CENTRO DE EDUCAÇÃO SUPERIOR DE BRASÍLIA

CENTRO UNIVERSITÁRIO INSTITUTO DE EDUCAÇÃO SUPERIOR DE BRASÍLIA

Mário Sérgio Oliveira de Queiroz
Pedro Martins

# Paradigmas de Programação da linguagem LUA

Brasil

25 de Novembro de 2013

Mário Sérgio Oliveira de Queiroz
Pedro Martins

# Paradigmas de Programação da linguagem LUA

Projeto para a disciplina Projeto Integrador VI - Paradigmas de Linguagem de Programação, do Centro Universitário Instituto de Educação Superior de Brasília, DF.

IESB - Centro Universitário Instituto de Ensino Superior de Brasília

Ciência da Computação

Orientador: João Paulo Ataíde Martins

Brasil

25 de Novembro de 2013

Mário Sérgio Oliveira de Queiroz
Pedro Martins
Paradigmas de Programação da linguagem LUA. – Brasil, 25 de Novembro de 2013-
39 p.

Orientador: João Paulo Ataíde Martins

TCC (Graduação) – IESB - Centro Universitário Instituto de Ensino Superior de Brasília
Ciência da Computação, 25 de Novembro de 2013.

1. Lua. 2. Paradigmas. I. João Paulo. II. IESB. III. Ciência da Computação.
CDU 02:141:005.7

Mário Sérgio Oliveira de Queiroz
Pedro Martins

# Paradigmas de Programação da linguagem LUA

Projeto para a disciplina Projeto Integrador VI - Paradigmas de Linguagem de Programação, do Centro Universitário Instituto de Educação Superior de Brasília, DF.

Trabalho aprovado. Brasil, 25 de Novembro de 2013:

---

**João Paulo Ataíde Martins**
Orientador

---

**Professor**
Convidado 1

---

**Professor**
Convidado 2

Brasil
25 de Novembro de 2013

*Este trabalho é dedicado às crianças adultas que,*
*quando pequenas, sonharam em se tornar cientistas.*

# Agradecimentos

Os agradecimentos principais são direcionados à Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes, sendo eles os criadores da linguagem Lua, e todos aqueles que contribuíram para a produção deste trabalhos acadêmico conforme as normas estabelecidas.

Agradecimentos especiais são direcionados ao Centro Universitário de Ensino Superior de Brasília por fornecer toda a infraestrutura e base acadêmica para a elaboração de projetos como este.

*"Tudo tem o seu tempo determinado,*
*e há tempo para todo o propósito debaixo do céu.*
*Há tempo de nascer, e tempo de morrer;*
*tempo de plantar, e tempo de arrancar o que se plantou*
*(Bíblia Sagrada, Eclesiastes 3, 1 e 2)*

# Resumo

Este projeto se refere à estudos e pesquisas, relativas aos paradigmas e conceitos da linguagem de programação Lua.

Para isso, foi utilizado a documentação oficial da linguagem como base contida em (IERUSALIMSCHY, 2006), e os conceitos relacionados a projetos de linguagem de programação do livro (SEBESTA, 2003). Como ferramentas para a elaboração do trabalho foi utilizado o latex para a parte de documentação e o sublime para a execução de códigos Lua.

Com base no que foi mencionado, o projeto deseja abordar todos os conceitos e paradigmas da linguagem de programação Lua.

**Palavras-chaves**: latex. abntex. editoração de texto.

# Abstract

This is the english abstract.

**Key-words**: latex. abntex. text editoration.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| TV | Televisão |
| PUC-Rio | Pontifícia Universidade Católica do Rio de Janeiro |
| Tecgraf | Instituto de Desenvolvimento de Software Técnico-Científico PUC-Rio |
| LabLua | Laboratório de desenvolvimento da linguagem Lua |
| PETROBRAS | Petróleo Brasileiro S.A. |
| DEL | Linguagem para Especificação de Diálogos |
| SOL | Simple Object Language |
| BSD | Berkeley Software Distribution |
| MIT | Massachusetts Institute of Technology |
| BNF | Formalismo de Backus-Naur |

# Sumário

# 1 Introdução

Este trabalho acadêmico se refere ao desenvolvimento de um estudo e pesquisa, relativo aos paradigmas e conceitos da linguagem de programação Lua. Desta forma, serão atingidos temas como implementação de sintaxe e semântica da linguagem.

