Capítulo 5

Uma Introdução à Programação em Lua

Roberto Ierusalimschy

Abstract

Lua is a scripting language widely used in several areas, from desktop applications, such as Adobe Photoshop Lightroom, to software for embedded systems. It is the leading language for scripting games and it is part of Ginga, the standard middleware for the Brazilian Digital TV system. Lua is also frequently used in security, being the scripting language embedded in tools like Wireshark, snort, and nmap.

This text presents the Lua language emphasizing its unconventional mechanisms. The goal is to introduce the language together with some non-conventional programming techniques, such as higher-order functions, coroutines, and APIs between languages. The text assumes some programming maturity from the reader and some knowledge about the C language, for the discussion about the Lua–C API.

Resumo

Lua é uma linguagem de script *amplamente usada nas mais diversas áreas, desde grandes aplicativos para desktops, como o Adobe Photoshop Lightroom, até software para sistemas embarcados. Lua é a linguagem mais usada atualmente para scripting em jogos, e é parte do padrão Ginga para o Sistema Brasileiro de TV Digital. Lua também é muito usada na área de segurança, sendo a linguagem de* script *embutida em ferramentas como Wireshark, snort e nmap.*

Este texto apresenta a linguagem Lua com ênfase nos seus mecanismos menos convencionais. O objetivo é introduzir a linguagem e ao mesmo tempo apresentar algumas técnicas de programação não convencionais, como o uso de funções de mais alta ordem, co-rotinas e APIs entre linguagens. Espera-se do leitor alguma maturidade na área de programação e conhecimento da linguagem C, para a discussão da API entre Lua e C.

5.1. Introdução

O objetivo deste texto é introduzir o leitor à programação na linguagem Lua. Assumimos que o leitor (você) possui uma certa maturidade em programação com alguma linguagem qualquer.

Programar em Lua não é muito diferente de programar em outras linguagens dinâmicas, mas é diferente. Cada linguagem apresenta características próprias, e um bom programador sabe explorar as características particulares de cada linguagem. Neste texto, vamos procurar enfatizar as particularidades de Lua, aspectos que tornam a programação em Lua diferente da programação em outras linguagens dinâmicas. Em particular, em Lua, temos como importantes diferenciais o uso de técnicas de programação funcional, o uso ubíquo de tabelas como estruturas de dados para os mais variados fins, o uso de co-rotinas e a comunicação com código escrito em C.

Bom, então programar em Lua não é tão diferente de programar em outras linguagens dinâmicas. Mas afinal, o que é uma linguagem dinâmica? Como ocorre frequentemente em computação, esse termo não possui uma definição precisa e universalmente aceita. Mas existe um certo consenso de que linguagens dinâmicas apresentam as seguintes características:

Interpretação dinâmica: isso significa que a linguagem é capaz de executar trechos de código criados dinamicamente, no mesmo ambiente de execução do programa. Como exemplos dessa facilidade temos a função `loadstring` em Lua e a função `eval` em Scheme/Lisp e Perl.

Tipagem dinâmica forte: tipagem dinâmica significa que a linguagem faz verificação de tipos em tempo de execução do programa. Linguagens com tipagem dinâmica em geral não possuem declarações de tipos no código e não fazem verificação de tipos em tempo de compilação. Tipagem forte significa que a linguagem jamais aplica uma operação a um tipo incorreto.

Gerência automática de memória dinâmica (coleta de lixo): isso significa que não precisamos gerenciar memória explicitamente no nosso programa; em especial, não há necessidade de um comando para liberar memória após seu uso.

Em geral, linguagens dinâmicas são interpretadas, e não compiladas para código nativo da máquina; mas essa é uma característica das implementações dessas linguagens, não das linguagens em si. Obviamente, as características acima favorecem uma implementação via um interpretador e dificultam a construção de compiladores.

Dessas características, a interpretação dinâmica é a mais exclusiva de linguagens dinâmicas. Obviamente, em qualquer linguagem Turing-completa podemos escrever um interpretador para a própria linguagem, mas os trechos de código interpretados não serão executados no mesmo ambiente do programa interpretador. Por exemplo, podemos escrever um interpretador para C em C, mas os programas interpretados não terão acesso às variáveis e funções declaradas no programa compilado onde o interpretador está sendo usado.

Apesar de não ser uma característica exclusiva de linguagens dinâmicas, a gerência automática de memória é um mecanismo importante dessa lista, por haver uma enorme diferença entre programarmos em uma linguagem com e em uma linguagem sem gerência automática de memória. Mesmo na programação em ponto grande (*programming in the large*) a gerência automática de memória tem um impacto significativo, ao simplificar as interfaces entre componentes. (Como um exercício, pegue a API de qualquer biblioteca C de porte razoável e verifique quanto de sua complexidade é devida à gerência de memória.)

Na verdade, existe um contínuo entre linguagens estáticas e dinâmicas. Por exemplo, Java é uma linguagem muito mais dinâmica do que C, pois apresenta gerência automática de memória, um certo grau de tipagem dinâmica e um mecanismo embrionário de interpretação dinâmica (por meio da carga dinâmica de classes, que por sua vez podem ser criadas dinamicamente). Mesmo entre as linguagens reconhecidamente dinâmicas existem diferenças. Por exemplo, nem todas as linguagens dinâmicas têm gerência automática de memória sobre módulos ou classes.

Lua se destaca de outras linguagens dinâmicas por ser uma linguagem de *script*. Uma linguagem de script é uma linguagem projetada para controlar e coordenar componentes geralmente escritos em outra linguagem. As primeiras linguagens de script foram as linguagens de *shell* do Unix, usadas para conectar e controlar a execução de programas. Apesar de várias linguagens dinâmicas poderem ser usadas para script, poucas foram projetadas para essa finalidade. Lua seguiu um caminho criado por Tcl [Ousterhout 1990], onde a linguagem é estruturada como uma biblioteca C com uma API que permite tanto código na linguagem chamar funções escritas em C como código C chamar funções escritas na linguagem. Lua se destaca de outras linguagens de script por sua simplicidade, portabilidade, economia de recursos e desempenho [Ierusalimschy et al. 2007].

5.2. Como usar Lua

A linguagem Lua conta com uma única implementação principal, mantida pelos autores da linguagem no site `www.lua.org`, mas essa implementação conta com diversas distribuições mantidas por desenvolvedores independentes.

Em muitos usos reais de Lua, o interpretador é distribuido embutido na aplicação final. Afinal, um dos principais objetivos de Lua é exatamente esse tipo de uso. Nesses casos, detalhes de como usar Lua são dependentes da aplicação. Esses detalhes incluem que editor usar, onde e como armazenar os programas, como executar um programa, etc. Neste texto, como não estamos visando nenhuma aplicação particular, vamos usar o interpretador independente (*stand alone*) de Lua.

Para máquinas Windows, uma ótima opção de instalação é a distribuição *Lua for Windows* (LfW).[1] Essa distribuição é um pacote completo para Windows, incluindo não apenas o interpretador Lua com suas bibliotecas padrão, mas também um editor e várias bibliotecas extras populares.

Para máquinas Linux não há uma receita pronta, dada a diversidade de distribuições de Linux. Compilar Lua em uma máquina Linux é muito simples e rápido. Algumas

[1] `http://luaforwindows.luaforge.net/`

distribuições já vêm com Lua instalado por default. Outras oferecem pacotes prontos: por exemplo, em Ubuntu e Debian, basta instalar o pacote `lua5.1`, que é o interpretador com as bibliotecas padrão. Várias bibliotecas externas também são oferecidas como pacotes extras. Em qualquer caso, o interpretador independente de Lua é um programa de linha de comando, para ser executado por meio de um terminal.

Para máquinas Mac OS X, existe a opção de compilar diretamente o fonte, desde que a máquina já tenha as ferramentas de desenvolvimento em C instaladas. O processo é simples e rápido como no Linux. Outra opção é usar um gerenciador de pacotes; por exemplo, tanto o MacPorts quanto o Fink oferecem pacotes prontos para Lua.

Uma vez instalado, é muito fácil usarmos o interpretador. Lua não tem o conceito de uma função `main`; qualquer comando passado ao interpretador é imediatamente executado. O exemplo a seguir é um programa completo para imprimir $\sqrt{2}$:

```
print(2^(1/2))        --> 1.4142135623731
```

O operador `^` é o operador de exponenciação em Lua. Lua trabalha sempre com números reais em ponto flutuante e a exponenciação funciona para expoentes fracionários (e negativos também). Em geral, usamos a notação `-->` para indicar o resultado de um comando. Como em Lua dois traços `--` iniciam um comentário que vai até o final da linha, podemos incluir aquela indicação no programa.

Ao usar Lua via um terminal, você tem pelo menos quatro maneiras de executar esse pequeno "programa":

- Você pode usar a opção de linha de comando `-e`:

  ```
  $ lua -e "print(2^0.5)"
  ```

  (Estou assumindo que `$` é o *prompt* do terminal.)

- Você pode entrar com o programa em modo interativo:

  ```
  $ lua
  > print(2^0.5)
  ```

  (O `>` é o *prompt* do interpretador Lua em modo interativo.)

- Você pode escrever esse programa em um arquivo e executá-lo via linha de comando:

  ```
  $ lua nome-do-arquivo
  ```

- Você pode escrever esse programa em um arquivo e executá-lo via modo interativo, por meio da função predefinida `dofile`:

  ```
  $ lua
  > dofile("nome-do-arquivo")
  ```

Podemos dispensar os parênteses em chamadas de função onde o único argumento é uma string literal. Assim, você pode reescrever o exemplo anterior como a seguir:

```
> dofile"nome-do-arquivo"
```

Quando você chama o interpretador, ele cria um estado Lua que persiste até o fim da sua execução. Assim, todos os efeitos colaterais de cada comando se propagam para os próximos comandos, mesmo que eles sejam executados como "programas" em separado. Veja o exemplo a seguir:

```
$ lua
> x = 1
> print(x)           --> 1
```

Cada uma das linhas é executada como um *trecho* (*chunk*, em inglês) separado, mas o valor da variável global `x` se mantém após o primeiro trecho ter terminado.

5.3. Alguns Exemplos

À primeira vista, Lua é uma linguagem imperativa, razoavelmente convencional. Como já discutimos, não vamos perder muito tempo descrevendo essa parte mais convencional da linguagem. Uma boa parte você vai aprender apenas vendo os exemplos; se precisar de maiores detalhes, consulte o manual de referência [Ierusalimschy et al. 2006] ou o livro *Programming in Lua* [Ierusalimschy 2006].

