o arquivo de configuração e executá-lo em caso de sucesso. Caso algum erro ocorra, imprimimos uma mensagem de erro e retornamos.

As linhas $11-13$ vão colocar na pilha o valor das variáveis *speed*, *fuel* e *color*. Caso essas variáveis não existam, um valor **nil** será empilhado, já que esse é o valor padrão de uma variável não declarada/inicializada.

Na linha 18, retornamos o elemento na posição -3 da pilha como um valor inteiro. Poderíamos também usar uma abordagem mais cuidadosa e testar se o valor pode ser convertido para um número, utilizando a função *lua_isnumber*. Lembre-se que o valor 0 será retornado quando o elemento na posição -3 não puder ser convertido para um número.

A linha 22 testa se o valor no topo da pilha pode ser convertido para uma cadeia de caracteres. Caso ele não possa, devemos tratar o erro, já que esperávamos uma cadeia.

Quando o valor no topo da pilha puder ser convertido para uma cadeia, executaremos o bloco *else* (linhas $24-29$). A função *lua_tolstring* retorna um ponteiro para a cadeia e atualiza a variável *len* com o tamanho da cadeia retornada. Note que estamos copiando a cadeia retornada e não guardando um ponteiro para ela. Lembre-se que caso o valor seja removido da pilha, o coletor de lixo de Lua pode tornar a referência inválida.

7.2 Incrementado o arquivo de configuração

Vamos colocar agora em nosso arquivo de configuração uma tabela Lua, com campos X, Y e Z, para indicar a localização do carro.

O novo arquivo de configuração pode ser visto a seguir:

```
fuel = 43
speed = 100
color = "blue"
pos = {X=100, Y=200, Z=100}
```

Temos agora que modificar a função *loadconfig*, de modo a obter o valor da variável global *pos* e em seguida o valor de cada um dos seus campos, como ilustra o código a seguir:

```
lua_getglobal (L, "pos");
```

```
01 void loadconfig (lua_State *L, const char *file, Car *car) {
02   const char *color;
03   size_t len;
04
05   if (luaL_loadfile (L, file) || lua_pcall (L, 0, 0, 0)) {
06     printf ("Erro lendo arquivo de configuração: %s\n", lua_tostring (L, -1));
07     return;
08   }
09
10   /* Coloca o valor da variável no topo da pilha */
11   lua_getglobal (L, "speed");
12   lua_getglobal (L, "fuel");
13   lua_getglobal (L, "color");
14
15   /* O valor de "color" está no topo da pilha e o valor de "speed" na base */
16
17   /* Inicia car->speed com 0 caso o elemento -3 da pilha não possa ser convertido */
18   car->speed = lua_tointeger (L, -3);
19
20   car->fuel = lua_tointeger (L, -2);
21
22   if (!lua_isstring (L, -1)) {
23     /* Trata o erro */
24   } else {
25     color = lua_tolstring (L, -1, &len);
26     car->color = malloc (len + 1);
27     /* Copia o valor da cor (não guarda o ponteiro) */
28     strcpy (car->color, color);
29   }
30 }
```

Figura 7.1: Usando um Arquivo de Configuração Simples

```
if (!lua_istable (L, -1) {
  /* Trata o erro */
} else {
  lua_getfield (L, -1, "X");
  car->pos.X = lua_tointeger (L, -1);
  lua_pop (1);
}
```

Após obtermos o valor da variável *pos*, verificamos se o valor que ela guarda é mesmo uma tabela. Caso seja, usamos a função *lua_getfield* para colocar na pilha o valor do seu campo *X*. A função *lua_getfield* possui como segundo parâmetro o índice na pilha de uma tabela e como terceiro parâmetro a chave da tabela que queremos acessar. O valor correspondente a aquela chave da tabela é então colocado na pilha.

Em seguida, usamos a função a *lua_tointeger* para obter o valor retornado como um número inteiro e a função *lua_pop* para retirar o valor da pilha. O próximo passo é realizar uma operação similar para obter o valor dos campos *Y* e *Z* da tabela.

Para evitar repetir o mesmo código várias vezes e dado que o acesso ao campo de uma tabela é uma tarefa muito comum, vamos criar uma função em C que faça isso. A definição da função *c_getfield* é mostrada a seguir:

```
01 int c_getfield (lua_State *L, const char *k) {
02   /* Assume que a tabela está no topo */
03   int v;
04
05   lua_getfield (L, -1, k);
06   if (!lua_isnumber (L, -1)) {
07     /* Trata o erro */
08   }
09   v = lua_tointeger (L, -1);
10   lua_pop (L, 1);
11
12   return v;
13 }
```

A nova definição da função *loadconfig* é mostrada na figura 7.2. Podemos notar que o código anterior da função, que acessava o valor das variáveis *speed*, *fuel* e *color*, não foi alterado.

