UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

FELIPE FISCHER COMERLATO
LEONARDO EICH

Grupo Equipe 7
Projeto Space Invaders Utilizando Lua

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2018

SUMÁRIO

1 INTRODUÇÃO

Este trabalho tem como objetivo o estudo de uma linguagem de programação moderna com características híbridas contextualizando os conceitos vistos em aula ao longo do semestre e, por fim, analisar e avaliar diferentes linguagens de programação, seguindo os critérios vistos em aula.

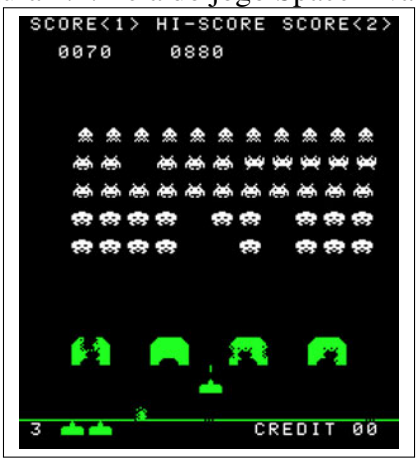
A tarefa principal do trabalho consiste em experimentar e comparar as características e funcionalidades orientadas a objeto e funcionais da linguagem de programação escolhida. De posse de uma linguagem, é necessário escolher um problema a ser solucionado com ela. O problema será, então, implementado duas vezes na mesma linguagem: uma delas usando somente Orientação a Objetos e a outra usando somente características funcionais.

O desenvolvimento do trabalho encontra-se em <https://github.com/felipefcomerlato/mlp_equipe7_2018-2>.

2 O PROBLEMA

O problema a ser resolvido neste trabalho é o desenvolvimento do jogo Space Invaders. O jogo expõe o jogador como uma espaçonave que deve destruir as espaçonaves inimigas que querem invadir o planeta do jogador. Na medida que elas avançam na tela (de cima para baixo), o jogador guia sua espaçonave horizontalmente e efetua disparos para destruir todas as ondas de inimigos que se seguem, como visto na Figura 2.1

Figura 2.1: Tela do jogo Space Invaders



Fonte: http://g1.globo.com/tecnologia/noticia/2011/10/homem-quebra-recorde-mundial-em-space-invaders-jogo-de-1978.html

3 VISÃO GERAL DA LINGUAGEM

A linguagem escolhida para o trabalho foi **Lua**. Lua é uma linguagem de script de multiparadigma, pequena, reflexiva e leve, projetada para expandir aplicações em geral, por ser uma linguagem extensível (que une partes de um programa feitas em mais de uma linguagem), para prototipagem e para ser embarcada em softwares complexos, como jogos. Assemelha-se com Python, Ruby e Icon, entre outras.

Lua foi criada por um time de desenvolvedores do Tecgraf da PUC-Rio, a princípio, para ser usada em um projeto da Petrobras. Devido à sua eficiência, clareza e facilidade de aprendizado, passou a ser usada em diversos ramos da programação, como no desenvolvimento de jogos (a Blizzard Entertainment, por exemplo, usou a linguagem no jogo World of Warcraft), controle de robôs, processamento de texto, etc. Também é frequentemente usada como uma linguagem de propósito geral.

Uma das características principais da linguagem é sua única estrutura de dados: tabelas - que podem ser usadas para representar arrays comuns, sequências, tabelas de símbolos, conjuntos, registros, grafos, árvores, etc.

Incluir Lua numa aplicação não aumenta quase nada o seu tamanho. O pacote de Lua 5.3.5, contendo o código fonte e a documentação, ocupa 297K comprimido e 1.2M descompactado. O fonte contém cerca de 24000 linhas de C. No Linux de 64 bits, o interpretador Lua contendo todas as bibliotecas padrões de Lua ocupa 247K e a biblioteca Lua ocupa 421K (ABOUT, 2018).

Lua possui interpretação dinâmica, ou seja, 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. Lua possui também tipagem dinâmica forte, ou seja, 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. Além disso, Lua conta com gerência automática de memória dinâmica ("coleta de lixo"), por tanto não precisamos gerenciar memória explicitamente no nosso programa; em especial, não há necessidade de um comando para liberar memória após seu uso (UMA. . . , 2018).