Seguem alguns exemplos de funções simples em Lua, para você se familiarizar com o básico da linguagem.

Soma dos elementos de um array

```
function add (a)
  local sum = 0
  for i = 1, #a do sum = sum + a[i] end
  return sum
end
```

Algumas observações sobre o código acima:

- Como a linguagem tem tipagem dinâmica, não há tipos nas declarações de variáveis, parâmetros, etc.
- A palavra reservada `local` declara uma variável local, cujo escopo vai da declaração até o fim do bloco mais interno que contém a declaração. No exemplo, `sum` é visível até o fim da função `add`.
- A expressão `#a` retorna o comprimento do array `a`. Como arrays em Lua começam no índice 1, o comprimento é também o valor do último índice.

- O comando `for` vai repetir seu corpo com o valor da variável `i` variando de 1 até o comprimento do array (`#a`). A variável de controle `i` é declarada pelo próprio comando `for` e só é visível dentro do seu corpo.
- Todas as estruturas de controle têm um terminador explícito. Tanto o corpo da função quanto o corpo do `for` terminam com a palavra reservada `end`.

Soma das linhas de um arquivo A função a seguir recebe o nome de um arquivo texto, que deve conter uma lista de números, e retorna a soma desses números:

```
function addfile (filename)
  local sum = 0
  for line in io.lines(filename) do
    sum = sum + tonumber(line)
  end
  return sum
end
```

Essa função é semelhante à do exemplo anterior, com exceção do `for`. No exemplo anterior, usamos um *for numérico*, que iterage sobre uma progressão aritmética de números. Neste exemplo, usamos um *for genérico*, que usa um *gerador* (`io.lines`, no caso, fornecida pela biblioteca padrão de Lua) para gerar os valores da iteração.

Note também o uso da função `tonumber`, para converter o numeral lido (uma string) para um número. Lua faz esse tipo de conversão automaticamente sempre que uma string é usada em uma operação aritmética, mas consideramos mais educado efetuar a conversão explicitamente.

Casamento de prefixos Um problema comum em várias áreas é, dada uma lista de palavras, decidir se uma dada string é prefixo de alguma das palavras da lista. Por exemplo, muitos sistemas de linha de comando permitem que entremos com um comando digitando apenas os primeiros caracteres do nome do comando.

Uma solução usual para esse problema é construir uma *tabela de prefixos*, que mapeia todos os prefixos de cada palavra na lista para a palavra completa. Dada essa tabela, o problema original é resolvido com uma simples consulta.

A função na Figura 5.1 recebe uma lista (array) de palavras e retorna sua tabela de prefixos. Nesse código temos novamente várias novidades:

- A expressão `{}` cria uma *tabela* vazia, que é atribuida à variável local `t`.
- O laço externo usa o gerador `ipairs`, que percorre todos os índices e valores do array dado (`list`). Os índices são atribuídos à primeira variável (que nomeamos _, já que não estamos interessados no seu valor), e os valores à segunda variável (`name`).
- A função `string.sub` retorna uma substring de uma dada string (`name`, no caso). Assim como arrays, caracteres em strings são indexados a partir de 1. Em

```
function buildPrefixTable (list)
  local t = {}
  for _, name in ipairs(list) do
    for len = 1, #name do
      local prefix = string.sub(name, 1, len)
      if t[prefix] then
        t[prefix] = true        -- colisao
      else
        t[prefix] = name
      end
    end
  end
  return t
end
```

Figura 5.1. Função para construir uma tabela de prefixos.

particular, a chamada como feita no exemplo vai retornar um prefixo de `name` com comprimento `len`.

- Na condição do `if`, usamos o fato de que o valor `nil`, que é o valor de campos não inicializados em uma tabela, é equivalente a falso em qualquer condição. Assim, o que está sendo testado é se o campo da tabela com chave `prefix` já foi preenchido anteriormente.

- No caso do teste dar positivo (isto é, a tabela já ter um elemento com a dada chave), a função coloca o valor `true` na posição já ocupada. Como este valor não é uma string, ele serve como uma marca para colisões.[2]

Após a construção da tabela de prefixos, seu uso é bem simples. Dado um prefixo, o código a seguir retorna a palavra completa ou dá um erro adequado:

```
function complete (t, prefix)
  local w = t[prefix]
  if type(w) == "string" then return w
  elseif w == true then error("ambiguous prefix")
  else error("invalid prefix")
  end
end
```

A função predefinida `type`, quando aplicada a qualquer valor, retorna uma string com seu tipo. Os valores de retorno possíveis são `"nil"`, `"number"`, `"string"`, `"boolean"`, `"table"`, `"function"`, `"thread"` e `"userdata"`. Já vimos, pelo

[2]Outra opção seria colocar na posição do conflito uma lista com todas as possíveis palavras para o dado prefixo.

menos brevemente, os tipos *number*, *string*, *table* e *nil* (que é o tipo do valor `nil`). Iremos abordar os tipos *function* e *thread* mais a frente. O tipo *userdata* é usado para representar objetos externos a Lua (e.g., arquivos). Isso deixa faltando apenas o tipo *boolean*.

Como em outras linguagens, o tipo *boolean* em Lua tem apenas dois valores, `true` e `false`. Mas os valores booleanos não têm exclusividade para testes. Em qualquer teste da linguagem (`if`, `while` e mesmo operadores lógicos) os valores `nil` e `false` resultam em um teste negativo, e qualquer outro valor (incluindo `true`, mas também 0, a string vazia, etc.) resulta em um teste positivo. Antes de sua versão 5.0, Lua nem tinha um tipo booleano. A principal motivação para a inclusão desse tipo na linguagem foi permitir a distinção entre variáveis com valor falso (`false`) e variáveis não inicializadas (e portanto com valor `nil`). Um teste como `if not x` dá positivo nos dois casos, mas um teste como `if x == false` só dá positivo se o valor de `x` for `false`.

Em Lua, assim como em várias outras linguagens dinâmicas, booleanos não têm exclusividade como resultado de operadores lógicos. O operador `or` retorna sempre o valor do primeiro operando que define o valor final da disjunção. A seguir listamos algumas expressões e seus respectivos resultados:

```
5 or 7        -->  5
nil or 7      -->  7
nil or false  -->  false
false or nil  -->  nil
```

De forma análoga, o operador `and` retorna o valor do primeiro operando que define o valor final da conjunção:

```
"a" and "b"    -->  "b"
nil and "alo"  -->  nil
nil and false  -->  nil
false and nil  -->  false
```

O operador `not`, entretanto, sempre retorna um booleano. Em particular, a expressão `not not x` normaliza o valor de `x` para um booleano correspondente.

Um exemplo de uso dessas propriedades é a expressão a seguir, que calcula o máximo entre `x` e `y`:

```
x >= y and x or y
```

Você está convidado a explicar como ela funciona.

5.4. Programando com Funções

Funções em Lua são valores dinâmicos de primeira classe. Em Lua, não declaramos funções, de forma estática. Funções são criadas dinamicamente, durante a execução de um programa.

Considere o exemplo a seguir:

```
> (function (a,b) print(a+b) end)(10, 20)
```

Esse comando cria uma função para imprimir a soma de seus dois parâmetros, e imediatamente chama essa função com argumentos 10 e 20. Após a execução do comando, não há mais referências para a função, e portanto os recursos usados por ela (memória) serão eventualmente reciclados pelo coletor de lixo.

De modo geral, a sintaxe `function (<pars>) <body> end` representa uma expressão que, ao ser executada, cria uma nova função. Ela é equivalente a notação `(lambda (<pars) <body>)` da linguagem Scheme e a uma abstração no λ-cálculo.

Como funções são valores de primeira classe, podemos fazer com elas tudo que fazemos com outros valores, como números ou strings. Em particular, podemos armazenar seu valor em uma variável global:

```
> foo = function (a,b) print(a+b) end
> foo(10, 20)
```

Agora, podemos usar a função armazenada em `foo` quantas vezes quisermos.

Como é bastante comum querermos armazenar funções em variáveis globais para uso futuro, Lua oferece um açúcar sintático para a atribuição acima. Podemos escrevê-la na forma a seguir:

```
> function foo (a,b) print(a+b) end
```

Essa sintaxe é exatamente a que estivemos usando até agora para declarar funções. Ela passa uma aparência de normalidade à linguagem, por ser bastante semelhante à declaração de uma função em outras linguagens. Lembre-se, no entanto, que ela nada mais é do que açúcar sintático. O significado dessa "declaração" é criar uma função dinamicamente e atribuí-la a uma variável global.

5.4.1. Múltiplos Retornos

Outro aspecto pouco convencional de funções em Lua é que elas podem retornar múltipos valores. Considere o exemplo a seguir:

```
function foo (x) return x, x+1 end
print(foo(3))    --> 3   4
```

Nesse exemplo, a função `foo` retorna dois valores, `x` e `x+1`. Na segunda linha, todos os retornos de `foo` são passados como argumentos para a função `print`.

Outra forma de se acessar os múltiplos retornos de uma função é com uma *atribuição múltipla*:

```
a, b = foo(10)
```

Nessa atribuição, `a` recebe o primeiro valor retornado por `foo` e `b` recebe o segundo.

Em uma atribuição múltipla, Lua não exige que o número de valores seja igual ao número de variáveis. Se houver mais valores, os valores extras são descartados; se houver mais variáveis, as variáveis extras recebem o valor `nil`. O mesmo ocorre em uma chamada de função, em relação aos argumentos fornecidos na chamada e os parâmetros esperados pela função: se chamamos uma função com mais argumentos que o necessário, os argumentos extras são descartados; se houver argumentos faltando, eles assumem o valor `nil`.

De maneira geral, em todos os lugares que Lua aceita uma lista de expressões (passagem de parâmetros, atribuição múltipla, construção de listas, retornos múltiplos) podemos usar uma função com múltiplos retornos como último (ou único) elemento da lista. Todos os valores retornados se juntam ao final dos valores das outras expressões. Entretanto, quando uma chamada de função não é a última expressão da lista, apenas um valor de retorno dela é usado, mesmo que ela retorne vários valores. Por exemplo, na atribuição

```
a, b, c = foo(10), foo(20)
```

`a` recebe o valor 10, `b` recebe o valor 20 e `c` recebe 21 (segundo retorno de `foo(20)`). O segundo retorno da primeira chamada a `foo`, 11, é descartado. Essa regra garante um mínimo de ordem na correspondência entre expressões e valores. Por exemplo, em uma chamada como `f(g(x),y)`, a regra garante que o argumento `y` será o segundo argumento, mesmo que `g(x)` retorne mais de um resultado.