7.3 Usando funções

Usando a API C de Lua podemos definir funções em Lua que serão chamadas pelo código C. No mesmo arquivo de configuração em que definimos o valor de algumas variáveis, iremos definir agora uma função *canachieve*. A função *canachieve* receberá como parâmetros a quantidade de gasolina que o carro

```
01 void loadconfig (lua_State *L, const char *file, Car *car) {
02   const char *color;
03   size_t len;
04
05   if (luaL_loadfile (L, file) || lua_pcall (L, 0, 0, 0)) {
06     printf ("Erro lendo arquivo de configuração: %s\n", lua_tostring (L, -1));
07     return;
08   }
09
10   /* Coloca o valor da variável no topo da pilha */
11   lua_getglobal (L, "speed");
12   lua_getglobal (L, "fuel");
13   lua_getglobal (L, "color");
14
15   /* O valor de "color" está no topo da pilha e o valor de "speed" na base */
16
17   /* Inicia car->speed com 0 caso o elemento -3 da pilha não possa ser convertido */
18   car->speed = lua_tointeger (L, -3);
19
20   car->fuel = lua_tointeger (L, -2);
21
22   if (!lua_isstring (L, -1)) {
23     /* Trata o erro */
24   } else {
25     color = lua_tolstring (L, -1, &len);
26     car->color = malloc (len + 1);
27     /* Copia o valor da cor (não guarda o ponteiro) */
28     strcpy (car->color, color);
29   }
30
31   lua_getglobal (L, "pos");
32   if (!lua_istable (L, -1)) {
33     /* Trata o erro */
34   } else {
35     car->pos.X = c_getfield (L, "X");
36     car->pos.Y = c_getfield (L, "Y");
37     car->pos.Z = c_getfield (L, "Z");
38   }
39
40 }
```

Figura 7.2: Nova definição de *loadconfig*

possui e uma distância em quilômetros que se deseja percorrer. O valor de retorno deve ser um booleano, indicando se a distância pode ser percorrida com aquela quantidade de gasolina. A definição de *canachieve* é dada a seguir:

```
function canachieve (fuel, dist)
  if dist / fuel > 10 then   -- muito longe, a gasolina não dá
    return false
  else
    return true
  end
end
```

Agora, supondo que o estado Lua já foi inicializado pelo código da nossa aplicação C e o arquivo de configuração foi executado, podemos usar *canachieve* seguindo um protocolo simples: a função deve ser colocada na pilha (do mesmo modo que fizemos com as variáveis *speed*, *color*, etc); em seguida, devemos colocar os valores dos parâmetros (o primeiro parâmetro primeiro, depois o segundo e assim por diante); usamos *lua_pcall* para realizar a chamada à função; e por último obtemos os resultados e os removemos da pilha.

A função *canachievec*, apresentada a seguir, é uma função C que usa a função *canachieve* definida em Lua:

```
01 int canachievec (lua_State *L, int fuel, int dist) {
02   int res;
03
04   lua_getglobal (L, "canachieve");
05
06   lua_pushinteger (L, fuel);
07   lua_pushinteger (L, dist);
08
09   if (lua_pcall (L, 2, 1, 0) != 0) {
10     printf ("Erro chamando 'canachieve': %s\n", lua_tostring (L, -1));
11   }
12
13   res = lua_toboolean (L, -1);
14   lua_pop (L, 1);
15   return res;
16 }
```

Na linha 4, colocamos na pilha a função *canachieve* e em seguida (linhas $6-7$) o valor dos seus parâmetros. O próximo passo é usar *lua_pcall* para chamar *canachieve* (linhas $9-11$). Como vimos antes, o segundo argumento de *lua_pcall* indica o número de parâmetros que a função que está sendo chamada recebe e o terceiro argumento indica o número de resultados que ela retorna. O quarto argumento indica o índice na pilha de uma função que irá tratar o erro, onde 0 indica que não usaremos nenhuma função, a mensagem de erro será deixada na pilha.

```
01 int main () {
02   Car *car = malloc (sizeof (Car));
03   lua_State *L = luaL_newstate ();
04   int d;
05
06   loadconfig (L, "config.lua", car);
07
08   printf ("speed = %d\n", car->speed);
09   printf ("fuel = %d\n", car->fuel);
10   printf ("color = %s\n", car->color);
11   printf ("pos.X = %d\n", car->pos.X);
12   printf ("pos.Y = %d\n", car->pos.Y);
13   printf ("pos.Z = %d\n", car->pos.Z);
14
15   d = canachievec (L, car->fuel, 40);
16   printf ("Achieve: %d\n", d);
17
18   d = canachievec (L, car->fuel, 430);
19   printf ("Achieve: %d\n", d);
20
21   d = canachievec (L, car->fuel, 440);
22   printf ("Achieve: %d\n", d);
23
24   return 0;
25 }
```

Figura 7.3: Exemplo de uso das funções *loadconfig* e *canachievec*

Se tudo ocorreu bem, guardamos o valor retornado por *canachieve* na variável *res*, removemos esse valor da pilha e retornamos o resultado. No caso de uma função que retorna mais de um resultado, o último valor retornado fica no topo da pilha. Dessa forma, se uma função retorna dois valores, o primeiro valor retornado ficará no índice -2 da pilha e o segundo (e último) no índice -1.