3.1 Modelo Funcional

Lua oferece um ótimo suporte ao modelo de programação funcional, nos permitindo utilizar a maioria dos conceitos desse paradigma de programação. Recursos como *pattern matching*, *currying*, funções anônimas, recursão e etc. são facilmente implementáveis, como será apresentado nas próximas seções. (MANUAL. . . , 2018)

3.2 Modelo Orientado a Objetos

Lua não é uma linguagem orientada a objetos por definição, mas nos permite adaptá-la para que seja utilizada dessa forma. A partir do momento em que podemos utilizar tabelas como classes, vincular métodos e atributos exclusivos a essas classes (encapsulamento) e implementar mecanismo de herança, envolvendo os diferentes tipos de polimorfismo, temos então a linguagem Lua no modelo orientado à objetos.

4 RECURSOS FUNCIONAIS

4.1 Elementos Imutáveis e Funções Puras

Funções puras são funções que não causam efeitos colaterais, ou seja, que não alteram valores de variáveis do programa. Em Lua não existe mecanismos explícitos para a criação de funções puras, ou seja, fica a cuidado do programador a implementação dessa técnica (IMMUTABLE. . . , 2018).

Um exemplo de função pura criada para o projeto é a função *newStateOfEnemies*, definida pelo código a seguir:

```
function newStateOfEnemies(enemies,death_row,death_col)
  copyEnemies = {}
  insertRows = function(r)
    if r > 0 then
      copyEnemies[r] = {}
      insertCols = function(c)
        if c > 0 then
          if r == death_row and c == death_col then
            copyEnemies[r][c] = 0
          else
            copyEnemies[r][c] = enemies[r][c]
          end
          insertCols(c-1)
        end
      end
      insertCols(#enemies[r])
      insertRows(r-1)
    end
  end
  insertRows(#enemies)
  return copyEnemies
end
```

A função *newStateOfEnemies* foi criada com a finalidade de atualizar o estado dos inimigos no jogo. A função recebe a matriz (uma tabela de tabelas, em Lua) de inimigos e a posição da matriz na qual um inimigo morreu, e devolve uma matriz totalmente nova representando o conjunto atualizado de inimigos.

4.2 Funções Anônimas

Em Lua, por definição, todas as funções são anônimas. Todavia, é possível armazenar as funções em variáveis, para que possam ser utilizadas da maneira tradicional (UMA..., 2018).

No projeto, um exemplo de uso dessa técnica é a implementação da função *setPlayerShot*, mostrada no código a seguir:

```
setPlayer = function()
    player_image = love.graphics.newImage("images/baseshipb.png")
    x_player = love.graphics.getWidth() / 2 - player_image:getWidth() / 2
    y_player = love.graphics.getHeight() - 2 * player_image:getHeight()
    max_x_player = love.graphics.getWidth() - player_image:getWidth()
    min_x_player = 0
    x_player_shift = 10

    setPlayerShot = function()
      player_shot_image = love.graphics.newImage("images/tiro.png")
      x_player_shot = 0
      y_player_shot = love.graphics.getWidth()
      player_shot_speed = 900
      y_player_shot_shift = 0
      player_shot_on_the_screen = false
    end
    setPlayerShot()
  end
```

Como pode ser visto, *setPlayerShot* foi implementada dentro da função *setPlayer*, portanto só existe dentro desse contexto. A função define a textura e posição inicial do tiro do jogador.

4.3 Currying

Currying é a técnica de transformar uma função de N parâmetros em N funções de 1 parâmetro cada (CURRYING..., 2018). Em nosso projeto, foi implementada uma função genérica *curry2*, mostrada abaixo:

```
function curry2(f)
  return function(a)
    return function(b)
      return f(a, b)
    end
  end
end
```

A função *curry2* é utilizada para fazer a verificação de colisão de cada inimigo com o tiro do jogador. A chamada da função de verificação de colisão com *curry2* é como no código a seguir:

```
verifyCollision = curry2(verifyCollision)
funRow = verifyCollision(row)
funCol = funRow(enemies_on_the_row)
```