O exemplo a seguir ilustra uma função que recebe uma string e retorna como resultados todos os prefixos da string:

```
function prefixes (s, len)
  len = len or 0
  if len <= #s then
    return string.sub(s, 1, len),
           prefixes(s, len + 1)
  end
end

print(prefixes("alo"))    -->     a    al    alo
```

Na verdade, a função recebe uma string e um número `len`, e retorna todos os prefixos da string com tamanho maior ou igual a `len`. Esse segundo parâmetro tem um valor *default* de zero; com isso, quando chamamos a função com apenas um parâmetro (a string), recebemos de volta todos os seus prefixos com tamanho maior ou igual a zero, ou seja, todos.

A primeira linha da função implementa o valor *default* para `len`. Quando chamamos a função pela primeira vez, passando apenas a string, o parâmetro `len` recebe o valor `nil`. Assim, o resultado do `or` na primeira atribuição é 0, que se torna o novo

valor do parâmetro. Nas chamadas recursivas, `len` recebe um valor numérico; portanto, o resultado do `or` é o próprio valor de `len`, e a atribuição não altera seu valor.

O resto da função é razoavelmente simples. Se o limite de tamanho for maior que o comprimento da string, não há nada a ser retornado. Caso contrário, a função retorna o prefixo com tamanho `len`, calculado com a função `string.sub` (que já vimos anteriormente), seguido dos prefixos com tamanho maior que `len`, calculados por meio da chamada recursiva.

5.4.2. Regiões Geométricas

Além de serem valores de primeira classe, funções em Lua oferecem escopo léxico. Isso permite empregarmos em Lua várias técnicas de programação funcional, como por exemplo o uso de funções para representação de dados. Para darmos uma pequena ilustração sobre essas técnicas, vamos desenvolver aqui um exemplo simples.

O exemplo a seguir é baseado em [Hudak and Jones 1994]. A ideia é desenvolver um sistema para representação de regiões geométricas. Uma *região* é um conjunto de pontos, e a única operação relevante no nosso sistema é pertinência: dado um ponto e uma região, saber se o ponto está dentro da (pertence à) região. Para desenharmos uma região, podemos percorrer a área de desenho testando a pertinência de cada pixel. O sistema deve ser capaz de representar uma ampla gama de regiões geométricas, e deve ser capaz de representar união, interseção e complemento de regiões genéricas.

Para desenvolver esse sistema, podemos começar pesquisando quais estruturas de dados se adequam melhor ao problema. Ou podemos subir o nível de abstração e ignorar as estruturas de dados, representando uma região diretamente por meio de sua função característica. Mais especificamente, uma região pode ser representada por uma função que, dado um ponto, retorna verdadeiro se e somente se o ponto pertence à região. Como exemplo, a função a seguir representa uma região circular com centro $(1.0, 3.0)$ e raio 4.5:

```
return function (x, y)
         return (x - 1.0)^2 + (y - 3.0)^2 <= 4.5^2
       end
```

Essa função é bastante simples: ela apenas verifica se a distância do ponto dado ao centro do círculo é menor ou igual ao raio.

Podemos ser mais abstratos e definir uma função para criar regiões circulares:

```
function circle (cx, cy, r)
  return function (x, y)
           return (x - cx)^2 + (y - cy)^2 <= r^2
         end
end
```

Essa função apenas retorna uma função que representa a região circular com centro e raio dados.

```
function union (g1, g2)
  return function (x, y)
           return g1(x, y) or g2(x, y)
         end
end

function inter (g1, g2)
  return function (x, y)
           return g1(x, y) and g2(x, y)
         end
end

function diff (g1, g2)
  return function (x, y)
           return g1(x, y) and not g2(x, y)
         end
end
```

Figura 5.2. União, interseção e diferença de regiões.

Para o funcionamento correto desse tipo de função, é fundamental que a linguagem ofereça *visibilidade léxica*. No nosso caso, a função que representa o círculo deve ser capaz de acessar as variáveis locais à função externa (`cx`, `cy` e `r`) mesmo após esta ter retornado.

Para adicionarmos novos formatos, como áreas retangulares ou triangulares, não há necessidade de nenhuma modificação no sistema; basta escrevermos as funções adequadas. Mas o mais interessante é como podemos modificar e combinar regiões. Por exemplo, dada qualquer região `g`, a função a seguir cria uma nova região que é o complemento de `g`:

```
function complement (g)
  return function (x, y) return not g(x, y) end
end
```

União, interseção e diferença de duas regiões são igualmente simples; vide Figura 5.2. Além de operações sobre conjuntos, também podemos definir vários outros tipos de transformações gráficas. Por exemplo, a função a seguir translada uma figura:

```
function translate (g, dx, dy)
  return function (x, y)
           return g(x + dx, y + dy)
         end
end
```

Observe como todas as funções que definimos necessitam visibilidade léxica, ao retornarem funções que necessitam acesso a valores externos para funcionarem corretamente.

```
local M, N = 500, 500      -- tamanho do desenho
function plot (f)
  io.write("P1\n", M, " ", N, "\n")
  for i = 1, N do          -- percorre as linhas
    local y = (N/2 - i)*2/N
    for j = 1, M do        -- percorre as colunas
      local x = (j - M/2)*2/M
      io.write(f(x, y) and "1" or "0")
    end
    io.write("\n")
  end
end
```

Figura 5.3. Função para desenhar uma região no formato PBM.

Como comentamos anteriormente, podemos visualizar as regiões geradas percorrendo a área de desenho e testando cada pixel. Para ilustrar o processo de uma forma simples, vamos escrever uma função para gerar um arquivo PBM (*portable bitmap*) com o desenho de uma dada região.

Arquivos PBM têm um formato bastante simples[3]: em sua variante de modo texto, ele começa com um cabeçalho de uma linha contendo a sequência `"P1"`; em seguida há uma linha com a largura e a altura do desenho, em pixels. Finalmente, há uma sequência de algarismos com os valores de cada pixel da imagem (1 para preto, 0 para branco), separados ou não por brancos ou quebras de linha. A função na Figura 5.3 gera o arquivo para uma região dada, mapeando os pixels da área sendo desenhada $[1,M] \times [1,N]$ para o espaço virtual de desenho $[-1,-1] \times [1,1]$: A iteração externa percorre as linhas do desenho, que correspondem à coordenada vertical; para cada linha a iteração interna percorre suas colunas, que correspondem à coordenada horizontal. Observe o uso idiomático dos operadores lógicos para transformar o resultado da função característica (um booleano) no caractere apropriado (0 ou 1).

A Figura 5.4 ilustra o resultado do comando a seguir:

```
plot(diff(circle(0, 0, 1), circle(0.3, 0, 1)))
```

5.4.3. Desenho de Polígonos

Um exemplo interessante do uso de funções em Lua vem da biblioteca NCLua, que é a interface entre Lua e o *middleware* Ginga, da TV Digital Brasileira. Uma das funções oferecidas por essa interface é a `drawPolygon`, para desenhar um polígono em uma área de desenho [ABNT 2007]. Um dos problemas de funções para desenhar polígonos é que a representação de um polígono geralmente requer uma estrutura de dados específica, que pode não ser compatível com a estrutura usada pelo programa. A função `drawPolygon` de NCLua evita essa dificuldade por meio de funções. Quando chamamos essa função

[3]Esse formato também é bastante ineficiente, mas aqui nossa ênfase é a simplicidade.

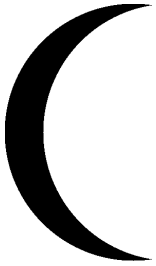

Figura 5.4. Desenho da diferença de duas regiões circulares

passamos apenas dois parâmetros: a tela (*canvas*) onde desenhar e o modo de desenho: *aberto*, *fechado* ou *preenchido*. Essa função então retorna uma nova função, que adiciona pontos ao polígono. Chamando repetidas vezes essa função retornada, podemos adicionar todos os pontos do polígono sem necessidade da criação de uma estrutura de dados particular. Quando chamamos essa função sem parâmetros, indicamos que não há mais pontos a inserir.

Para ilustrar o uso dessa função, o código a seguir desenha um polígono armazenado em um array de pontos, onde cada ponto é uma estrutura com dois campos, `x` e `y`:

```
local f = drawPolygon(canvas, "close")
for i = 1, #points do
  f(points[i].x, points[i].y)
end
f()
```

Na primeira linha, chamamos `drawPolygon`, que cria um novo polígono e retorna a função para adicionar pontos a esse polígono. Em seguida, fazemos uma iteração sobre o array `points` para adicionar os pontos ao polígono. Finalmente, chamamos a função de adição sem argumentos, para indicar que não há mais pontos a serem inseridos.

Uma facilidade adicional de `drawPolygon` é que cada chamada à função de adição de pontos retorna a própria função. Isso permite uma sintaxe simples para criarmos polígonos pequenos, via um encadeamento de chamadas. O código a seguir ilustra esse uso:

```
drawPolygon(c, "fill")(1.5,2.3)(3.6,4.5)(0.4,9.7)()
```

Não é difícil adaptarmos uma função para desenhar polígonos com uma interface convencional para essa interface funcional. Por exemplo, suponha que temos uma função como a descrita a seguir:

```
DP(canvas, mode, n, x, y)
```

Nessa função, `n` é o número de pontos do polígono, `x` é um array com as coordenadas horizontais dos pontos e `y` é um array com suas coordenadas verticais. Com essa função, podemos implementar a nova interface por meio do código mostrado na Figura 5.5. Quando

```
function drawPolygon (canvas, mode)
  local list_x, list_y, n = {}, {}, 0
  return function (x, y)
    if x == nil and y == nil then
      DP(canvas, mode, n, list_x, list_y)
    elseif type(x) ~= "number" then
      error("'x' deve ser um numero")
    elseif type(y) ~= "number" then
      error("'y' deve ser um numero")
    else
      n = n + 1
      list_x[n] = x
      list_y[n] = y
    end
  end
end
```

Figura 5.5. Uma implementação para `drawPolygon` sobre uma interface convencional.

chamada, nossa função `drawPolygon` cria dois arrays vazios (`list_x` e `list_y`) e um contador `n`, e retorna a função de adição de pontos, que executa o grosso do trabalho. Se chamada sem argumentos, ela invoca a primitiva `DP` para desenhar o polígono acumulado. Caso contrário, ela verifica se os argumentos possuem o tipo correto e insere um novo ponto na sua estrutura de dados. Essa estrutura, que é uma particularidade da função `DP`, fica completamente escondida do resto do programa.