## 1.1 Motivação

Conforme a proposta de projeto para o semestre, no que se refere ao estudo dos paradigmas de uma linguagem de programação, a escolha do grupo pela linguagem Lua, teve vários estímulos, como o fato da linguagem ter surgido em uma universidade brasileira, além de possuir uma ampla aplicação no ambiente de jogos e na indústria de TV digital.

Em virtude do que foi mencionado, existiram muitas influências para a escolha de Lua para esse projeto, existia o interesse em outras linguagens como Python, más devido as outras escolhas, optamos por Lua, que inclusive temos algumas experiências de trabalho.

## 1.2 Objetivos

### 1.2.1 Geral

Este trabalho tem como objetivo aplicar os conhecimentos obtidos na disciplina de paradigmas de linguagem de programação à linguagem Lua. Aprofundar e colocar em prática os conceitos aprendidos em sala de aula, documentando e exemplificando o funcionamento da linguagem Lua.

### 1.2.2 Específicos

Com base no objetivo geral derivam-se os seguintes objetivos específicos.

- Embasar historicamente a linguagem.
- Explicar o funcionamento da sintaxe e semântica.
- Demonstrar os paradigmas envolvidos na linguagem.
- Explicar e exemplificar o funcionamento de variáveis. Incluindo os tipos, sua vinculação, verificação de tipo e escopo.

- Apresentar as vantagens, desvantagens e as áreas a qual Lua melhor se aplica.
- Criar códigos para exemplificar os conceitos apresentados.

## 1.3 Organização do Trabalho

# 2 Histórico

A linguagem Lua foi totalmente projetada, e implementada no Brasil, por Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes, que eram membros do Computer Graphics Technology Group na PUC-Rio, a Pontifícia Universidade Católica do Rio de Janeiro. Lua nasceu e cresceu no Tecgraf, Grupo de Tecnologia em Computação Gráfica da PUC-Rio. Atualmente, Lua é desenvolvida no laboratório LabLua. Tanto o Tecgraf quanto Lablua são laboratórios do Departamento de Informática da PUC-Rio.

O estímulo inicial para a construção da linguagem veio de um projeto entre a PETROBRAS e a PUC-RIO, a fim de produzir um programa de interfaces gráficas para várias aplicações.

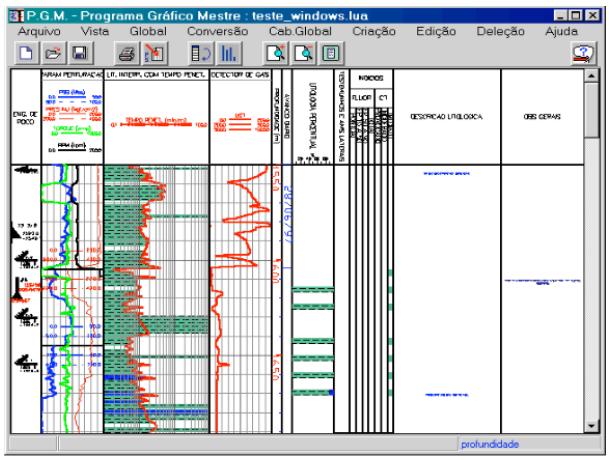


Figura 1 – Programa Gráfico Mestre

Logo surgiu o primeiro protótipo, DEL - Linguagem para Especificação de Diálogos, que trabalhava com lista de parâmetros e tipos e valores padrões. Com o passar do tempo após pesquisas e mudanças no projeto surgiu a linguagem 'SOL' - Simple Object Language, sendo que era uma linguagem para descrição de objetos, inspirada no bibTex.

```
type @track {x:number, y:number=23, z}

type @line {t:@track=@track{x=8}, z:number*}

-- create an object 't1', of type 'track'
t1 = @track{y=9, x=10, z="hi!"}

l = @line{t=@track{x=t1.y, y=t1.x}, z=[2,3,4]}
```

Figura 2 – Trecho de código da linguagem SOL

No entanto, tanto DEL como SOL tinha várias limitações como, pouco recurso

para construção de diálogos, pouca abstração de dados e incompleta se comparadas às linguagens contemporâneas a elas. Então Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes se juntaram para achar uma solução comum a seus problemas. As propostas de solução era formular uma nova linguagem de configuração genérica, que fosse facilmente acoplável, portável, simples e uma sintaxe fácil. Para o resultado desse projeto foi dado o nome LUA, como um contráste da antiga SOL.