Um exemplo de programa *main* que utiliza as funções *loadconfig* e *canachievec*, definidas anteriormente, é apresentado na figura 7.3.

Capítulo 8

Usando Funções C em Lua

Do mesmo modo que a API C de Lua permite que funções Lua sejam usadas em código C, é possível fazermos chamadas a funções C em nosso programa Lua.

Para conseguir isso, devemos primeiro definir uma função C com a seguinte assinatura:

```
int myfunction (lua_State *L)
```

Toda função C que será usada em programas Lua deve seguir esse padrão: a função retorna um número inteiro e recebe como único parâmetro um *lua_State*.

Assim como fizemos anteriormente com funções Lua que eram usadas em C, devemos seguir um certo protocolo. Quando a função C é chamada por Lua, os parâmetros passados estarão disponíveis na pilha, com o primeiro elemento no índice 1, o segundo no índice 2 e assim por diante. Vamos então definir uma função *pow_l*, que realiza a operação de exponenciação:

```
01 static int pow_l (lua_State *L) {
02   double a = luaL_checknumber (L, 1);
03   double b = luaL_checknumber (L, 2);
04
05   lua_pushnumber (L, pow (a, b));
06
07   return 1;
08 }
```

Nas linhas $2 - 3$ obtemos o valor dos parâmetros usando *luaL_checknumber*, que lança um erro com uma mensagem de erro caso o elemento na posição correspondente da pilha não seja um número.

Em seguida, na linha 5, colocamos o resultado da função na pilha. O último passo é indicar quantos valores estamos retornando, isso é feito na linha 7, onde indicamos que um único valor está sendo retornado pela função. Os valores de retorno devem estar no topo da pilha no momento que a função retorna.

Agora que definimos a nossa função *pow_l*, devemos exportá-la, de modo que ela possa ser usada por programas Lua. Uma boa maneira de se fazer isso

é exportando as funções como um módulo Lua, usando um array de registro, onde listamos o nome pelo qual a função pode ser acessada em Lua e o nome da função correspondente em C, como mostrado a seguir:

```
static const struct luaL_Reg mylib [] = {
  {"pow", pow_l},
  {NULL, NULL}
};
```

Colocamos um par {*NULL, NULL*} para indicar o fim do array. Agora, basta definirmos uma função principal, que irá registrar o nosso módulo:

```
int luaopen_mylib (lua_State *L) {
  luaL_register (L, "mylib", mylib);
  return 1;
}
```

A função *luaL_register* recebe um array de registro e exporta as funções do array para uma tabela cujo nome é indicado pelo seu segundo parâmetro. Nesse caso, estamos criando uma tabela *mylib*, que terá uma função *pow*. Caso o nome fornecido para *luaL_register* fosse o de uma tabela Lua já existente, as função do nosso array de registro seriam adicionadas à tabela, que manteria as suas funções antigas. Poderíamos, por exemplo, exportar a nossa função *pow* na tabela global *_G*.

O próximo passo é compilar o nosso arquivo C como uma biblioteca dinâmica, de modo que Lua possa carregar dinamicamente as novas funções que definimos. Em sistemas Unix, uma solução possível seria usar a seguinte linha de compilação:

```
gcc -shared -o mylib.so mylib.c
```

Gerando assim um arquivo *.so*. Em um ambiente Windows, devemos gerar um arquivo equivalente, ou seja, um arquivo *.dll*.

Agora que já compilamos o nosso arquivo C como uma biblioteca dinâmica, tudo que temos que fazer em Lua é importar o módulo *mylib* que acabamos de definir:

```
require 'mylib'

print (mylib.pow (3, 4))   --> 81
print (mylib.pow (5, 3))   --> 125
```

Podemos notar que a função *pow* é utilizada da mesma forma que uma função declarada em Lua.

8.1 Definindo Funções com Estado

Da mesma forma que definimos fechos em Lua, também podemos definir em C funções que possuem um estado. A API C de Lua oferece três maneiras de fazer isso, que veremos a seguir.