5.4.4. Trechos de Código

Como discutimos anteriormente, a unidade de execução de código em Lua é chamada de *trecho*. Vamos ver agora mais detalhes sobre como Lua trata trechos de código.

Antes de executar um trecho de código, Lua pré-compila o trecho para um formato interno. Esse formato é uma sequência de instruções para uma máquina virtual, algo semelhante ao código de máquina para uma CPU convencional. Para essa pré-compilação, Lua trata o trecho exatamente como se ele fosse o corpo de uma função anônima. Além disso, o resultado da compilação é uma função Lua, com todos os direitos de qualquer função.

Normalmente, Lua executa a função anônima correspondente a um trecho imediatamente após sua compilação, de modo que essa fase de pré-compilação fica transparente para o programador. Mas podemos ter acesso a esse passo quando necessário. Em especial, a função `loadstring` compila um trecho de código arbitrário e retorna a função resultante, sem executá-la. Veja o exemplo a seguir:

```
i = 0
f = loadstring("print(i); i = i + 1")
f()     --> 0
```

```
f()     --> 1
```

O fato de trechos serem compilados como funções permite algumas técnicas úteis. Em particular, trechos podem ter variáveis locais. Como Lua oferece visibilidade léxica, essas variáveis são acessíveis para as funções declaradas dentro do trecho, mas são invisíveis fora dele. Em particular, uma função armazenada em uma variável local de um trecho só é visível dentro daquele trecho. Lua inclusive oferece um açúcar sintático para declararmos funções locais. Veja o exemplo a seguir:

```
local function foo (x)
  print(2*x)
end
```

Essa declaração é equivalente ao código a seguir:

```
local foo
foo = function (x)
  print(2*x)
end
```

Uma outra opção de tradução seria assim:

```
local foo = function (x)
              print(2*x)
            end
```

Mas essa tradução não é tão conveniente quanto a primeira opção. (Você sabe explicar por que? O que ocorre nos dois casos se quisermos definir uma função recursiva?)

Como qualquer função, trechos de código também podem retornar valores. Isso é útil em algumas situações particulares. Por exemplo, o interpretador independente de Lua imprime qualquer valor retornado por um trecho executado em modo interativo. Além disso, ele substitui um sinal de igual no início de uma linha por um `return`. Assim, podemos imprimir o resultado de uma expressão em modo interativo iniciando a linha com um sinal de igual:

```
$ lua
> = 2^-3       --> 0.125
```

Lembre-se que cada linha em modo interativo é tratado como um trecho completo.

Em Lua, a definição de funções é uma operação feita em tempo de execução, não em tempo de compilação. Afinal, ela é apenas açúcar sintático para uma atribuição. Assim, quando pré-compilamos um trecho de código contendo definições, essas definições não são válidas até executarmos a função resultante da compilação. Veja o exemplo a seguir:

```
f = loadstring("function foo (x) print(10*x) end")
print(foo)     --> nil
f()            -- executa trecho
print(foo)     --> function: 0x807ad58
foo(10)        --> 100
```

Mais especificamente, a função `loadstring` nunca gera nenhum efeito colateral. Seu único efeito é retornar uma nova função correspondente ao trecho de código passado como parâmetro. Qualquer efeito do trecho só é executado quando (e se) chamamos a função retornada.

5.5. Programando com Tabelas

Lua oferece um único mecanismo para estruturação de dados, chamado *tabela*. Tabelas nada mais são do que *arrays associativos*, isto é, uma estrutura de dados que associa chaves com valores e permite um rápido acesso ao valor associado a uma dada chave.

Tabelas são, provavelmente, a característica mais marcante de Lua. Muitas outras linguagens, em especial linguagens dinâmicas, oferecem arrays associativos, mas nenhuma os usa de modo tão extensivo quanto Lua. Em Lua usamos tabelas para implementar estruturas de dados como arrays, estruturas (registros), conjuntos e listas, e também para implementar conceitos mais abstratos como objetos, classes e módulos.

A semântica básica de tabelas é bastante simples. A expressão `{}` cria uma tabela vazia e retorna uma referência para ela. Uma atribuição `t[x]=y` associa o valor `y` à chave `x` na tabela referenciada por `t`. E a expressão `t[x]` retorna o valor associado à chave `x` na tabela referenciada por `t`, ou o valor `nil` caso a tabela não contenha a chave dada. Analogamente, se atribuimos `nil` a uma chave, eliminamos aquela chave da tabela: as chaves de uma tabela são aquelas com um valor associado diferente de `nil`.

Já vimos alguns usos de tabelas nos exemplos das seções anteriores. Um dos primeiros exemplos deste texto, de uma função para somar os elementos de um array, na verdade usa uma tabela. Lua não tem arrays. O que chamamos de array em Lua é meramente uma tabela cujas chaves são números naturais. Lua usa um algoritmo que garante que tabelas usadas como arrays são efetivamente armazenadas internamente como arrays. Essa implementação é completamente transparente para o programador; não existe nada na linguagem que dependa dessa implementação, com exceção do desempenho de certas operações.

Para manipularmos tabelas como estruturas, Lua oferece um açúcar sintático bastante simples: a notação `t.x` é equivalente a `t["x"]`, isto é, a tabela `t` indexada pela string literal `"x"`. O exemplo a seguir ilustra esse uso:

```
t = {}
t["x"] = 10;  t.y = 20;
print(t.x, t["y"])        --> 10    20
```

Note que, nessa sintaxe, só podemos usar como nomes de campos identificadores válidos na linguagem. Por exemplo, a expressão `t.or` é inválida, pois `or` é uma palavra reservada em Lua. Obviamente, a sintaxe básica `t["or"]` é sempre válida.

5.5.1. Construtores

A expressão para criar uma tabela vazia, `{}`, é na verdade um caso particular de um *construtor*. Construtores oferecem uma sintaxe bastante rica para a criação e inicialização de tabelas em Lua. Existem basicamente três tipos de construtores: um para listas, outro para estruturas e um genérico.

O construtor de listas tem a forma a seguir:

```
{exp1, exp2, exp3, ...}
```

Essa expressão cria uma tabela com o valor de `exp1` associado ao índice 1, `exp2` associado ao índice 2, e assim por diante. O trecho a seguir ilustra um uso simples desse tipo de construtor:

```
dia = {"domingo", "segunda", "terça", "quarta",
       "quinta", "sexta", "sábado"}
print(dia[5])      --> quinta
```

O construtor de estruturas tem a forma a seguir:

```
{nome1 = exp1, nome2 = exp2, nome3 = exp3, ...}
```

Essa expressão cria uma tabela com o valor da expressão `exp1` associado à string `"nome1"`, `exp2` associado à string `"nome2"`, e assim sucessivamente. O trecho a seguir ilustra um uso simples desse tipo de construtor:

```
point = {x = 10.5, y = -15.34}
print(point.x)       --> 10.5
```

O construtor genérico tem a forma a seguir:

```
{[e1] = exp1, [e2] = exp2, [e3] = exp3, ...}
```

Essa expressão cria uma tabela com o valor de `exp1` associado ao valor da expressão `e1`, `exp2` associado ao valor da expressão `e2`, e assim por diante. Em particular, qualquer construtor de listas pode ser reescrito na forma a seguir:

```
{ [1] = exp1, [2] = exp2, [3] = exp3, ...}
```

De forma similar, construtores de estruturas também podem ser reescritos usando-se o construtor genérico:

```
{ ["nome1"] = exp1, ["nome2"] = exp2, ...}
```

(Certifique-se que você entendeu por que essas equivalências estão corretas.)

Apesar de sua genericidade, o construtor genérico é menos usado que os dois anteriores, pois a maior parte dos usos típicos se encaixa naqueles padrões. Uma situação particular onde ele é útil é quando chaves do tipo string não são identificadores válidos, como no exemplo a seguir:

```
op = {["+"] = "add", ["-"] = "sub",
      ["*"] = "mul", ["/"] = "div"}
```

Um construtor também pode usar uma mistura desses três formatos. Por exemplo, considere o construtor a seguir:

```
{23, "ho", op = 10, ot = "a", ["or"] = 30}
```

Ele criará uma tabela com o valor `23` na chave `1`, `"ho"` na chave `2`, `10` na chave `"op"`, `"a"` na chave `"ot"` e `30` na chave `"or"`. (Como `or` é uma palavra reservada em Lua, ela não pode ser usada como um identificador.)

Quando escrevemos uma chamada de função onde o único argumento é um construtor, o uso dos parênteses é opcional. Isso dá um aspecto mais declarativo quando usamos Lua para descrição de dados, como neste fragmento:

```
character{"merlin",
  image = "files/img/merlin.jpg",
  strength = 100.5,
  category = wizard
}
```

Essa "descrição" é na verdade uma chamada à função `character` passando como argumento uma tabela com a string `"merlin"` no índice 1 mais os outros campos com chaves explícitas.

5.5.2. Arrays

Como já comentamos, em Lua representamos arrays diretamente como tabelas, usando números inteiros positivos como índices.

O uso de tabelas como arrays traz diversos benefícios. Por exemplo, Lua manipula arrays esparsos[4] de forma bastante eficiente, sem necessidade de algoritmos especiais. Se temos uma tabela `t` vazia, uma atribuição como `t[1000000000]=1` insere apenas um elemento na tabela, o par com chave 1000000000 e valor 1. (O mesmo programa em Perl dá erro de "Out of memory!") Além disso, todas as operações oferecidas pela linguagem para tabelas se estendem naturalmente para arrays.

Quando manipulamos arrays, frequentemente é necessário sabermos seu tamanho. Em algumas linguagens estáticas o tamanho de um array é parte do tipo do array (e.g.,

[4] Arrays onde a grande maioria dos elementos têm valor `nil` ou zero.