4.4 Pattern Matching

Pattern matching ou casamento de padrões é a verificação por padrões dentro de uma estrutura de dados. Lua possui este recurso, no tratamento de *strings*, por exemplo (PATTERNS. . . , 2018). Em nosso projeto, utilizamos esse recurso de forma um pouco diferente. Lua nos permite definir variáveis dentro de tabelas, simulando assim uma espécie de dicionário. Assim, definimos a coordenada do *player* na tela por uma variável do tipo tabela, composta pelos identificadores *x* e *y*, como mostrado abaixo.

```
player_position = {
      x = love.graphics.getWidth() / 2 - player_image:getWidth() / 2,
      y = love.graphics.getHeight() - 100
    }
```

O acesso aos valores das coordenadas x e y do *player* é feita identificando as próprias variáveis *x* e *y* de dentro da variável de posição, como mostrado a seguir.

```
player_position.x
```

ou

```
player_position.y
```

4.5 Funções de Ordem Superior

Funções de ordem superior são aquelas que recebem uma função como parâmetro e/ou retornam uma função. Foi implementada genericamente a função *map*, que recebe uma função, uma tabela e o tamanho original da tabela, e aplica a função para cada um dos elementos da tabela, iterando sob um mecanismo simples de recursão. Esse tipo de função é comum no modelo de programação funcional, mas Lua não possui nativamente tal função, por isso foi implementada (HIGHER..., 2018). A função *map* implementada acabou não sendo utilizada no funcionamento do jogo, mas apenas para mostrar a permissividade do uso de funções de ordem superior em Lua. O código da função *map* pode ser visto em seguida.

```
function map(fun, t, sizeT)
  if sizeT > 0 then
    fun(t[sizeT])
    map(fun, t, sizeT-1)
  end
end
```

4.6 Funções de Ordem Maior Fornecidas Pela Linguagem

Conforme estudos realizados sobre Lua, foram encontrados poucos exemplos de funções de alta ordem disponíveis na linguagem. Um exemplo (não utilizado neste projeto) é a função *table.sort*, mostrada abaixo, que espera como um dos parâmetros uma função que define o sentido de ordenação de uma tabela (FUNCTIONS..., 2018).

```
list = {{3}, {5}, {2}, {-1}}
table.sort(list, function (a, b) return a[1] < b[1] end)
```

4.7 Funções como Elementos de Primeira Ordem

Funções são utilizadas como elementos de primeira ordem quando elas podem ser passadas como parâmetro e também retornadas por outras funções. Como mostrado na seção 4.3, a função *verifyCollision* é passada como parâmetro para a função *curry2* e retornada pela mesma (FUNCTIONS. . . , 2018).

4.8 Recursão

Conforme especificação, foi utilizada recursão como único mecanismo de iteração sobre as estruturas. Um exemplo de recursão implementada foi com a função *drawEnemies*, mostrada a seguir:

```
function drawEnemies(rows)

  if rows > 0 then
    drawEnemiesOnTheRow = function(row, enemies_on_the_row)
      x_enemy = enemies_on_the_row * x_distance_btw_enemies + x_enemies_shift
      y_enemy = row * y_distance_btw_enemies + y_enemies_shift

      if states_of_enemies[current_state][row][enemies_on_the_row] == 1 then
        fire = math.random(1, 20)
          if fire > 10 and fire < 20 then
            make_enemy_shot(x_enemy + enemy1_image:getWidth()/2,
                                      y_enemy+enemy1_image:getHeight())
        end
      end

      verifyCollisionPlayer(player_position.y,player_position.x)

      -- Implementacao currying
      local verifyCollision = currying2(verifyCollision)
      local funRow = verifyCollision(row)
      local funCol = funRow(enemies_on_the_row)

      if enemies_on_the_row > 0 then
        if states_of_enemies[current_state][row][enemies_on_the_row] == 1 then
            if row <= 2 then
              love.graphics.draw(enemy1_image, x_enemy, y_enemy)
            elseif row == 3 or row == 4 then
              love.graphics.draw(enemy2_image, x_enemy, y_enemy)
```

```
            else
              love.graphics.draw(enemy3_image, x_enemy, y_enemy)
            end
        end
        -- Desenha os demais inimigos da linha recursivamente
        drawEnemiesOnTheRow(row, enemies_on_the_row-1)
      end

    end
    drawEnemiesOnTheRow(rows, #states_of_enemies[current_state][rows])
    drawEnemies(rows-1)
  end
  -- Desenha o inimigo "mystery"
  x_mystery_enemy = x_mystery_enemy + x_mystery_enemy_shift
  love.graphics.draw(mystery_enemy_image, x_mystery_enemy, y_mystery_enemy)
end
```

A função *drawEnemies* desenha na tela cada inimigo através de uma tabela de tabelas, que é percorrida recursivamente para verificar se o inimigo de cada posição deve ou não ser desenhado.