As linguagens que mais se aproximam das características de Lua são o Icon, por sua concepção, e Python, por sua facilidade de utilização. Em um artigo publicado no Dr. Dobb's Journal, os criadores de Lua também afirmam que Lisp e Scheme foram de grande influência na decisão de desenvolver a tabela como a principal estrutura de dados de Lua. Lua tem sido usada em várias aplicações, tanto comerciais como não-comerciais.

Versões de Lua antes da versão 5.0 foram liberadas sob uma licença similar à licença BSD. A partir da versão 5.0, Lua foi licenciada sob a licença MIT.

Hoje a linguagem é uma das mais utilizadas do mundo estando entre as vinte mais utilizadas.

# 3 Aspectos léxicos e sintáticos de Lua

Esta capítulo descreve os principais aspectos léxicos, sintáticos e semânticos da linguagem Lua. sendo assim, serão descritas quais itens léxicos são válidos, como eles são combinados, e qual o significado da sua combinação.

O estudo de linguagens de programação pode ser orientado à verificação dos aspectos semânticos e sintáticos de uma linguagem. Pois a sintaxe é a forma das expressões e instruções, ou seja, como é feita a construção das mesmas. Não obstante, a semântica é o sforam ignificado das expressões e instruções.

## 3.1 Convenções Léxicas

Em Lua, os nomes podem ser qualquer cadeia de letras, dígitos, e sublinhados que não começam com um dígito, assim como em outras linguagens tradicionais, como C/C++. Os identificadores são usados para nomear variáveis e campos de tabelas.

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

As seguintes cadeias denotam outros itens léxicos: + - * == = <= >= < > = ( ) [ ] ; : , . .. ...

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

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

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

- and
- break
- do
- else
- elseif
- end
- false
- for
- function
- if
- in
- local
- nil
- not
- or
- repeat
- return

- then
- true
- until
- while

## 3.2 Sintaxe de Lua

Aqui está a sintaxe completa de Lua na notação BNF estendida. (Ela não descreve as precedências dos operadores.)

```
trecho ::= {comando [`;´]} [ultimocomando [`;´]]

bloco ::= trecho

comando ::=  listavar `=´ listaexp |
         chamadadefuncao |
         do bloco end |
         while exp do bloco end |
         repeat bloco until exp |
         if exp then bloco {elseif exp then bloco} [else bloco] end |
         for Nome `=´ exp `,´ exp [`,´ exp] do bloco end |
         for listadenomes in listaexp do bloco end |
         function nomedafuncao corpodafuncao |
         local function Nome corpodafuncao |
         local listadenomes [`=´ listaexp]

ultimocomando ::= return [listaexp] | break

nomedafuncao ::= Nome {`.´ Nome} [`:´ Nome]

listavar ::= var {`,´ var}

var ::=  Nome | expprefixo `[´ exp `]´ | expprefixo `.´ Nome

listadenomes ::= Nome {`,´ Nome}

listaexp ::= {exp `,´} exp

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

expprefixo ::= var | chamadadefuncao | `(´ exp `)´

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

```
args ::=  `(´ [listaexp] `)´ | construtortabela | Cadeia

funcao ::= function corpodafuncao

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

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

construtortabela ::= `{´ [listadecampos] `}´

listadecampos ::= campo {separadordecampos campo} [separadordecampos]

campo ::= `[´ exp `]´ `=´ exp | Nome `=´ exp | exp

separadordecampos ::= `,´ | `;´

opbin ::= `+´ | `-´ | `*´ | `/´ | `^´ | `%´ | `..´ |
          `<´ | `<=´ | `>´ | `>=´ | `==´ | `~=´ |
          and | or