Pascal); em outras, é responsabilidade do programador saber o tamanho (e.g., C). Em linguagens mais dinâmicas, é comum existir um operador para se consultar o tamanho de um dado array. Como já vimos, Lua oferece o operador de comprimento (o operador prefixado #), que quando aplicado a um array retorna o seu comprimento. Mas o que é o "comprimento" de um array em Lua?

Como arrays são na verdade tabelas, o conceito de comprimento (ou tamanho) não é claro em todos os casos. Por exemplo, qual o tamanho de um array esparso? Seu número de elementos? Seu último índice? Para simplificar essas questões, Lua define o comportamento do operador de comprimento apenas para arrays *bem comportados*. Se os únicos índices numéricos presentes em uma tabela `t` são inteiros consecutivos de 1 até algum *n*, então esse *n* é o resultado da expressão `#t`; se a tabela não tem nenhum índice numérico, então o resultado da expressão `#t` é zero. Em todos os outros casos, o resultado de `#t` não é definido univocamente.

Em particular, arrays com *buracos*, isto é, com elementos com valor `nil`, não são bem comportados. Considere o construtor a seguir:

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

Para nós, pode parecer óbvio que queremos um array de quatro elementos, onde o terceiro tem valor `nil`. Mas para Lua, entretanto, não existe um terceiro elemento. Não é claro se o array termina neste `nil` e por acaso tem um outro elemento desrelacionado no índice 4. O operador de comprimento, se aplicado ao resultado desse construtor, pode retornar 2 ou 4. O ideal é evitar esses casos, usando o valor `false` em vez de `nil` na tabela. Se isso não for possível, deve-se usar algum artifício para indicar o tamanho do array, como por exemplo armazenar esse tamanho explicitamente:

```
{10, 20, nil, 40, n = 4}
```

Para arrays bem comportados, o operador # é bastante útil. A atribuição a seguir ilustra uma construção idiomática muito comum em Lua:

```
t[#t + 1] = v
```

Ela anexa o valor `v` ao final da lista `t`. De forma análoga, a atribuição a seguir apaga o último elemento de uma lista:

```
t[#t] = nil
```

5.5.3. Palavras mais Frequentes

Vamos ver agora um exemplo de um pequeno programa completo, que ilustra vários conceitos que vimos até agora. O objetivo do programa é listar as *n* palavras mais frequentes em um dado texto [Bentley et al. 1986].

O algoritmo geral desse programa é bastante simples. Primeiro percorremos todas as palavras do texto, contando quantas vezes cada uma aparece. Para isso, mantemos

```
local t = io.read("*all")

local count = {}
for w in string.gmatch(t, "%w+") do
  count[w] = (count[w] or 0) + 1
end

local words = {}
for w in pairs(count) do
  words[#words + 1] = w
end

table.sort(words, function (a,b)
  return count[a] > count[b]
end)

for i=1, (arg[1] or 10) do
  print(words[i], count[words[i]])
end
```

Figura 5.6. Programa para listar as palavras mais frequentes em um texto.

uma tabela que associa cada palavra já vista com o número de vezes que ela apareceu. Em seguida, ordenamos o resultado por ordem decrescente de frequências e listamos os *n* primeiros elementos da lista. A Figura 5.6 mostra o código completo do programa. Na primeira linha, lemos o arquivo de entrada todo de uma vez como uma única string (com a opção `*all` para a função de leitura `io.read`), e armazenamos o resultado em `t`. Isso é bastante razoável para arquivos com até algumas dezenas de megabytes. Mais adiante vamos ver como tratar arquivos realmente grandes.

Em seguida, criamos a tabela `count`, para armazenar a frequência de cada palavra, e percorremos todas as palavras do texto, contando-as. Para esse percorrimento, usamos um gerador baseado em um padrão: o laço será repetido para cada substring de `t` que case com o padrão `"%w+"`. Esse padrão significa “uma sequência de um ou mais caracteres alfanuméricos”, que é nossa definição de “palavra”. (Esse padrão é similar ao padrão `"\w+"` de Perl, por exemplo; em Lua usamos o caractere `%` como escape nos padrões para evitar conflito com o significado de `\` em strings.) Note o uso idiomático do conectivo `or` no corpo do laço: se a palavra `w` já tem um contador, o resultado da disjunção é esse contador; caso contrário, o resultado é zero. Em qualquer caso, o valor é incrementado e atribuído como a nova contagem associada àquela palavra.

O próximo passo é ordenar as palavras. Isso é um pouco mais sutil do que parece. Não podemos ordenar diretamente a tabela `count`, pelo simples fato de que tabelas não têm ordem; elas apenas mapeiam chaves para valores. Por isso, para impormos uma ordenação sobre as palavras, precisamos colocá-las em uma lista.[5] Assim, criamos uma

[5]Algumas pessoas se confundem com essa ideia: afinal, se listas também são tabelas, como ordená-las?

nova tabela (chamada `words`) e inserimos nela todas as palavras presentes como chave na tabela `count`.

Para ordenar a lista `words`, usamos a função predefinida `table.sort`. O segundo parâmetro dessa função é a função usada por `sort` para comparar os valores sendo ordenados. Essa função recebe como parâmetros dois valores sendo ordenados, e retorna verdadeiro se e somente se o valor do primeiro parâmetro deve preceder o valor do segundo parâmetro na ordem final. No nosso caso, os valores são as palavras sendo ordenadas, e a função então consulta a tabela `count` para comparar qual tem maior frequência.

Finalmente, no último laço imprimimos as *n* palavras mais frequentes, que são as primeiras da lista ordenada, e suas respectivas frequências. O valor de *n* pode ser dado como um argumento na chamada do programa; o interpretador independente armazena esses argumentos em uma tabela global `arg`. Quando não é fornecido um valor, o programa usa um valor preestabelecido (10, no exemplo).

Como comentamos anteriormente, a técnica de ler o arquivo inteiro para posterior tratamento é bastante eficiente para arquivos não muito grandes. Mas para arquivos grandes (da ordem de centenas de megabytes ou mais) ela pode se tornar inviável. Nesse caso, podemos modificar o início do programa para percorrer o arquivo linha a linha, e para cada linha percorrer suas palavras:

```
local count = {}
for line in io.lines() do
  for w in string.gmatch(line, "%w+") do
    count[w] = (count[w] or 0) + 1
  end
end
```

5.5.4. Módulos

A combinação de tabelas com funções de primeira classe é bastante poderosa. Lua não oferece nenhum mecanismo específico para a construção de módulos, pois módulos em Lua podem ser diretamente implementados como tabelas.

Quase todas as bibliotecas padrão de Lua são implementadas como módulos via tabelas. Nesse texto já usamos várias funções dessas bibliotecas. Por exemplo, a função `string.sub` é definida na biblioteca de manipulação de strings. Essa biblioteca exporta todas as suas funções dentro de uma tabela, armazenada na variável global `string`. Quando escrevemos `string.sub`, isso nada mais é que uma indexação convencional:

```
print(type(string))         --> table
print(type(string.sub))     --> function
```

Normalmente, um módulo é definido por meio de um trecho de código Lua armazenado em um arquivo.[6] Lua é indiferente em relação a como um módulo é escrito, desde

Realmente listas também não têm nenhuma ordem interna, mas suas chaves, inteiros positivos, têm uma ordenação natural independente da lista. Em outras palavras, do ponto de vista de Lua uma lista não tem ordem, mas definimos que o elemento associado à chave *n* é o *n*-ésimo elemento.

[6] Também podemos definir módulos para Lua em C.

que sua execução resulte na criação de uma tabela global contendo os itens exportados pelo módulo. Por razões que veremos mais adiante, também é educado o módulo retornar sua tabela de exportações. Como ilustração, o trecho de código a seguir define um módulo `vector` exportando duas funções, `norm1` e `norm2`.

```
vector = {}
function vector.norm1 (x, y)
  return (x^2 + y^2)^(1/2)
end
function vector.norm2 (x, y)
  return math.abs(x) + math.abs(y)
end
return vector
```

Observe que Lua oferece um açúcar sintático para definirmos funções diretamente como campos em tabelas.

Para carregar um módulo, podemos simplesmente executá-lo, por exemplo com a função predefinida `dofile`. Entretanto, Lua oferece uma função bem mais conveniente para a carga de módulos, chamada `require`. A função `require` difere de um simples `dofile` em dois aspectos importantes:

- Ela mantém uma lista de módulos já carregados, de modo que requerer um módulo já carregado não o carrega novamente.
- Ela usa uma lista de lugares onde procurar o arquivo contendo o módulo; com a função `dofile` temos que especificar o caminho completo até o arquivo.

A função `require` retorna como resultado o valor retornado pelo trecho que criou o módulo. Assim, se o trecho retorna a tabela de exportação, essa tabela será retornada quando chamarmos `require` para carregar o módulo. Isso permite o idioma a seguir:

```
local v = require("vector")
```

Esse idioma permite usarmos um outro nome para o módulo no nosso código (`v`, no exemplo), além de evitar o uso de globais.

5.5.5. Objetos

Programação orientada a objetos (OO) em Lua também se vale da combinação de tabelas com funções de primeira classe. Em Lua, um objeto é meramente uma tabela, contendo campos com seus dados (variáveis de instância) e operações (métodos).

O exemplo a seguir ilustra uma primeira abordagem com um objeto bem simples:

```
Rectangle = {x = 0, y = 0, width = 10, height = 20}
```

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

Uma dificuldade óbvia com esse esquema é que o método `area` só opera para esse retângulo particular. Para o método ser útil para qualquer retângulo, podemos incluir o objeto como parâmetro do método:

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

Agora, para chamarmos o método, temos que passar o objeto como argumento, o que não é muito conveniente:

```
print(Rectangle.area(Rectangle))
```

Para resolver esse inconveniente, Lua oferece o operador de dois pontos (*colon operator*). Com esse operador, podemos reescrever a chamada anterior da seguinte forma:

```
print(Rectangle:area())
```

O operador de dois pontos automaticamente insere o receptor do método como um primeiro argumento adicional na chamada.

De forma análoga, também podemos usar o operador de dois pontos na definição do método, como ilustrado a seguir:

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

Nesse caso, o operador insere automaticamente um primeiro argumento adicional na definição do método, com o nome *self*.