8.1.1 Usando *registry*

A primeira maneira é através do uso da tabela global *registry*, que pode ser acessada somente pelo código C. A tabela *registry* está localizada em um pseudo-índice, isto quer dizer que a tabela *registry* não está realmente na pilha. Para acessar a chave *"bola"* dessa tabela, devemos usar o seguinte trecho de código:

```
lua_getfield (L, LUA_REGISTRYINDEX, "bola");
```

Ao invés de usarmos índices numéricos como 1, −2 ou 4, usamos o pseudo-índice *LUA_REGISTRYINDEX*. Ao contrário de outras tabelas de Lua, onde as chaves podem ser qualquer valor, você não deve usar chaves numéricas com a tabela *registry*, pois os resultados são imprevisíveis.

Como a tabela *registry* é global, ela será compartilhada por vários módulos. Se você deseja que funções de diferentes módulos acessem a mesma variável, então o uso de *registry* é aconselhável.

8.1.2 Usando o ambiente

Se o que você deseja é que várias funções do mesmo módulo utilizem a mesma variável, então você deve usar a facilidade de ambiente de funções. A idéia é fazer com que as funções de um módulo compartilhem o mesmo ambiente, usando o pseudo-índice *LUA_ENVIRONINDEX*. O seguinte trecho de código ilustra o uso de *LUA_ENVIRONINDEX*:

```
01 int luaopen_mylib (lua_State *L) {
02   lua_newtable (L);
03   lua_replace (L, LUA_ENVIRONINDEX);
04   luaL_register (L, "nomeDaLib", arrayDeRegistro);
05
06   lua_pushnumber (L, 16);
07   lua_setfield (L, LUA_ENVIRONINDEX, "bola");
08
09   return 1;
10 }
```

O primeiro passo foi criar uma nova tabela, que vai ficar no topo da pilha. Em seguida, usamos a função *lua_replace*, de modo que a tabela será colocada no pseudo-índice *LUA_ENVIRONINDEX* e irá servir como o ambiente a ser compartilhado pelas funções do módulo.

O próximo passo é registrar as funções do módulo, como fazemos na linha 4. Agora, podemos inicializar as variáveis que serão compartilhadas. Vamos ter somente a variável *bola*, cujo valor inicial é 16.

Para manipular a variável *bola*, vamos definir as funções *get_l* e *multiply_l*. A função *get_l* irá simplesmente retornar o valor associado com a variável *bola*, ao passo que *multiply_l* irá multiplicar por 2 o valor de *bola*. O programa completo é mostrado na figura 8.1.

```
#include "lua.h"
#include "lauxlib.h"
#include <math.h>

static int
get_l (lua_State *L) {
  lua_getfield (L, LUA_ENVIRONINDEX, "bola");
  return 1;
}

static int
multiply_l (lua_State *L) {
  int d;

  lua_getfield (L, LUA_ENVIRONINDEX, "bola");
  d = lua_tonumber (L, -1);
  lua_pop (L, 1);
  lua_pushnumber (L, d * 2);
  lua_setfield (L, LUA_ENVIRONINDEX, "bola");
  return 0;
}

static const struct luaL_Reg mylib [] = {
  {"get", get_l},
  {"multiply", multiply_l},
  {NULL, NULL}
};

int luaopen_mylib (lua_State *L) {
  lua_newtable (L);
  lua_replace (L, LUA_ENVIRONINDEX);
  luaL_register (L, "mylib", mylib);

  lua_pushnumber (L, 16);
  lua_setfield (L, LUA_ENVIRONINDEX, "bola");

  return 1;
}
```

Figura 8.1: Exemplo de uso de *LUA_ENVIRONINDEX*

8.1.3 Usando *upvalues*

Por fim, podemos fazer uso de *upvalues*, que são visíveis somente dentro de uma função. Quando criamos uma nova função C para ser usada por Lua, podemos associar a ela um número arbitrário de upvalues, formando um fecho.

Para mostrar o uso de upvalues, vamos definir uma versão mais simples da função *generateinc*, apresentada anteriormente. A nova versão de *generateinc* recebe apenas um parâmetro, o valor inicial do contador, que será sempre incrementado em uma unidade. A seguir, temos a definição da versão C de *generateinc*:

```
01 static int inc (lua_State *L) {
02   int d = lua_tointeger (L, lua_upvalueindex (1)); /* Obtém valor do upvalue */
03   lua_pushinteger (L, ++d);
04   lua_pushvalue (L, -1);                  /* Copia o topo da pilha */
05   lua_replace (L, lua_upvalueindex (1));  /* Atualiza o upvalue */
06   return 1;
07 }
08
09 static int generateinc_l (lua_State *L) {
10   int d = luaL_checkint (L, -1);  /* Recebe valor inicial do contador */
11   lua_pushinteger (L, d);
12   lua_pushcclosure (L, &inc, 1);  /* Cria um novo fecho, com 1 upvalue */
13   return 1;
14 }
```

