# Curso de Lua 3.2


Fábrica Digital

# Índice

# Introdução

## O que é Lua?

- Uma linguagem de script
- Uma linguagem de extensão
- implementada como uma biblioteca em C

## Por que Lua?

- Simples e flexível
- Pequena
- Eficiente
- Portável
  - **A biblioteca inteira escrita em ANSI C, compilando o mesmo código-fonte em qualquer plataforma**

## Execução

O interpretador de Lua *command-line* e as bibliotecas para incluir Lua em programas C podem ser baixados em: http://www.lua.org (buscar a versão 3.2. A atual é 5.0)

CGILua 3.2 pode ser baixado em http://www.tecgraf.puc-rio.br/cgilua

Para rodar um programa Lua com o interpretador *command-line*, basta digitar:
lua nomedoarquivo.lua

Ao rodar o interpretador sem parâmetros, ele funciona em modo interativo.

## Observações

- Não há formatação rígida para o código Lua. Tanto faz colocar um comando por linha ou diversos comandos em uma mesma linha, que podem ser separados por espaço ou ponto-e-vírgula.
- Lua é case-sensitive
- Comentários em Lua iniciam-se com traços duplos (--) e seguem até o final da linha.
- Possui gerenciamento automático de memória e coleta de lixo.

# Variáveis e Escopo

## Tipos de Dados

- *nil*
- number
- string
- function
- userdata
- table

, onde:

- ***nil*:** representa o valor indefinido ou inexistente.

  Ex: if(a)...
  Testa a existência de valor na variável a, ou seja, se seu valor é diferente de *nil*
  Diferente de qualquer outro valor
  Variáveis não inicializadas recebem o valor *nil*
  - local a equivale a local a = *nil*
  - Expressão (a), caso a ainda não tenha sido inicializada no escopo com valor não-*nil*, retorna falso

  *Ex: a = nil*

  ```
  if (a) then
    …
  else
    …
  end
  -- entrará no else
  ```

- **number:** Representa valores numéricos, sejam inteiros ou reais.
  Internamente, são números de ponto flutuante com dupla precisão (*double* em C).
  *Ex: pi = 3.14*

- **string:** Representa uma cadeia de caracteres.

Delimitadores:
- Aspas duplas: útil quanto se necessita de aspas simples na string.
  *Ex: a = "Isto é um teste"*
- Aspas simples: útil quanto se necessita de aspas duplas na string.
  *Ex: a = 'Isto é uma "citação".'*
- Duplos colchetes: útil quando se necessita escrever a string em mais de uma linha. As quebras de linha farão parte da string.

  *Ex: a = [[Esta é uma string*
  *em mais de uma linha]]*
  *•Caracter \ indica seqüências de escape, como em C*
  *Ex: \n \t \r \" \' *

- **function:** Funções em Lua são do tipo de dados *function.*
  É possível armazená-las em variáveis, em índices de tabelas, passá-las como parâmetros para outras funções ou retorná-las como resultados.

  *Ex: function func1() … end (ou func1 = function() … end)*
  *func1 é uma variável que armazenada uma função, ou seja, um valor do tipo function*

  *Para executar a função, basta chamar: func1()*

- **userdata:** Armazena um ponteiro do tipo void* de C.
  Útil para quem utiliza Lua dentro de um programa em C.

- **table:** Implementa vetores associativos – valores indexados por qualquer outro tipo (exceto *nil*).
  Principal mecanismo de estrutura de dados de Lua.
  Uma variável contém uma referência a uma tabela, e não a própria tabela.
  Designação, passagem de parâmetros e retorno manipulam referências a tabelas, não implicando em cópia.
  Tabelas devem ser criadas antes de serem utilizadas.

  *Ex: a = {}*
  *a[1] = "Teste"*
  *a["nome"] = "João" (equivale a a.nome = "João")*
  *a.func1 = function() … end*
  *b = {1, "casa"; nome1 = "João"}*
  *-- resulta em: b[1] = 1, b[2] = "casa",*
  *-- b.nome1 (ou b["nome1"]) = "João"*

Variáveis não são tipadas em Lua, somente os valores armazenados nas mesmas o são
A seguinte seqüência de instruções é válida:

```
a = nil
-- armazena em a um dado do tipo nil
a = 1
-- passa a armazenar em a um dado do tipo number
a = "Teste"
-- passa a armazenar em a um dado do tipo string
```