O uso do parâmetro *self*, mais o suporte sintático do operador de dois pontos, possibilita que um mesmo método possa operar sobre vários objetos semelhantes. Mas como criamos esses objetos? Em particular, partindo da definição de `Rectangle`, como podemos criar outros retângulos?

Uma opção seria copiarmos todos os campos do objeto original nos novos objetos; isto é, *clonarmos* o objeto original. Essa opção não é muito convidativa quando o objeto oferece muitos métodos. Uma outra opção, bem mais interessante, exige um novo mecanismo de Lua, chamado *delegação*, que foi inspirado na linguagem Self [Ungar et al. 1987].

Delegação permite que uma tabela "herde" campos de outra tabela. Mais especificamente, suponha que uma tabela `A` delega sua indexação para outra tabela `B`. Se indexamos `A` com uma chave presente, o valor associado é retornado normalmente. Mas se

indexamos A com uma chave ausente, Lua automaticamente irá procurar essa chave na tabela B.

Para construirmos uma relação de delegação entre A e B precisamos de uma tabela intermediária, chamada de *metatabela* de A.[7] Para a tabela A delegar suas buscas para B, o campo `__index` de sua metatabela deve referenciar B. O exemplo a seguir deve ajudar a clarificar essas relações:

```
A = {x = 10}
B = {x = 20, y = 30}
mt = {__index = B}        -- metatabela
print(A.x, A.y)           --> 10    nil
setmetatable(A, mt)
print(A.x, A.y)           --> 10    30
```

A chamada à função `setmetatable` estabelece `mt` como a metatabela de A. Podemos trocar a metatabela de uma tabela a qualquer hora (chamando a função `setmetatable`), assim como podemos trocar o conteúdo do campo `__index` (via atribuições convencionais). No momento do acesso a um campo ausente de uma tabela, o valor corrente do campo `__index` de sua metatabela corrente irá indicar a quem delegar o acesso. Se a metatabela ou seu campo `__index` não existir, o acesso retorna o valor `nil`, que é o comportamento normal de acesso a um campo ausente.

Nada nos obriga a criarmos uma terceira tabela para ser a metatabela. No exemplo anterior, tanto A quanto B poderiam ser a metatabela de A. Adotando a segunda alternativa, o código ficaria assim:

```
A = {x = 10}
B = {x = 20, y = 30}
B.__index = B
setmetatable(A, B)
print(A.x, A.y)           --> 10    30
```

Na Figura 5.7 juntamos tudo que temos até agora para definir novamente um objeto `Rectangle`. Desta vez, esse objeto vai oferecer um método `new` para criar novos retângulos, e usar delegação para que esses novos retângulos possam usar os métodos definidos no objeto original.

Vamos acompanhar, em detalhes, o que ocorre quando Lua executa as duas últimas linhas desse fragmento. Na primeira, o programa cria uma tabela e a passa como argumento para a função `Rectangle.new`. Devido à notação de dois pontos tanto na chamada quanto na sua definição, essa função recebe um outro parâmetro, `self`, com o valor de `Rectangle`. Essa função apenas atribui o valor de `Rectangle` como metatabela do novo objeto e o retorna.

Na próxima linha chamamos o método `area` nesse novo objeto. Lembre-se que `r:area()` é equivalente a `r.area(r)`. Ou seja, o primeiro passo é acessar o campo

[7] Delegação em Lua é uma instância específica de um mecanismo mais geral chamado *metamétodos*, que justifica o uso de metatabelas. Neste texto vamos discutir apenas delegação.

```
Rectangle = {x = 0, y = 0, width = 10, height = 20}
Rectangle.__index = Rectangle

function Rectangle:new (o)
  setmetatable(o, self)
  return o
end

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

r = Rectangle:new{width = 40, height = 60}
print(r:area())        --> 2400
```

Figura 5.7. Classe `Rectangle` usando delegação.

`area` na tabela `r`. Como essa tabela não tem esse campo, o interpretador consulta sua metatabela para uma alternativa. A metatabela de `r` é `Rectangle`, que também é o valor do seu campo `__index`. Assim, a função `Rectangle.area` se torna o resultado do acesso `r.area`. Lua então chama essa função, passando `r` como argumento extra para o campo `self`.

5.6. Programando com Co-rotinas

Co-rotinas são um tipo de mecanismo de controle bastante poderoso, mas pouco convencional. Como sempre, o termo em si não tem um significado preciso. Vários mecanismos de controle, muitos não equivalentes entre si, recebem o nome de "co-rotinas".

Genericamente, o termo co-rotina se refere a um mecanismo que permite que um procedimento suspenda temporariamente sua execução e continue mais tarde. Esse mecanismo básico permite diversas variações [de Moura and Ierusalimschy 2009]:

- Co-rotinas podem ser valores de primeira classe, isto é, co-rotinas podem ser manipuladas livremente e reinvocadas em qualquer ponto do programa, ou podem haver restrições sobre seu uso.
- O controle de fluxo pode ser simétrico ou assimétrico. O controle simétrico tem uma única primitiva para transferir o controle entre co-rotinas; o assimétrico tem duas primitivas, uma para suspender a execução e outra para reiniciar.
- Co-rotinas podem ser suspensas enquanto executando outras funções (*com pilha*, ou *stackful*) ou apenas enquanto executando sua função principal.

Em particular, Lua oferece um mecanismo de co-rotinas assimétrico, de primeira classe e *com pilha* [de Moura et al. 2004]. Nesta seção, vamos apresentar esse mecanismo e mostrar alguns de seus usos.

Co-rotinas, como implementadas por Lua, são bastante similares a linhas de execução (*threads*) cooperativas. Cada co-rotina em Lua representa uma linha de execução independente, com sua própria pilha de chamadas[8]. Mas, ao contrário de um sistema *multithreading* convencional, não há preempção em um sistema de co-rotinas. Uma co-rotina só interrompe sua execução quando termina ou quando invoca explicitamente uma primitiva de suspensão (*yield*).

A função `coroutine.wrap` cria uma co-rotina e retorna uma função que, ao ser chamada, executa (*resume*) a co-rotina.[9] O parâmetro único para `wrap` é uma função Lua contendo o código do corpo da co-rotina:

```
co = coroutine.wrap(function () print(20) end)
co()               --> 20
```

Essa função pode opcionalmente ter parâmetros, cujos valores são dados na chamada à co-rotina:

```
co = coroutine.wrap(function (x) print(x) end)
co("alo")          --> alo
```

Como já vimos, o poder de co-rotinas vem da possibilidade de uma co-rotina suspender sua execução para continuar posteriormente. Para isso, ela deve chamar a função `yield`, como no exemplo a seguir:

```
co = coroutine.wrap(function (x)
       print(x)
       coroutine.yield()
       print(2*x)
     end)

co(20)             --> 20
```

Note que, ao ser chamada, a co-rotina nesse exemplo executou apenas até a chamada a `yield`. Mas ela não terminou sua execução, apenas a suspendeu. Ao ser chamada novamente, ela irá continuar do ponto onde parou:

```
co()               --> 40
```

A função `yield` também pode passar um valor de retorno para o ponto onde a co-rotina foi invocada:

```
co = coroutine.wrap(function ()
       for i = 1, 10 do coroutine.yield(i) end
```

[8] Por isso é classificada como *stackful*.

[9] Também podemos criar uma co-rotina com a função `coroutine.create`, que não vamos tratar neste texto.

```
      return "fim"
    end)

print(co())        --> 1
print(co())        --> 2
  ...
print(co())        --> 10
print(co())        --> fim
```

O exemplo anterior, apesar de simples e de pouca utilidade, ilustra a essência da construção de *geradores*, que está intimamente ligada à construção de iteradores.

5.6.1. Iteradores e Geradores

Uma questão recorrente em programação é a construção de iteradores: estruturas de controle para percorrer os elementos de uma estrutura de dados em uma determinada ordem. Tradicionalmente, existem duas maneiras de se estruturar iteradores, chamadas de *exportação de dados* e *importação de ações* [Eckart 1987].

Exportação de dados se baseia em *geradores*: funções que, cada vez que são chamadas, retornam um próximo elemento da estrutura de dados (segundo alguma ordem). Essa é a forma mais usada em Java, por exemplo, por meio da interface `Iterator`.

Importação de ações usa funções de mais alta ordem: nesse esquema, uma *função de iteração* recebe uma função como parâmetro e a aplica a todos os elementos da estrutura de dados. Essa é a forma mais usada em Ruby, por exemplo (apesar de Ruby não usar funções de mais alta ordem, mas um mecanismo especialmente dedicado a esse fim). A vantagem dessa forma de iteradores é que é muito mais fácil manter o estado da iteração: considere, por exemplo, o percorrimento de uma árvore em pré-ordem. A desvantagem é que o laço de iteração não pode ser modificado, o que dificulta algumas tarefas: considere, por exemplo, o percorrimento de duas estruturas em paralelo.

Existe um problema famoso que ilustra simultaneamente os problemas das duas abordagens, chamado "problema das bordas iguais" (*same-fringe problem*). O problema consiste em, dadas duas árvores, determinar se o percorrimento das folhas das duas árvores produz a mesma sequência de elementos. Com exportação de dados, usando geradores, é difícil percorrer cada árvore, pois não temos recursão para manter o estado do percorrimento. Com importação de ações, usando funções de iteração, é difícil percorrer as duas árvores em paralelo.

O uso de co-rotinas elimina essa dicotomia entre as duas abordagens, pois torna trivial a transformação de uma função de iteração em um gerador. Para ilustrar essa transformação, vamos resolver o problema das bordas iguais.

Percorrer as folhas de uma árvore é uma tarefa bem simples, se usarmos recursão. O código na Figura 5.8 implementa um iterador via importação de ações. Esse iterador recebe a ação como uma função (`f`), e aplica essa função a cada folha da árvore `a` seguindo a ordem da esquerda para a direita.

Construir o iterador foi bem fácil; mas, como comentamos, um iterador não é

```
function leaves (a, f)
  if a ~= nil then
    if a.left or a.right then
      leaves(a.left, f)
      leaves(a.right, f)
    else      -- folha
      f(a.value)
    end
  end
end
```

Figura 5.8. Função para percorrer as folhas de uma árvore binária.

muito útil para o nosso problema. Não temos como percorrer duas árvores em paralelo usando esse iterador, já que o laço de percorrimento está embutido no iterador, e percorre apenas uma única árvore.