Devemos começar a olhar a definição a partir da linha 9, onde declaramos a função *generateinc_l*. Na linha 10, obtemos o valor inicial do contador que foi passado na pilha. Esse valor será o upvalue do nosso fecho, sendo que todos os upvalues devem ser colocados na pilha antes do fecho. Então, na linha 11, empilhamos o upvalue e na linha 12 o fecho. O segundo parâmetro de *pushcclosure* é a função base do fecho e o terceiro parâmetro é o número de upvalues. Na linha 13, retornamos 1 para indicar que há um valor de retorno (o fecho).

Agora, devemos olhar a definição da função *inc*, nas linhas $1 - 7$. O valor do upvalue é obtido na linha 2, através da função *lua_upvalueindex*, e depois é empilhado acrescido de uma unidade na linha 3. Na linha 4, empilhamos uma cópia do valor no topo da pilha e a usamos para atualizar o valor do upvalue. A função *inc* retorna somente um resultado, o novo valor do upvalue.

Capítulo 9

Definindo Novos Tipos de Dados em C

Através da API C de Lua, podemos não somente definir novas funções para estender a linguagem, como também podemos definir novos tipos de dados que serão usados pela aplicação Lua.

Para definir um novo tipo de dado em C, iremos usar a função *lua_newuserdata*, que aloca um novo bloco de memória de um tamanho especificado. O bloco de memória criado representará um objeto do tipo *userdata* de Lua e pode ter a sua própria meta-tabela e ser recolhido pelo coletor de lixo.

Um objeto do tipo *userdata*, da mesma forma que uma tabela, somente é igual a ele mesmo.

Para ilustrar o uso de *userdata*, vamos definir em C uma estrutura que irá representar um buffer circular.

```
typedef struct CircBuffer {
  int start, end;
  int n, size;
  int array [1];
} CircBuffer;
```

As variáveis *start* e *end* guardam, respectivamente, as posições de início e fim do buffer. A variável *n* irá armazenar o número de elementos atualmente no buffer, ao passo que *size* representa o número máximo de elementos que o buffer pode armazenar. A variável *array* representa o buffer e será inicializada com um tamanho adequado quando o buffer for criado. Inicialmente *array* possui tamanho 1, uma vez que não é possível criar um array com tamanho 0 em C 89.

O trecho de código a seguir mostra a função que cria um novo buffer circular:

```
01 static int buffernew (lua_State *L) {
02   int n;
03   size_t bsize;
04   CircBuffer *cb;
```

```
05
06   n = luaL_checkinteger (L, 1);
07   luaL_argcheck (L, n >= 1, 1, "invalid buffer size");
08   bsize = sizeof (CircBuffer) + (n - 1) * sizeof (int);
09   cb = (CircBuffer *) lua_newuserdata (L, bsize);
10
11   cb->size = n;
12   cb->start = 0;
13   cb->end = 0;
14   cb->n = 0;
15
16   return 1;
17 }
```

A função *luaL_argcheck* é útil para gerar mensagens de erro sobre algum argumento da função. O segundo parâmetro de *luaL_argcheck* é a condição a ser testada, o terceiro parâmetro indica qual argumento está sendo testado e o último parâmetro especifica uma parte da mensagem de erro. Assumindo que *buffer.new* seja a função Lua equivalente para *buffernew*, no caso da chamada a seguir:

```
buffer.new (0)
```

teríamos a seguinte mensagem de erro:

```
stdin:1: bad argument #1 to 'new' (invalid buffer size)
```

Na função *buffernew*, após armazenarmos na variável n o tamanho do buffer circular, iremos calcular o tamanho, em bytes, do objeto userdata que iremos criar na linha 9. As linhas $11-14$ inicializam algumas variáveis do buffer circular e na linha 16 indicamos que *buffernew* retorna um elemento, o userdata que foi colocado na pilha por *lua_newuserdata*. O próximo passo é definir operações que operem sobre o buffer circular. Primeiro, vamos definir uma função que insere elementos no final do buffer:

```
01 static int bufferinsert (lua_State *L) {
02   CircBuffer *cb = (CircBuffer *) lua_touserdata (L, 1);
03
04   cb->array [cb->end] = luaL_checkinteger (L, 2);
05   cb->end = (cb->end + 1) % cb->size;
06
07   if (cb->n == cb->size)
08     cb->start = (cb->start + 1) % cb->size;
09   else
10     cb->n++;
11
12   return 0;
13 }
```