5 RECURSOS DE ORIENTAÇÃO À OBJETOS

5.1 Classes

Em Lua não existe explicitamente o conceito de classe. No entanto, é possível simular classes através de tabelas - única estrutura de dados oferecida pela linguagem, como visto na seção 3. Podemos declarar uma tabela dentro de um arquivo .lua e retornar desse arquivo a tabela. Uma vez declarada uma tabela, é possível declarar, por exemplo, métodos vinculados a ela através do padrão NomeDaTabela.Método(). Um outro arquivo qualquer do programa pode obter essa tabela - classe - com todos seus métodos e atributos através de um require, simulando assim uma hierarquia de classes.

No código abaixo ... [blabl abl abla]

5.2 Encapsulamento e Proteção dos Atributos

Em Lua é possível tornar um atributo privado o declarando como *local*. Tal declaração restringe o atributo de forma que ele só fique acessível dentro do escopo da função na qual ele foi declarado.

Desta forma, declaramos os atributos de cada classe como *local* e implementamos *getters* e *setters* para obtê-los ou modificá-los de fora da classe, respeitando assim, o princípio de encapsulamento do paradigma de orientação à objetos. No código abaixo vemos um exemplo de encapsulamento utilizado na classe .....

5.3 Construtores

Como visto na seção 5.1, Lua não possui o conceito de classe explicitamente, e consequentemente não possui o conceito explícito de construtor. No entanto, podemos simular um construtor através de um método que inicialize os atributos e métodos de um objeto - tabela.

Em nosso trabalho utilizamos para cada classe, por convenção, um método chamado *new*, para simular o construtor-padrão de um objeto. Esse método retorna um objeto com os atributos inicializados e os métodos desse objeto.

Em Lua, é possível chamar uma função sem passar todos os parâmetros que ela

espera. Se isso acontecer, os parâmetros que não foram passados como argumento, são interpretados como valores *nil* dentro da função. Como Lua sobrescreve métodos com mesmo nome, independentemente dos parâmetros, para implementar um construtor alternativo, o próprio construtor padrão deve esperar todos os parâmetros possíveis para um objeto. Dessa forma, o construtor espera todos os parâmetros e só inicializa os diferentes de *nil*. O código abaixo mostra o construtor do objeto Enemy.

[CODIGO AQUI CARAI]

e depois fala do código.

5.4 Destrutores

Lua realiza gerenciamento automático da memória. Isto significa que você não precisa se preocupar com a alocação de memória para novos objetos nem com a liberação de memória quando os objetos não são mais necessários. Lua gerencia a memória automaticamente executando um coletor de lixo de tempos em tempos para coletar todos os objetos mortos (ou seja, objetos que não são mais acessíveis). Toda memória usada por Lua está sujeita ao gerenciamento automático de memória: tabelas, userdata, funções, fluxos de execução, cadeias de caracteres, etc. (REFERENCIA AQUI (MANUAL - secao 2.10))

No projeto, temos como exemplo de objetos a se tornarem inacessíveis depois de "mortos"os inimigos (classe *enemy*) do jogador. Quando o jogo é inicializado, são gerados todos os *enemies* e colocados dentro de uma tabela, acessível na *main*. Quando um inimigo morre, seu atributo *state* é alterado para 0. A cada *frame* do jogo, a função *deleteEnemies*, mostrada abaixo, percorre a tabela de inimigos e remove dela os inimigos com state = 0 [COLOCAR EM FORMATO DE CÓDIGO TALVEZ]. Quando um objeto *enemy* é removido da tabela, ele passa a não ser mais acessível de nenhum outro lugar do programa, portanto fica apto a ser recolhido pelo coletor de lixo de Lua.