Portanto, vamos tranformar esse iterador em um gerador. Tudo que temos a fazer é especificar `yield` como a função de visita e executar a função de percorrimento dentro de uma co-rotina:

```
function gen (a)
  return coroutine.wrap(function ()
           leaves(a, coroutine.yield)
         end)
end
```

A função `gen`, quando chamada, retorna uma função que cede as folhas da árvore dada uma a uma, na ordem correta; ou seja, é um gerador. Com poucas linhas de código transformamos um iterador por importação de ações em um iterador por exportação de dados.

Agora, fica bem fácil resolver o problema das bordas iguais:

```
function samefringe (a, b)
  local gen_a, gen_b = gen(a), gen(b)
  repeat
    local a, b = gen_a(), gen_b()
    if a ~= b then return false end
  until a == nil
  return true
end
```

A função `samefringe` cria um gerador para cada árvore dada, e percorre as duas em paralelo comparando os elementos gerados.[10]

[10] Essa função se vale de uma sutileza das regras de escopo de Lua. Note que a variável local `a`, declarada dentro do laço, ainda é visível na sua condição.

```
local main = coroutine.wrap(function () end)
local next

current = main

transfer = function (co, val)
  if current ~= main then
    next = co
    return coroutine.yield(val)
  else
    current = co
    while current ~= main do
      next = main
      val = current(val)
      current = next
    end
    return val
  end
end
```

Figura 5.9. Implementação de `transfer` em Lua.

5.6.2. Co-rotinas Simétricas

A forma de co-rotinas usada por Lua, chamada de *assimétrica*, oferece duas primitivas para passagem de controle entre co-rotinas: `resume`, que é invocada implicitamente quando chamamos a co-rotina, e `yield`. Existe uma outra forma de implementarmos co-rotinas, chamada de *simétrica*. Essa segunda forma é mais conhecida por alguns programadores, devido a seu uso na linguagem Modula-2 [Wirth 1982].

Co-rotinas simétricas utilizam uma única primitiva de controle de fluxo, tradicionalmente chamada de `transfer`. Essa primitiva simultaneamente suspende a execução da co-rotina em execução e transfere o controle para uma outra co-rotina qualquer, dada como parâmetro na chamada a `transfer`.

A maioria das implementações de co-rotinas assimétricas não são com pilha (e.g., geradores em Python). Por essa razão, algumas pessoas tendem a achar que co-rotinas assimétricas são menos expressivas que co-rotinas simétricas. Mas isso não é verdade. Em particular, podemos implementar a função `transfer` em Lua, como veremos agora.

Como um `transfer` simultaneamente suspende uma co-rotina e ativa outra, podemos implementá-lo usando uma combinação de um `yield`, para suspender uma co-rotina, seguido de um `resume`, para ativar a outra. O código da Figura 5.9, adaptado de [de Moura and Ierusalimschy 2009], implementa essa ideia. Na primeira linha, o código cria uma nova co-rotina para representar o fluxo de execução principal, de modo a ser possível voltarmos o controle para esse fluxo. Em seguida, declara uma variável local `next`, que será usada pela `transfer`, e uma variável global `current`, que irá sempre

```
ping = coroutine.wrap(function ()
  while true do
    print('ping')
    transfer(pong)
  end
end)

pong = coroutine.wrap(function ()
  while true do
    print('pong')
    transfer(ping)
  end
end)

transfer(ping)
```

Figura 5.10. Um pequeno programa com co-rotinas simétricas.

conter a co-rotina que estiver em execução. Finalmente, temos a definição de `transfer`.

Quando a função `transfer` é chamada pela co-rotina principal, ela entra em um laço e invoca a co-rotina destino (`co`). A partir daí, qualquer co-rotina que chame `transfer` irá colocar o valor da co-rotina destino na variável `next` e ceder o controle, que voltará ao laço. Se a co-rotina destino for a principal, o laço termina e a co-rotina principal continua sua execução. Caso contrário, o laço repete e invoca a co-rotina destino (`current`).

A Figura 5.10 apresenta um pequeno programa que ilustra o uso da função transfer. Apesar de sua simplicidade, o programa apresenta uma típica arquitetura produtor–consumidor.

5.7. A API Lua–C

Como comentamos anteriormente, um dos pontos fortes de Lua é sua facilidade para se comunicar com C. Nessa seção, vamos ver como essa comunicação é feita.

Lua foi projetada para se comunicar com C. Isso é tão importante que Lua é organizada como uma biblioteca em C, não como um programa. O programa `lua`, que temos usado ao longo do texto para executar os exemplos, na verdade é um pequeno programa com menos de 400 linhas de código que é um cliente da biblioteca Lua. Essa biblioteca exporta pouco menos que 100 funções, que permitem executarmos trechos de código Lua, chamarmos funções, registrarmos funções C para serem chamadas por Lua, manipularmos tabelas, e outras operações básicas.

Sempre que manipulamos uma API como a de Lua, é importante lembrar que trabalhamos com dois níveis de abstração simultâneos. Por um lado, existe o nível do programa em C, que estamos escrevendo. Por outro lado, existe o nível do programa em Lua que está sendo manipulado por esse programa C. Por exemplo, existe uma função

na API para consultar o valor de uma variável global. Quando chamamos essa função, estamos executando uma chamada de função, no nível C, mas estamos consultando uma global no nível Lua.

Uma consideração importante na implementação de Lua é o tamanho e a portabilidade do código. Lua é comumente usada em plataformas bastante não convencionais, como consoles de jogos, conversores (*set-top box*) para TVs, cameras fotográficas, etc. Para diminuir seu tamanho e aumentar sua portabilidade, o código de Lua é dividido em três partes: o *núcleo*, uma *biblioteca auxiliar* e as *bibliotecas padrão*.

O núcleo contém toda a parte básica de Lua, como o pré-compilador, o interpretador, os algoritmos de manipulação de tabela e de coleta de lixo. Porém, ele não assume nada sobre o sistema operacional. Por exemplo, o núcleo faz toda sua alocação de memória chamando uma função externa, que deve ser fornecida a ele na sua inicialização. Da mesma forma, o núcleo não lê arquivos, mas carrega trechos de código chamando uma função externa apropriada. A API do núcleo é totalmente definida no arquivo `lua.h`; todos os nomes definidos por essa API tem o prefixo `lua_`.

A estrutura do núcleo é bastante apropriada para aplicações embarcadas, rodando em plataformas não convencionais que muitas vezes nem dispõem de um sistema operacional. Para plataformas mais convencionais, entretanto, ela é demasiadamente detalhista. Para isso existe a biblioteca auxiliar. Essa biblioteca usa a API do núcleo e funções normais do sistema operacional para oferecer uma interface de mais alto nível para o programador. Essa API auxiliar é definida no arquivo `lauxlib.h`; todos os nomes definidos por essa API tem o prefixo `luaL_`. Neste texto vamos usar a biblioteca auxiliar sempre que necessário, pois assumimos estar programando em uma plataforma convencional.

A comunicação Lua–C é bi-direcional. Por um lado, Lua pode chamar funções que na verdade estão escritas em C. Por exemplo, todas as bibliotecas padrão de Lua, como para manipulação de strings e para manipulação de arquivos, são escritas em C. Por outro lado, também é bastante fácil C chamar funções escritas em Lua. Isso permite que partes de um programa escrito em C sejam configuráveis por meio de código Lua.

Uma distinção útil é entre embutir (*embed*) e estender uma linguagem de script. *Embutir* é usar a linguagem como uma biblioteca C dentro de um programa hospedeiro, enquanto *estender* é escrever o programa principal na linguagem dinâmica *estendida* com funções escritas em C. Uma linguagem dinâmica poder chamar código escrito em C é bastante comum; mesmo em Java podemos fazer isso, por meio da *Java Native Interface*. Entretanto, poucas linguagens oferecem suporte adequado para serem embutidas, e portanto induzem o projetista a usar uma arquitetura de extensão, com o programa principal escrito na linguagem dinâmica [Muhammad and Ierusalimschy 2007]. Em Lua, podemos escolher qual é a melhor arquitetura para cada aplicação particular. Por exemplo, tanto *World of Warcraft* quanto Ginga usam Lua de forma embutida, enquanto o Photoshop Lightroom estende Lua.

5.7.1. Embutindo Lua

A comunicação Lua–C envolve sempre duas estruturas centrais, que vamos apresentar agora. A primeira é o *estado Lua*, representado por um ponteiro de tipo `lua_State *`.

```
#include "lua.h"
#include "lauxlib.h"

int main (int argc, char **argv) {
  lua_State *L = luaL_newstate();
  if (luaL_loadfile(L, argv[1]) != LUA_OK)
    fprintf(stderr, "error: %s\n",
                    lua_tostring(L, -1));
  else if (lua_pcall(L, 0, 0, 0) != LUA_OK)
    fprintf(stderr, "error: %s\n",
                    lua_tostring(L, -1));
  else {
    lua_getglobal(L, "result");
    printf("resultado: %f\n", lua_tonumber(L, -1));
  }
  lua_close(L);
  return 0;
}
```

Figura 5.11. Um programa para executar um arquivo Lua e imprimir o valor da variável global `result`.

Como Lua é implementada como uma biblioteca, seu código C não possui nenhuma variável global (*extern* ou *static*). Todo o estado do interpretador é armazenado na estrutura dinâmica `lua_State`. Para qualquer programa usar Lua, ele deve criar pelo menos um estado, por meio da função `luaL_newstate`. O valor retornado por essa função, um ponteiro de tipo `lua_State *`, deve ser passado como parâmetro para todas as outras funções da API. A função `lua_close` finaliza um estado e libera todos os recursos alocados por ele, incluindo memória, arquivos abertos e bibliotecas dinâmicas carregadas.

A segunda estrutura central é a *pilha*. A pilha é uma estrutura abstrata, que o código C só acessa por meio de chamadas à API. Como o nome implica, essa estrutura é uma pilha de valores Lua. Esses são os únicos valores que o código C consegue acessar. Para se acessar qualquer outro valor, ele deve primeiro ser copiado para a pilha.

Para tornar as coisas um pouco mais concretas, a Figura 5.11 mostra um programa completo usando a biblioteca Lua. Esse programa executa um arquivo Lua e imprime o valor numérico final da variável global `result`. Vamos analisá-lo passo a passo, ignorando alguns detalhes por enquanto.