A primeira coisa que a função *bufferinsert* faz é converter o userdata que está na pilha para um ponteiro para uma estrutura C. Em seguida, na linha 4, obtemos o valor do elemento que desejamos inserir no buffer circular e na linha 5 atualizamos a variável que guarda o fim do buffer. Caso o buffer esteja cheio, então o começo do buffer deve ser incrementado, caso contrário, iremos incrementar o número de elementos do buffer circular. O próximo passo é definir uma função que remova os elementos do buffer, como mostra o trecho de código a seguir:

```
01 static int bufferremove (lua_State *L) {
02   CircBuffer *cb = (CircBuffer *) lua_touserdata (L, 1);
03
04   if (cb->n < 1)
05     return 0;
06
07   cb->n--;
08   cb->end--;
09   if (cb->end < 0)
10     cb->end += cb->size;
11
12   return 0;
13 }
```

Assim como *bufferinsert*, a função *bufferremove* começa convertendo o userdata para uma estrutura C. Em seguida, atualizamos algumas variáveis do buffer circular e retornamos 0, indicando que a função não deixou nenhum valor de retorno na pilha.

Agora, precisamos criar uma função que acesse os elementos do buffer circular. O trecho de código a seguir faz isso:

```
01 static int bufferelement (lua_State *L) {
02   CircBuffer *cb = (CircBuffer *) lua_touserdata (L, 1);
03   int i = luaL_checkinteger (L, 2);
04
05   luaL_argcheck (L, 1 <= i && i <= cb->n, 2, "index out of range");
06
07   lua_pushinteger (L, cb->array [(cb->start + i -1) % cb->size]);
08   return 1;
09 }
```

Na linha 5, usamos novamente a função *luaL_argcheck* para testar se o elemento que estamos tentando acessar está dentro dos limites do buffer circular. Optamos por indexar os elementos do buffer circular em Lua a partir do índice 1, de modo que precisamos subtrair uma unidade da variável i, na linha 7.

Por fim, iremos criar funções que retornam o tamanho atual do buffer e o tamanho máximo do mesmo, como ilustrado a seguir:

```
01 static int buffersize (lua_State *L) {
02   CircBuffer *cb = (CircBuffer *) lua_touserdata (L, 1);
```

```
03
04   lua_pushinteger (L, cb->n);
05   return 1;
06 }
07
08 static int buffermaxsize (lua_State *L) {
09   CircBuffer *cb = (CircBuffer *) lua_touserdata (L, 1);
10
11   lua_pushinteger (L, cb->size);
12   return 1;
13 }
```

O último passo é registrar as funções que manipulam o buffer circular, de modo que elas possam ser exportadas e usadas por Lua:

```
01 static const struct luaL_Reg bufferlib [] = {
02   {"new", buffernew},
03   {"insert", bufferinsert},
04   {"remove", bufferremove},
05   {"size", buffersize},
06   {"maxsize", buffermaxsize},
07   {"get", bufferelement},
08   {NULL, NULL}
09 };
10
11 int luaopen_buffer (lua_State *L) {
12   luaL_register (L, "buffer", bufferlib);
13   return 1;
14 }
```

Agora, podemos importar o módulo do buffer circular e utilizá-lo em um programa Lua:

```
require 'buffer'

b = buffer.new (3)

buffer.insert (b, 100)
buffer.insert (b, 2)
print ("size = ", buffer.size (b))

buffer.insert (b, 33)
buffer.remove (b)
buffer.insert (b, 44)

for i=1, buffer.size (b) do
  print (buffer.get (b, i))
end
```

9.1 Usando meta-tabelas

A implementação anterior do buffer circular possui um problema. Estamos sempre obtendo o valor de um userdata e o convertendo para uma estrutura *CircBuffer*, contudo, não estamos verificando se o valor que está na pilha é mesmo um userdata que representa um buffer circular.

Uma maneira de se fazer essa verificação é através do uso de meta-tabelas. A idéia é associar uma meta-tabela com objetos userdata que representam um buffer circular, de modo que depois, ao tentarmos converter um userdata na pilha, iremos verificar qual é a sua meta-tabela associada, assim saberemos se aquele userdata representa realmente um buffer circular.

O trecho de código a seguir mostra a nova implementação da função *luaopen_buffer*:

```
01 int luaopen_buffer (lua_State *L) {
02   luaL_newmetatable (L, "CircBuffer");
03   luaL_register (L, "buffer", bufferlib);
04   return 1;
05 }
```

Na linha 2, usamos a função *luaL_newmetatable*, que cria uma nova meta-tabela na tabela global *registry*. O nome da meta-tabela que passamos para *luaL_newmetatable* deve ser único, pois caso uma entrada com o mesmo nome já exista em *registry* a meta-tabela não será criada e o valor 0 será retornado. Caso a meta-tabela seja criada com sucesso, o valor 1 será retornado.