[COLOCAR CÓDIGO AQUI CARAI]

5.5 Espaços de Nomes Diferenciados

Em Lua, se duas funções são declaradas com mesmo nome, a que foi avaliada por último sobrescreve a primeira. Em nosso projeto, todas as funções acessíveis a partir de

um mesmo objeto possuem nomes diferenciados. Foram implementados alguns métodos com mesmo nome mas pertencentes à classes diferentes, e assim são chamados somente explicitando a classe ou a instância. No código abaixo....

5.6 Mecanismos de Herança

Para implementar o mecanismo de herança, é necessário primeiramente importar a classe pai utilizando o comando *require*, salvando-a em uma variável. Uma classe torna-se filha de outra a partir do momento que em seu método construtor é invocado o método construtor da classe pai, salva na variável de requisição.

No projeto do Space Invaders, especificamos a classe abstrata *object*, que contém somente os atributos textura e posição. Em um segundo nível de hierarquia as classes *character* e *obstacle* herdam os atributos de *object* e implementam novos atributos e métodos. A classe *character* abstrai o atributo *speed*, herdado por um terceiro nível de hierarquia: as classes *enemy* e *player*.

[CÓDIGO BL BLABLABLBAL]

5.7 Polimorfismo por Inclusão

O polimorfismo por inclusão permite que um método de uma classe pai seja chamado por instâncias de diferentes classes, uma vez que elas herdam o método através do mecanismo de herança, explicado na seção 5.6.

Em nosso projeto, um exemplo de polimorfismo por inclusão está presente na classe *character*. Nela é implementado o método *character:collisionTest(shooter)*, sendo o parâmetro *shooter* o objeto que disparou um tiro, que verifica se houve colisão do tiro com a instância que chamou o método, tratada como *self*.

[CÓDIGO AQUI]

5.8 Polimorfismo Paramétrico

Como Lua implementa tipagem dinâmica, ou seja, o tipo de uma variável é definido no instante em que ela recebe um valor, as funções podem receber parâmetros de qualquer tipo e tratá-los da maneira adequada. Esse tratamento de variáveis é desvan-

tajoso pois transfere ao programador a responsabilidade de prever o comportamento do programa conforme o tipo de parâmetro passado para uma função, por exemplo.

Um exemplo simples de polimorfismo paramétrico, que não foi implementado no projeto mas que mostra permissividade em Lua, é mostrado no código abaixo.

5.9 Polimorfismo por Sobrecarga

A sobrecarga (overload) consiste em permitir, dentro da mesma classe, mais de um método com o mesmo nome. Entretanto, eles necessariamente devem possuir argumentos diferentes para funcionar(REFERENCIA AQUI). Porém, como mencionado na seção 5.5, Lua sobrescreve métodos com mesmo nome implementados em uma mesma classe. A implementação do construtor da classe *enemy*, mostrado na seção 5.3, exemplifica uma maneira de utilizar método com comportamento alternativo ao invés de implementar dois métodos com mesmo nome.

5.10 Delegates

O recurso de *delegates* não foi implementado no projeto.

6 PARALELISMO

Lua não oferece recurso nativo de paralelismo. Uma alternativa semelhante ao uso de *threads*, mas que não funciona de fato em paralelo, são as co-rotinas. A principal diferença entre *threads* e co-rotinas é que conceitualmente um programa com *threads* roda várias *threads* simultaneamente. Co-rotinas, por outro lado, são colaborativas: um programa com co-rotinas está a cada tempo rodando apenas uma de suas co-rotinas. Uma co-rotina pode ser suspensa para dar vez a outra, compartilhando recursos com ela.

Existem bibliotecas que possibilitam o uso paralelo de threads, porém o compartilhamento de recursos é extremamente limitado. Para o projeto, por exemplo, a biblioteca gráfica love2d não permite o compartilhamento de recursos gráficos entre threads, que seria a principal aplicação do conceito de paralelismo no jogo Space Invaders.

Para exemplificar o uso de paralelismo, embora não utilizado efetivamente em nosso projeto, foi implementado um pequeno programa, mostrado abaixo.