opunaria ::= `-´ | not | `#´
```

# 4 Semântica das Variáveis

Este capítulo apresenta as questões fundamentais das variáveis. Como tipo, endereço e valores. Além da abordagem de vinculação e de escopo.

## 4.1 Variáveis

Uma variável em uma linguagem é a abstração do conteúdo de células de memória do computador. Variáveis podem se caracterizadas de acordo com os seguintes aspectos: nome, endereço, valor, tipo, tempo de vida e escopo.

Em Lua existem três tipos de variáveis, sendo elas as seguites: variáveis globais, variáveis locais e variáveis de tabelas. Sendo que, a diferença entre variáveis locais e globais é o uso da palavra reservada 'local' antes do nome da variável. Já as variáveis de tabela são os nomes dados aos índices das tabelas, tendo em vista que toda a estrutura de dados linguagem é orientada à tabelas.

## 4.2 Vinculação

O termo vinculação é uma associação ou uma referência, como, por exemplo, entre uma atributo e uma entidade e entre uma operação e um símbolo. O momento em que ocorre a vinculação é denominado como tempo de vinculação. Isso porquê, as vinculações podem ocorrer no tempo de projeto da linguagem, no tempo de implementação, no tempo de compilação, no tempo de ligação, no tempo de carregamento ou no tempo de execução. Um bom exemplo disso é que o operador '+' é vinculado no tempo de projeto da linguagem.

Lua é uma linguagem dinamicamente tipada. Isto significa que variáveis não possuem tipos, porém somente valores possuem tipos. Não existem definições de tipos na linguagem, pois todos os valores carregam o seu próprio tipo de dados. Logo Lua utilizaça um método de decleração implícita de variáveis.

A linguagem trabalha com vinculação dinâmica de tipos, logo o tipo não é especificado por uma instrução de declaração, como em C ou em Java. Em vez disso, a variável é vinculada a um tipo quando lhe é atribuida um valor em uma instrução de atribuição.

Este modelo apresenta muitas diferenças com relação aos tipos estaticamente vinculados. A principal vantagem de vinculação dinâmica de variáveis a tipos é que ele tráz muita flexibilidade para a programação.

## 4.3 Verificação de Tipos

A verificação de tipos é um módulo que assegura que os operandos de um operador sejam de tipos compatíveis.

Em Lua devido a vinculação dinâmica de tipos, permite somente a verificação dinâmica de tipos. No entanto há uma desvantagem, pois é melhor detectar erros durante a compilação do que na execução porque, quanto mais cedo feita a correção melhor, normalmente terá menos custo.

Sendo assim, a verificação de tipos em Lua é feita em tempo de execução pelo interpretador Lua.

## 4.4 Escopo

O escopo de uma variável em uma linguagem é a faixa de instruções na qual a variável é visível. Uma variável é visível em uma instrução se puder ser referenciada nessa instrução, ou seja se uma instrução conseguer ter acesso a essa variável no bloco ou setor em que se encontra.

Tendo em vista que existem dois tipos de escopo, sendo eles o escopo estático e o escopo dinâmico, Lua trabalha na modelagem de escopo léxico, logo baseia-se na sequência de chamadas de subprogramas. Dessa forma, o escopo pode ser determidado em tempo de execução.

Assume-se que toda variável é uma variável global a menos que ela seja explicitamente declarada como uma variável local. Variáveis locais possuem escopo léxico, por isso podem ser livremente acessadas por funções definidas dentro do seu escopo ou bloco.

Um bloco é uma lista de comandos; sintaticamente, um bloco é a mesma coisa que um trecho. Sendo que, o bloco pode ser explicitamente delimitado para produzir um único comando:

-comando ::= do bloco end

Blocos explícitos são úteis para controlar o escopo de declarações de variáveis, além de também usados às vezes para adicionar um comando return ou break no meio de outro bloco.

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

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

Figura 3 – Verificação de escopo em Lua

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

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

# 5 Tipos de Dados em Lua

Os tipos de dados de uma linguagem de programação, são um dos fatores de maior importância para determinar a facilidade com que os programas podem executar problemas do mundo real. Pois para algumas aplicações certos tipos de dados conseguem mapear bem um tipo de problema. Portanto é imprescindível que uma linguagem suporte vários tipos de dados e estruturas.

Existem oito tipos de dados básicos em Lua, são eles, nil, boolean, number, string, function, userdata, thread e table. Nil é o tipo do valor nulo, cuja propriedade principal é ser diferente de qualquer outro valor, ele geralmente representa a ausência de um valor útil. Boolean é o tipo dos valores false e true. Tanto nil como false tornam uma condição falsa, sendo que, qualquer outro valor torna a condição verdadeira. Number representa números reais (ponto flutuante de precisão dupla). O tipo string representa cadeias de caracteres.

O tipo userdata permite que dados C arbitrários possam ser armazenados em variáveis Lua. Este tipo corresponde a um bloco de memória e não tem operações pré-definidas em Lua. O tipo thread representa fluxos de execução independentes e é usado para implementar co-rotinas. Não se pode confundir o tipo thread de Lua com os processos leves do sistema operacional, pois Lua dá suporte a co-rotinas em todos os sistemas.

O tipo table implementa arrays associativos, isto é, arrays que podem ser indexados não apenas por números, mas por qualquer valor (exceto nil). Tabelas podem ser heterogêneas, ou seja, elas podem conter valores de todos os tipos (exceto nil). Tabelas são o único mecanismo de estruturação de dados em Lua, todavia, elas podem ser usadas para representar arrays comuns, tabelas de símbolos, conjuntos, registros, grafos, árvores, etc.

# 6 Expressões

Uma expressão em linguagens de programação é uma combinação de valores, variáveis, operadores e chamadas de funções que são interpretadas de acordo com as ordens de precedência e de associatividade particulares a uma determinada linguagem, que calcula e, em seguida, retorna um valor.

As expressões básicas em Lua, já vistas na sintaxe da linguagem são as seguintes:

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

Em virtude do que foi apresentado abordaremos agora algumas operações importantes que encontramos nas expressões.

## 6.1 Operadores Aritméticos

A linguagem Lua fornece os principais elementos de aritméticos binários como a adição (+), subtração (-), multiplicação (*), divisão (/), módulo (%) e exponenciação ($\hat{}$), além do aperador unário de negação (-). Se os operandos são números ou cadeias de caracteres que podem ser convertidas para números, então todas as operações possuem o seu significado usual.

## 6.2 Operadores Relacionais

Um operador relacional compara os valores de dois operandos, sendo que o valor da expressão relacional é booleano. Os operadores relacionais em Lua são:

- ==
- ≃
- <
- >
- <=
- >=

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

A igualdade (==) primeiro compara o tipo dos operandos envolvidos na comparação. Logo, se os tipos são diferentes, então o resultado é **false**. Caso contrário, os valores dos operandos são comparados. Números e cadeias de caracteres são comparados de maneira usual. Objetos (valores do tipo table, userdata, thread e function) são comparados por referência, porque dois objetos são considerados iguais somente se eles são o mesmo objeto. Sendo assim, todas as vezes que um novo objeto é criado este novo objeto é diferente de qualquer outro objeto que existia anteriormente.

## 6.3 Operadores Lógicos

Os operadores lógicos em Lua são **and**, **or** e **not**. Sendo que, todos os operadores lógicos consideram false e nil como falso e qualquer outro valor diferente como verdadeiro.

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

10 or 20 –> 10

10 or error() –> 10

nil or “a” –> “a”

nil and 10 –> nil

false and error() –> false

false and nil –> false

false or nil –> nil

10 and 20 –> 20

## 6.4 Concatenação e Operador de Comprimento

O operador de concatenação de cadeias de caracteres em Lua é denotado por dois pontos (..) . Se ambos os operandos são cadeias de caracteres ou números, então eles são convertidos para cadeias de caracteres.

Com base no que foi mencionado observe o seguinte exemplo:

```lua
local aluno = {} -- Declaração da tabela aluno
aluno.nome = "Luiz Marcos "
aluno.sobrenome = "Junqueira de Carvalho"
aluno.idade = 25

print("Nome completo: " .. aluno.nome .. aluno.sobrenome .. "\n" .. "Idade: " .. aluno.idade)
```

```
Nome completo: Luiz Marcos Junqueira de Carvalho
Idade: 25
```

Figura 4 – Concatenação de variáveis com cadeia de caracteres

O operador de comprimento é denotado pelo operador unário #. O comprimento de uma cadeia de caracteres, por exemplo é o seu número de bytes. O comprimento de uma tabela t é definido como qualquer índice inteiro n tal que t[n] não é nil e t[n+1] é nil, além disso, se t[1] é nil, n pode ser zero. Para um array comum, com todos os valores diferentes de nil indo de 1 até n, o seu comprimento é exatamente n. Se o array possuir intervalos descontínuos, então #t pode ser qualquer um dos índices que imediatamente precedem um valor nil.

## 6.5 Precedência

A precedência de operadores em Lua segue a *Tabela 1* na página seguinte, na ordem de menor prioridade para a maior:

Como padrão, não existe a necessidade de usar parênteses para mudar as precedências de uma expressão. Os operadores de concatenação e de exponenciação são associativos à direita. Todos os demais operadores binários são associativos à esquerda.

| **Tabela de Precedência** |
|---|
| or |
| and |
| $< ou > ou <= ou >= ou \tilde{=} ou ==$ |
| .. |
| $+ ou -$ |
| $* ou / ou \%$ |
| not $ou \# ou$ -(unary) |
| $\hat{}$ |

Tabela 1 – Tabela de precedência em Lua

## 6.6 Definições e Chamadas de Função

A sintaxe para a definição de uma função é

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

Podemos verificar em Lua que o comando **function f () corpo end** é traduzido para **f = function () corpo end**

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

Uma chamada de função em Lua tem a seguinte sintaxe:

chamadadefuncao ::= exppprefixo args

Em uma chamada de função, primeiro **exppprefixo** e **args** são avaliados. Se o valor de **exppprefixo** possui tipo function, então esta função é chamada com os argumentos fornecidos, conforme o exemplo abaixo:

```
function function_example(args)
    print "chamada de função"
    return args
end
```

Figura 5 – Chamada de Função em Lua

Uma chamada de funcao da forma return é denominada de chamada final. Portanto, Lua implementa chamadas finais próprias em uma chamada final, logo a função chamada reusa a entrada na pilha da função que a chamou. Sendo assim, não há limite no número de chamadas finais aninhadas que um programa pode executar. Note que uma chamada final somente acontece com uma sintaxe particular, onde o return possui uma única chamada de função como argumento.

# 7 Sentenças de Atribuição

A instrução de atribuição é uma das instruções fundamentais das linguagens imperativas, pois oferece um mecanismo por meio do qual o usuário pode modificar dinâmicamente as vinculações e os valores das variáveis.

Lua permite atribuições simples e mútiplas. Em virtude disto, a sintaxe para a atribuição define uma lista de variáveis no lado esquerdo e uma lista de expressões do lado direito. Sendo que os elementos em ambos os lados são separados por vírgula. Logo:
**Comando de Atribuição -> Lista de Variáveis = Lista de Expressões**

Contudo, antes da atribuição de fato ser realizada, a lista de valores a ajustada de acordo com o comprimento da lista de variáveis. Sendo assim, se houver mais valores do que variáveis esses valores são descartados, caso contrário, se houver menos valores a lista é completada com valores nulos (nil).

É importante lembrar que Lua não permite atribuições unárias como C, C++ ou Java que incluem dois operadores aritméticos unários, sendo eles ++ para incremento e – para decremento. Para isso a linguagem utiliza outro recurso de atribuição que é a atribuições de expressões, como por exemplo, *var = var + 1* ou *var = var -1*. Além disso é permitido atribuições de mútiplas expressões a múltiplas variáveis.

A semântica de atribuições para variáveis globais e campos de tabelas pode ser mudada através do uso de metatabelas. No anexo xxx são apresentados alguns exemplos de atribuições em Lua.

# 8 Estruturas de Controle

A definição de estrutura de controle é conceituada da seguinte forma, uma instrução de controle e sua coleção de comandos cuja a execução é controlada pela própria estrutura.

Em Lua as estruturas de controle **if**, **while** e **repeat** possuem o significado usual e a sintaxe familiar, conforme abaixo:

- comando ::= **while** exp **do** bloco **end**
- comando ::= **repeat** bloco **until** exp
- comando ::= **if** exp **then** bloco **elseif** exp **then** bloco [**else** bloco] **end**

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

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

O comando **return** é usado para retornar valores de uma função ou de um trecho de função. Sendo que, funções e trechos podem retornar mais de um valor, de modo que a sintaxe para o comando return é:

- comando ::= return [listaexp]

O comando **break** é usado para terminar a execução de um laço **while**, **repeat** ou **for**, pulando para o próximo comando depois do laço. Um break termina a execução do laço mais interno.

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

Lua possui duas variações do comando **for**, sendo uma numérica e outra genérica. O laço for numérico repete um bloco de código enquanto uma variável de controle varia de

acordo com uma progressão aritmética. Já o laço for genérico funciona utilizando funções, chamadas de iteradoras. A cada iteração, a função iteradora é chamada para produzir um novo valor, parando quando este novo valor é nil.

Além disso o **for** em Lua pode iterar específicamente sobre os elementos de uma tabela Lua de duas formas distintas, utilizando o modelo genérico já apresentado. Conforme abaixo:

```
1   local tabela = {"um", "dois", "tres", "quatro", "cinco"}
2
3   tabela["index"] = 20
4   tabela["value"] = 50
5
6   for i,v in ipairs(tabela) do
7       print(i,v)
8   end
9
10  print "\n"
11
12  for k,v in pairs(tabela) do
13      print(k,v)
14  end
```