A primeira coisa que o programa faz é criar um estado Lua. Em seguida, chama a função `luaL_loadfile` para compilar o arquivo `argv[1]` nesse estado. Se não houver erros na compilação, essa função deixa um valor do tipo `function` no topo da pilha e retorna o valor `LUA_OK`. Caso contrário, retorna um código de erro e deixa uma mensagem de erro no topo da pilha. A função `lua_tostring` converte esse valor, uma string em Lua, para uma string em C, que é então impressa na saída de erros (`stderr`).

A função criada pelo compilador representa o trecho de código lido do arquivo.

Até aqui esse código foi apenas compilado. Para executá-lo, precisamos chamar a função criada pelo compilador. Isso é feito pela função `lua_pcall`, que chama a função presente no topo da pilha. Enventuais parâmetros para a função devem também ser empilhados. Nesse caso, chamamos a função sem parâmetros, e isso é indicado pelo zero como segundo argumento de `lua_pcall`. Assim como `luaL_loadfile`, em caso de erros a função `lua_pcall` também deixa uma mensagem no topo da pilha e retorna um código diferente de `LUA_OK`. Em caso de sucesso, deixa na pilha o número de resultados pedidos. No nosso exemplo, queremos zero resultados, como indicado pelo terceiro argumento da chamada à `lua_pcall`.

No último trecho do código imprimimos o valor da variável `result`. A função `lua_getglobal` copia o valor da global nomeada para o topo da pilha e a função `lua_tonumber` converte esse valor para um número do tipo `double`.

Várias funções da API usam a pilha de modo normal, adicionando e retirando elementos pelo topo. Por exemplo, a função `luaL_loadfile` empilha a função criada ou a mensagem de erro; `lua_pcall` desempilha a função a ser chamada e eventuais argumentos e empilha eventuais resultados; e `lua_getglobal` empilha o valor da global nomeada. Outras funções podem acessar diretamente qualquer elemento dentro da pilha. Fazem parte dessa categoria todas as funções de *projeção*, que transformam valores Lua em valores C; no nosso exemplo vimos `lua_tostring` e `lua_tonumber`.

Para indicar um elemento na pilha, podemos usar um índice positivo ou negativo. Índices positivos contam a partir da base da pilha: o índice 1 indica o primeiro elemento que foi empilhado. Índices negativos contam a partir do topo: o índice -1 indica o elemento no topo, o último que foi empilhado. No nosso exemplo, usamos o índice -1 em todas as chamadas a funções de projeção, pois queríamos sempre referenciar o elemento no topo.

5.7.2. Estendendo Lua

O arquivo Lua executado pelo programa da Figura 5.11 pode ser algo tão simples quanto `result=12` ou algo bastante complexo, envolvendo muitas computações. Entretanto, ele não pode chamar nenhuma função: quando criamos um novo estado Lua ele está completamente vazio, sem nenhuma global definida. Nem mesmo as bibliotecas padrão estão abertas. Para abri-las, precisamos chamar a função `luaL_openlibs`, definida em `lualib.h`. Essa chamada vai povoar o estado Lua com todas as funções das bibliotecas padrão da linguagem.

A necessidade de chamar `luaL_openlibs` em separado é novamente explicada pela ênfase em flexibilidade da API de Lua. Muitos sistemas usam Lua sem oferecer todas as bibliotecas. Por exemplo, sistemas embarcados raramente podem implementar a biblioteca de entrada–saída, por não terem um sistema de arquivos. Outros sistemas podem não oferecer a biblioteca matemática para economizar espaço. Tais sistemas podem usar Lua sem nenhuma modificação, bastando não usar a função `luaL_openlibs`.

A função `luaL_openlibs`, assim como todas as funções da biblioteca padrão, não faz nada "mágico" para estender Lua. Ela apenas usa a API Lua–C. Vamos ver agora, por meio de um exemplo, como estender Lua com uma nova função.

```
static int math_sin (lua_State *L) {
  lua_pushnumber(L, sin(luaL_checknumber(L, 1)));
  return 1;
}
```

Figura 5.12. Função seno, da biblioteca matemática.

Uma função C, para poder ser chamada por Lua, deve respeitar um protocolo específico. No nível de C, a função deve seguir o protótipo abaixo:

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

Isso é, ela deve receber um único parâmetro, que é o estado Lua onde ela irá operar, e retornar um inteiro, que é o número de valores que ela está retornando no nível Lua. No nível de Lua, ela recebe seus parâmetros na pilha e retorna seus resultados também na pilha.

Cada função tem sua própria pilha local. Quando a função é chamada, a sua pilha contém apenas os parâmetros para a função, com o primeiro parâmetro na base da pilha. Ao retornar, os valores no topo da pilha são os resultados da função, com o último resultado no topo. O inteiro retornado no nível C indica quantos desses valores na pilha são resultados. Por exemplo, se a função retorna zero no nível C, significa que ela não está retornando nenhum valor no nível Lua, independente de quantos valores estão empilhados. Se ela retorna um no nível C, apenas o valor no topo da pilha é considerado valor de retorno. Esse esquema facilita a codificação das funções, pois não há necessidade de se limpar a pilha de outros valores ao retornar.

Como exemplo, na Figura 5.12 temos a implementação da função de seno, copiada diretamente da biblioteca matemática de Lua. A função `math_sin` é basicamente um encadeamento de três funções: `luaL_checknumber`, `sin`, da biblioteca matemática de C, e `lua_pushnumber`.

A função auxiliar `luaL_checknumber` verifica se o primeiro elemento da pilha é realmente um número e retorna seu valor; observe o uso do índice positivo 1 na sua chamada, para indexar o elemento na base da pilha. Essa função auxiliar é construída sobre a `lua_tonumber`, que já vimos anteriormente, mas usa antes a função `lua_isnumber` para se certificar que o parâmetro é realmente um número. Em caso negativo ela chama uma função de erro que, por meio de um *long jump*, salta diretamente para um tratador de erros, sem retornar. Assim, quando `luaL_checknumber` retorna sabemos que o parâmetro tem o tipo correto.

A função `lua_pushnumber`, como seu nome indica, empilha um número; no nosso caso, esse número é o resultado da chamada à função `sin`. Esse número será empilhado por cima do parâmetro original, que permanece na pilha. A função `math_sin` termina retornando 1, o que indica para Lua que apenas o valor do topo (o seno) deve ser considerado como resultado.

Após definirmos a função `math_sin`, precisamos registrá-la em Lua. A parte

principal do trabalho é feita pela função `lua_pushcfunction`: ela recebe um ponteiro para uma função C e empilha um valor Lua do tipo `function` que, quando chamado, invoca a função C correspondente. Como qualquer função em Lua, a função empilhada por `lua_pushcfunction` é um valor de primeira classe. Podemos por exemplo armazená-la em uma variável global, para uso futuro. O trecho de código a seguir ilustra como podemos modificar nosso programa da Figura 5.11 para registrar nossa função em uma global `sin`:

```
int main (int argc, char **argv) {
  lua_State *L = luaL_newstate();
  lua_pushcfunction(L, math_sin);
  lua_setglobal(L, "sin");
  /* ... como antes ... */
```

Após a criação do estado Lua, `lua_pushcfunction` cria a função no topo da pilha e `lua_setglobal` armazena o valor do topo da pilha na global nomeada (`sin`). Após essas chamadas, qualquer programa executado no estado `L` terá acesso à função `sin`.

5.8. Comentários Finais

Neste texto procurei ilustrar algumas das características mais marcantes da linguagem Lua. Abordamos funções de primeira classe, tabelas, co-rotinas e a API Lua–C. Procurei também enfatizar o poder expressivo desses mecanismos, mostrando alguns exemplos mais complexos.

Devido a restrições de tamanho, não pude abordar vários outros aspectos importantes de Lua. Em particular, o tratamento da API Lua–C foi bastante introdutório; por ser um mecanismo não muito convencional, temos que introduzir vários conceitos básicos novos para podermos chegar a tópicos mais avançados. Também não abordei as bibliotecas padrão de Lua; algumas são convencionais, como a biblioteca de manipulação de arquivos. Mas outras, como a biblioteca de strings com suas funções de casamento de padrões, merecem um tratamento mais longo. Espero que o leitor, após este texto introdutório, sinta-se motivado a aprofundar seus conhecimentos sobre Lua.

Referências bibliográficas

[ABNT 2007] ABNT (2007). *Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital*. Associação Brasileira de Normas Técnicas. ABNT NBR 15606-2.

[Bentley et al. 1986] Bentley, J., Knuth, D., and McIlroy, D. (1986). Programming pearls: a literate program. *Communications of the ACM*, 29(6):471–483.

[de Moura and Ierusalimschy 2009] de Moura, A. L. and Ierusalimschy, R. (2009). Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):6.1–6.31.

[de Moura et al. 2004] de Moura, A. L., Rodriguez, N., and Ierusalimschy, R. (2004). Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925.

[Eckart 1987] Eckart, J. D. (1987). Iteration and abstract data types. *SIGPLAN Notices*, 22(4):103–110.

[Hudak and Jones 1994] Hudak, P. and Jones, M. P. (1994). Haskell vs. Ada vs. C++ vs. Awk vs. ...— an experiment in software prototyping productivity. Technical report, Yale University.

[Ierusalimschy 2006] Ierusalimschy, R. (2006). *Programming in Lua*. Lua.org, Rio de Janeiro, Brasil, segunda edição.

[Ierusalimschy et al. 2006] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2006). *Lua 5.1 Reference Manual*. Lua.org, Rio de Janeiro, Brasil.

[Ierusalimschy et al. 2007] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2007). The evolution of Lua. Em *Third ACM SIGPLAN Conference on History of Programming Languages*, páginas 2.1–2.26, San Diego, CA.

[Muhammad and Ierusalimschy 2007] Muhammad, H. and Ierusalimschy, R. (2007). C APIs in extension and extensible languages. Em *XI Brazilian Symposium on Programming Languages*, páginas 137–150, Natal, RN.

[Ousterhout 1990] Ousterhout, J. (1990). Tcl: an embeddable command language. Em *Proc. of the Winter 1990 USENIX Conference*. USENIX Association.

[Ungar et al. 1987] Ungar, D. et al. (1987). Self: The power of simplicity. *Sigplan Notices*, 22(12):227–242. (OOPSLA'87).

[Wirth 1982] Wirth, N. (1982). *Programming in Modula-2*. Springer-Verlag.