Arquivo main.lua:

```
socket = require "socket"

function love.load()
  thread = love.thread.newThread("thread.lua")
  thread2 = love.thread.newThread("thread.lua")

  input_5 = 5
  input_4 = 4

  execution_start = socket.gettime() * 1000

  thread:start(input_4)
  thread2:start(input_5)
  thread:wait()
  thread2:wait()

  execution_time = socket.gettime() * 1000 - execution_start
  print(string.format("Tempo total de execucao: %.0f Milisegundos",
                      execution_time))
end
```

Arquivo thread.lua:

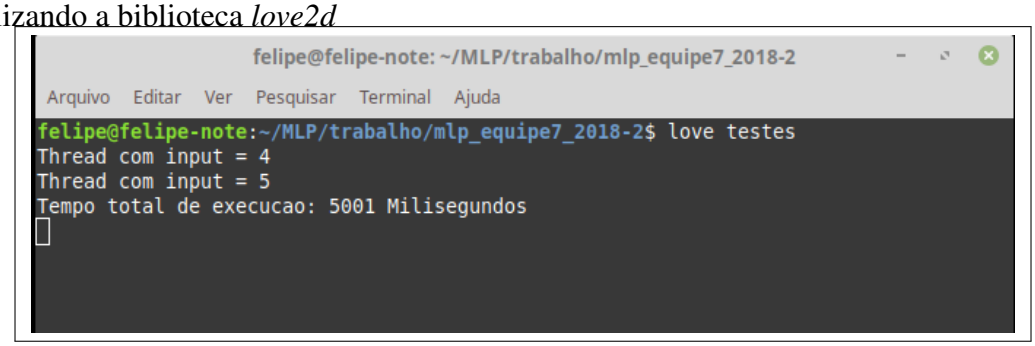
```
require "love.timer"

local input = ...

function testSleep(input)
  print("Thread com input = " .. input)
  love.timer.sleep(input)
end

testSleep(input)
```

Como utilizamos a biblioteca *love2d*, a função *load* é invocada automaticamente, por padrão. Primeiramente, são definidos os objetos do tipo *thread*. Uma *thread* é inicializada com o valor 4 e a outra com o valor 5. Os métodos *wait*, chamados por cada objeto *thread*, indicam que o programa só deve prosseguir quando a *thread* correspondente terminar. Por fim, é mostrado na saída o tempo de execução final do programa.

A *thread* em questão, utilizada para esse exemplo, executa apenas um *sleep* do tempo recebido por parâmetro, em segundos. Ou seja, o programa executa uma *thread* que dura 4 segundos e outra que dura 5 segundos.

Sem o uso de *threads*, o programa demoraria aproximadamente 9 segundos de execução entre um ponto imediatamente antes da inicialização da primeira *thread* e um ponto imediatamente depois do término da segunda *thread*. Já com o uso das *threads*, essa execução levou apenas 5 segundos, como mostrado na Figura 6.1.

Figura 6.1: *Print Screen* da saída do programa de exemplo do uso de paralelismo em Lua utilizando a biblioteca *love2d*



7 CONCLUSÃO FINAL

REFERÊNCIAS

ABOUT. 2018. Available from Internet: <https://www.lua.org/about.html>.

CURRYING in Lua. 2018. Available from Internet: <https://www.reddit.com/r/lua/comments/6yy9wu/currying_in_lua/>.

FUNCTIONS Tutorial. 2018. Available from Internet: <http://lua-users.org/wiki/FunctionsTutorial>.

HIGHER Order Functions in Lua. 2018. Available from Internet: <http://inmatarian.github.io/2015/08/09/send_more_money/>.

IMMUTABLE Objects. 2018. Available from Internet: <http://lua-users.org/wiki/ImmutableObjects>.

MANUAL de Referência de Lua 5.2. 2018. Available from Internet: <https://www.lua.org/manual/5.2/pt/manual.html#2.6%20%E2%80%93%20Co-rotinas>.

PATTERNS Tutorial. 2018. Available from Internet: <http://lua-users.org/wiki/PatternsTutorial>.

UMA Introdução à Programação em Lua. 2018. Available from Internet: <https://www.lua.org/doc/jai2009.pdf>.