Agora, ao criar um novo objeto userdata que representa um buffer circular, devemos associá-lo com a meta-tabela *CircBuffer*. Precisamos então modificar a função *buffernew*, que terá o seu código a partir da linha 16 alterado para:

```
16   luaL_getmetatable (L, "CircBuffer");
17   lua_setmetatable (L, -2);
18
19   return 1;
20 }
```

Na linha 16, usamos a função *luaL_getmetatable*, que coloca na pilha a meta-tabela associada com *CircBuffer* na tabela *registry*. Em seguida, na linha 17, associamos o userdata recém criado, que está logo abaixo do topo da pilha, com a meta-tabela *CircBuffer*.

Uma vez que fizemos a associação entre os objetos userdata e a meta-tabela correspondente, devemos agora modificar as outras funções, de modo a verificar se o userdata que está na pilha representa mesmo um buffer circular. A seguir, mostramos a nova implementação da função *buffermaxsize*:

```
01 static int buffermaxsize (lua_State *L) {
02   CircBuffer *cb = (CircBuffer *) luaL_checkudata (L, 1, "CircBuffer");
03
04   lua_pushinteger (L, cb->size);
```

```
05   return 1;
06 }
```

A única mudança que fizemos foi na linha 2, onde estamos utilizando a função *luaL_checkudata*, que verifica se o userdata em uma determinada posição da pilha está associado com uma dada meta-tabela. Agora, tente executar o seguinte trecho de código e veja o que acontece:

```
require 'buffer'

b = buffer.new (3)
t = {}

print (buffer.maxsize (b))
print (buffer.maxsize (t))
```

9.2 Usando um modelo OO

Uma vez que temos uma meta-tabela associada com os objetos userdata, podemos fazer uso dela também para prover um acesso orientado a objeto para as funções do userdata.

Para conseguir isso, iremos usar o meta-método *__index* da meta-tabela associada com o buffer circular, de uma maneira similar ao que fizemos no capítulo em que abordamos programação OO em Lua.

Com o novo modelo, a única função a ser exportada será *buffernew*, de modo que iremos alterar o nosso array de registro, como ilustrado a seguir:

```
01 static const struct luaL_Reg bufferlib_f [] = {
02   {"new", buffernew},
03   {NULL, NULL}
04 };
05
06 static const struct luaL_Reg bufferlib_m [] = {
07   {"insert", bufferinsert},
08   {"remove", bufferremove},
90   {"size", buffersize},
10   {"maxsize", buffermaxsize},
11   {"get", bufferelement},
12   {NULL, NULL}
13 };
```

Agora, vamos modificar *luaopen_buffer*:

```
01 int luaopen_buffer (lua_State *L) {
02   luaL_newmetatable (L, "CircBuffer");
03
04   lua_pushvalue (L, -1);  /* duplica a meta-tabela */
```

```
05   lua_setfield (L, -2, "__index");  /* CircBuffer.__index = CircBuffer */
06
07   luaL_register (L, NULL, bufferlib_m);
08
09   luaL_register (L, "buffer", bufferlib_f);
10   return 1;
11 }
```

Na linha 7, ao fornecermos *NULL* como o segundo parâmetro de *luaL_register*, estamos inserindo as funções do array de registro *bufferlib_m* na tabela que está no topo da pilha, que no caso é a meta-tabela *CircBuffer*.

Na linha 9, exportamos as funções do array de registro *bufferlib_f* para a tabela *buffer*, ou seja, estamos exportando para Lua apenas a função *buffernew* do buffer circular.

O trecho de código a seguir mostra a utilização do buffer circular em Lua usando uma notação OO:

```
require 'buffer'

b = buffer.new (2)
print (b:size())

b:insert (99)
b:insert (44)
b:insert (55)
b:remove ()
print ("size", b:size())
b:insert (44)
print ("size", b:size())

for i=1, b:size () do
  print (b:get (i))
end
```

9.3 Meta-métodos

Algumas das funções relacionadas com o buffer circular poderiam ser fornecidas como meta-métodos da meta-tabela *CircBuffer*, de modo que poderíamos, por exemplo, usar o operador de tamanho # para obter o tamanho atual de um buffer.

Uma maneira simples de fazer isso, é alterando o array de registro *bufferlib_m*, como mostrado a seguir:

```
01 static const struct luaL_Reg bufferlib_m [] = {
02   {"insert", bufferinsert},
03   {"remove", bufferremove},
```

```
04   {"__len", buffersize},
05   {"maxsize", buffermaxsize},
06   {"get", bufferelement},
07   {NULL, NULL}
08 };
```

O trecho de código abaixo mostra o uso do operador # para obter o tamanho do buffer circular:

```
require 'buffer'

b = buffer.new (2)
b:insert (4)
print (#b)

b:insert (33)

for i=1, #b do
  print (b:get (i))
end
```

9.4 Criando Interfaces

No exemplo anterior, as funções que operam sobre o buffer circular também são responsáveis por realizar a comunicação entre C e Lua.