```
1   um
2   dois
3   tres
4   quatro
5   cinco


1   um
2   dois
3   tres
4   quatro
5   cinco
index   20
value   50
[Finished in 0.1s]
```

Figura 6 – Exemplo de iteradores em tabelas Lua

Conforme visto acima o primeiro iterador utiliza a palavra reservada **ipairs**, isso significa que o **for** irá iterar obedecendo a ordenação e sequenciamento dos índices. Por isso, no log de execução do primeiro iterador não é verificada a vazualização dos dois últimos valores da tabela, já que os mesmo possuem índices no formado de cadeia de caracteres. Em contrapartida, o segundo exemplo do for itera sob todos o índices da tabela independentemente do sequenciamento.

É necessário lembrar que todas as estruturas de controle são adecuadas às regras de escopo das variáveis e regras de visibilidade.

# 9 Orientação à Tabelas

Lua é uma linguagem multiparadigma, que contém fortemente implementados os paradigmas de programação imperativa e programação procedural, baseada no conceito de chamadas de procedimentos. Além desses dois paradigmas, Lua também tráz características de abstração de dados, que por sua vez é muito utilizado na orientação a objetos, este recurso de abstração concedido por meio das metatabelas.

Conforme já foi definido no capítulo de tipos de dados, o tipo **table** em Lua utiliza arrays associativos. Tabelas podem ser heterogêneas, ou seja, elas podem conter valores de todos os tipos, excetuado pelo valor nil. Sendo que, as tabelas são o único mecanismo de estruturação de dados em Lua, todavia, elas podem ser usadas para representar arrays comuns, tabelas de símbolos, conjuntos, registros, grafos, árvores, e outros.

## 9.1 Abstração de dados

Como o tipo **table** é o único recurso de estrutura de dados em Lua, toda a abstração de dados da linguagem utiliza os recursos da tabela. Por esse motivo é possível implementar um método de desenvolvimento em Lua orientado a objetos ou a eventos.

No exemplo abaixo podemos verificar a aplicação de tabelas para criar classes e objetos em Lua.

```lua
Classe = {} -- É NECESSÁRIO A CRIAÇÃO DE UMA TABELA GLOBAL, QUE FARÁ O PAPEL DA CLASSE.

function main()
    local objeto = Classe:new() -- CHAMA O MÉTODO NEW PARA A CRIAÇÃO DO OBJETO.

    objeto:imprimirAtributos() -- EXECUTA O MÉTODO DA CLASSE PASSANDO O OBJETO COMO PARÂMETRO
end

function Classe:new()
    local instance = {}
    setmetatable(instance,self) -- AQUI É FEITA A INSTÂNCIA DA METATABELA
    self.__index = Classe

    instance.nome = "Pedro Martins"
    instance.idade = 20
    return instance
end

function Classe:imprimirAtributos()
    print(self.nome)
    print(self.idade)
end

main() -- INÍCIO

```

Figura 7 – Exemplo de iteradores em tabelas Lua

## 9.2 Manipulação de Tabelas

Em lua há uma biblioteca que provê funções genéricas para manipulação de tabelas, sendo que, ela provê todas as suas funções na tabela **table**.

A maioria das funções na biblioteca de tabelas assume que a tabela representa um array ou uma lista. Dessa forma são feitas as seguintes funções:

- table.concat - Concatena elementos dado um array onde todos os elementos são cadeias ou números.
- table.insert - Insere um elemento na tabela, na posição em que o programador desejar, deslocando os outros elementos para abrir espaço, se necessário.
- table.maxn - Retorna o maior índice numérico positivo da tabela fornecida ou zero se a tabela não possui índices numéricos positivos.
- table.remove - Remove de tabela o elemento na posição desejada pelo programador, deslocando os outros elementos para preencher o espaço, se necessário. Retorna o valor do elemento removido.

  table.sort - Ordena os elementos da tabela em uma dada ordem, de table[1] até table[n], onde n é o comprimento da tabela. O algoritmo de ordenação não é estável, isto é, elementos considerados iguais pela ordem fornecida podem ter suas posições relativas trocadas pela ordenação.

## 9.3 Metatabelas

Todos os valores em Lua pode possuir uma metatabela, que na verdade é uma tabela comum de Lua que define alguns comportamentos com relação a certas operações iniciais. Com isso, é possível alterar diversos aspectos do comportamento de operações, especificando campos específicos na metatabela do valor. Por exemplo, quando um valor não numérico é um operando de uma adição, Lua verifica se existe uma função associada com o campo "__add" na metatabela. Logo, se a função existir, o interpretador Lua chama esta função para realizar a adição.

Chamamos as chaves de uma metatabela de eventos e os seus valores de metamétodos. No exemplo anterior, o evento é a operação "add" e o metamétodo é a função que realiza a adição. Sendo assim, é possível obter a metatabela de qualquer valor usando a função **getmetatable**, e também pode ser alterada a metatabela através da função **setmetatable**. Entretanto metatabela não pode ser modificada para outros tipos de Lua (a menos que você use a biblioteca de depuração), neste caso obrigatoriamente deve ser usada a API C.

Tabelas e objetos do tipo **userdata** possuem metatabelas individuais. No entanto, valores de todos os outros tipos compartilham uma única metatabela por tipos, ou seja, há somente uma metatabela para todos os números, uma para todas as cadeias de caracteres e etc.

Uma metatabela controla como um objeto se comporta em operações aritméticas, comparações com relação à ordem, concatenação, operação de comprimento e indexação. Uma metatabela também pode definir uma função a ser chamada quando um objeto userdata é coletado pelo coletor de lixo. Para cada uma destas operações Lua associa uma chave específica chamada um evento.

# 10 Facilidades e Bibliotecas Padrões

Este capítulo tratará de alguns aspectos específicos de padrões no projeto da linguagem Lua, bom como suas características homogêneas.

## 10.1 Tratamento de Erros

Como Lua é uma linguagem de tipo embarcada de extensão, todas as suas ações de começam a partir de um código C, no programa hospedeiro que chama uma função da biblioteca Lua. Então, sempre que um erro ocorre durante a compilação ou execução, o controle retorna para o código C, que pode tomar as medidas apropriadas.

Um código Lua pode explicitamente gerar um erro através da chamada à função **error**. Caso, necessite capturar erros em Lua, pode ser usada a função **pcall**.

## 10.2 Ambientes

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

Objetos do tipo **thread** são criados compartilhando o ambiente da thread que os criou. Objetos do tipo **userdata** e funções C são criados compartilhando o ambiente da função C que os criou. As funções Lua não aninhadas (criadas por loadfile, loadstring ou load) são criadas compartilhando o ambiente da thread que as criou. E funções Lua aninhadas são criadas compartilhando o ambiente da função Lua que as criou.

O ambiente associado a uma função em C pode ser diretamente acessado pelo código fonte C. Isso é usado como o ambiente padrão para outras funções C e objetos **userdata** criados pela função.

É possível mudar o ambiente de uma função Lua ou do fluxo de execução que está sendo executado atualmente chamando **setfenv**. Além disso, é possível obter o ambiente de uma função Lua ou do fluxo de execução sendo executado atualmente chamando **getfenv**.

## 10.3 Coleta de Lixo

Lua funciona com um gerenciamento automático da memória. Isto significa que não é necessário se preocupar com a alocação de memória para novos objetos, nem com

a liberação de memória quando os objetos não são mais necessários. Pois, a linguagem Lua gerencia a memória automaticamente executando um coletor de lixo de tempos em tempos para coletar todos os objetos mortos. Toda memória usada por Lua está sujeita ao gerenciamento automático de memória: tabelas, userdata, funções, fluxos de execução, cadeias de caracteres e etc.

Lua implementa um coletor de lixo marca-e-limpa (mark-and-sweep) incremental. Sendo que, o coletor usa dois números para controlar o seu ciclo de coleta de lixo, a pausa do coletor de lixo e o multiplicador de passo do coletor de lixo.

A pausa do coletor controla quanto tempo irá esperar antes de iniciar um novo ciclo. O multiplicador de passo controla a velocidade relativa do coletor em relação à alocação de memória. É possível mudar estes números através de chamadas às funções **lua_gc** em C ou **collectgarbage** em Lua. Com estas funções você também pode controlar o coletor diretamente.

# 11 Conclusão

# Referências

IERUSALIMSCHY, L. H. d. F. R. *Manual Lua*. 2006. Disponível em: <http://www.lua-.org/manual/5.1/pt/>. Citado na página 7.

SEBESTA, R. W. *Conceitos de Linguagens de Programação*. 5ª. ed. Porto Alegre: Bookman, 2003. Citado na página 7.