Embora esse seja possível projetar nosso programas dessa forma, muitas vezes queremos exportar para Lua uma estrutura de dados que já foi definida em C sem ter que alterar as funções que manipulam essa estrutura.

Nesse caso, o que devemos fazer é criar apenas funções de interface, que pegam os dados da pilha de Lua, chamam a função C correspondente e empilham o resultado na pilha.

Vamos redefinir então a função *buffernew*, retirando dela todo o código que está relacionado com Lua. Agora, *buffernew* deve retornar o novo buffer circular que foi criado. A nova definição da função é mostrada a seguir:

```
01 CircBuffer *buffernew (int n) {
02   size_t bsize;
03   CircBuffer *cb;
04
05   bsize = sizeof (CircBuffer) + (n - 1) * sizeof (int);
06   cb = (CircBuffer*)malloc(bsize);
07
08   cb->size = n;
09   cb->start = 0;
10   cb->end = 0;
11   cb->n = 0;
```

```
12
13   return cb;
14 }
```

Notem que também retiramos o código que verificava se o tamanho do buffer era válido ou não. Poderíamos ter deixado algum código que fazia essa verificação (sem usar funções da API C de Lua) em *buffernew*, mas preferimos fazer a verificação do tamanho do buffer na função de interface *buffernew_l*, que é mostrada a seguir:

```
01 int buffernewl (lua_State *L) {
02   CircBuffer *cb;
03   CircBuffer **ud;
04   int n;
05
06   n = luaL_checkinteger (L, 1);
07   luaL_argcheck (L, n >= 1, 1, "invalid buffer size");
08
09   cb = buffernew (n);
10   ud = lua_newuserdata (L, sizeof (CircBuffer *));
11   *ud = cb;
12
13   luaL_getmetatable (L, "CircBuffer");
14   lua_setmetatable (L, -2);
15
16   return 1;
17 }
```

As linhas $6 - 7$ de *buffernewl* verificam se o tamanho do buffer que o usuário deseja criar é válido.

Como a função *buffernew* é quem está realmente criando o buffer circular e realizando a alocação de memória, iremos criar um userdata cujo tamanho é apenas um ponteiro para uma estrutura *CircBuffer* (linha 10). O userdata irá apontar então para o buffer circular que foi retornado pela função *buffernew* (linha 11).

As demais funções de interface são semelhantes a *buffernewl* e irão obter os parâmetros da pilha, fazer alguma verificação quando necessário, chamar uma função C e empilhar os resultados na pilha. A seguir, temos a definição da função *bufferinserl*:

```
01 int bufferinsertl (lua_State *L) {
02   CircBuffer **ud = (CircBuffer **) luaL_checkudata (L, 1, "CircBuffer");
03   CircBuffer *cb = *ud;
04   int n = luaL_checkinteger (L, 2);
05
06   if (buffersize (cb) == buffermaxsize (cb))
07     luaL_error (L, "Error inserting %d: buffer is full\n", n);
```

```
08
09   bufferinsert (cb, n);
10
11   return 0;
12 }
```

Na linha 7, usamos a função *luaL_error*, que dispara uma mensagem de erro. O segundo parâmetro de *luaL_error* é uma mensagem de erro, onde podemos usar alguns formatadores, como *%s* e *%d*. Notem que *bufferinsertl* não manipula o buffer diretamente, apenas chama as funções que operam sobre o buffer, como *buffersize* e *bufferinsert*.

9.5 Coleta de Lixo

Quando as funções que usam a API C de Lua são meramente uma interface para funções C, que fazem realmente o trabalho pesado, podemos ter um problema com o gerenciamento de memória.

Como a alocação das estruturas de dados é feita por C, Lua não tem controle sobre a memória e você (programador C) é responsável por liberar a memória quando ela não está mais sendo usada.

Uma solução simples, seria exportar a função C que libera a memória e chamá-la de Lua, de modo que quando um buffer *b* não estivesse mais sendo utilizado, poderíamos ter uma chamada como *buffer:free()*.

Essa solução, embora seja válida, faz com que o programador Lua tenha que gerenciar explicitamente a memória dos objetos, o que não é uma boa idéia.

Uma solução mais adequada é definir na meta-tabela *CircBuffer* um meta-método *__gc*. Dessa forma, quando um userdata que representa um buffer circular não estiver mais sendo referenciado em Lua, o coletor de lixo irá chamar o seu meta-método *__gc*, que será responsável por liberar a memória e outros recursos alocados pelo objeto.