# Variáveis Globais

Não precisam ser declaradas, podem ser utilizadas diretamente.

```
Ex: a = 2
-- a não foi declarada anteriormente. Recebe o valor 2 e é visível em todo o programa
b = (c == nil)
-- b e c não foram declaradas anteriormente. b recebe o valor 1 e é visível em todo o
programa. c continua indefinida (ou seja, nil)
```

# Variáveis Locais

Podem ser declaradas em qualquer lugar dentro de um bloco de comandos.
Têm escopo somente dentro daquele bloco.
Quando declaradas com o mesmo nome que uma global, encobrem o acesso à global naquele bloco.

```
Ex:  local a                    -- a é declarada e recebe nil

Ex:  local a, b = 1, "x"        -- declara a e b, onde a = 1 e b = "x"


Ex:  a = 2                      -- variável global a recebe 2
     if (a > 0) then
       local b = a              -- variável local b recebe valor da global a
                                -- (dentro do escopo do if)
      a = a + 1                 -- altera o valor da variável global a
       local a = b              -- declara variável local a que recebe valor de b
       print(a)                 -- imprime valor de variável local a
                                -- (que obscureceu a global a)
     end
     print (a)                  -- imprime o valor da variável global a
                                -- (a local já não existe)
```

# Upvalues

Uma função pode acessar suas próprias variáveis locais (incluindo parâmetros passados a ela) ou variáveis globais (quando não encobertas por locais).

Uma função não pode acessar variáveis locais de um escopo externo, já que essas variáveis podem não mais existir quando a função é chamada.

```
Ex:  function f()
       local a = 2
       local g = function()
         local b = a -- erro: a é definida num escopo externo
       end
     end
```

Porém, pode acessar o valor de uma variável local de um escopo externo, utilizando upvalues que são valores congelados quando a função dentro de onde eles aparecem é instanciada. Para acessar esses valores, utiliza-se a notação % seguido do nome da local do escopo externo.

```
Ex:  function f()
       local a = 2
       local g = function()
         local b = %a           -- ok, acessa valor congelado de a no momento
                                -- em que a função foi instanciada
       end
     end
```

# Exercícios de Fixação I

**1) O que é impresso na tela após a execução do seguinte trecho de código Lua?**

```
x = 10
if (x <= 10) then
   local x
   x = 20
   print(x)
end
if (x == 10) then
        x = x + 2
        print(x)
end
print(x)
```

**2) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
y = 10
if (y <= 10) then
   local x = 20
   print(x)
end
print(x)
if (y == 10) then
        local y = 2
        y = y + 2
        print(y)
end
print(y)
```

**3) O que é impresso na tela após a execução do seguinte trecho de código Lua?**

```
a = "Isto é um 'teste'!!!"
b = [[Isto é um
"teste"!!!]]
c = 'Isto é um\n"teste"!!!'
print(a)
print(b)
print(c)
```

**4) Qual dos seguintes trechos de código Lua é inválido?**

```
x = [[O "rato" roeu a 'roupa' do [rei] de Roma.]]
x = 'O "rato" roeu a roupa do [[rei]] de Roma.'
x = "O rato roeu a 'roupa' do [[rei] de Roma."
x = [[O "rato" roeu a 'roupa' do [rei] de Roma.]]
x = "O 'rato' roeu a "roupa" do [rei] de Roma."
```

# Atribuição e Operadores

## Atribuição Simples e Múltipla

Simples:

*Ex: a = 2*

Múltipla:

```
Ex: a, b = "Teste", 3
        -- a recebe o valor "Teste" e b recebe 3
Ex: a, b = "Teste"
        -- a recebe o valor "Teste" e b recebe nil
Ex: a, b = "Teste", 3, 5
        -- a recebe o valor "Teste",  b recebe 3
        -- e o valor 5 é desprezado
Ex: a, b = b, a
        -- Troca os valores de a e b entre si
```

## Operadores Aritméticos

São operadores aritméticos de Lua:

- Soma (+)
- Subtração (-)
- Multiplicação (*)
- Divisão (/)
- Operador unário de negativo (-)
- Parênteses podem alterar precedência de operadores.

Somente podem ser aplicados a valores do tipo number ou a valores do tipo string que possam ser convertidos para number.

```
Ex: a = 2 * 3.5 + 4
    -- armazena 11 em a
    b = (a + 1) / "2"
    -- armazena 6 em b.
    -- (repare na string "2" convertida para o número 2)
    c = b * (-3)
    --armazena -18 em c
```

## Operadores Relacionais

São operadores relacionais de Lua:

- Menor que (<)
- Maior que (>)
- Menor ou igual a (<=)
- Maior ou igual a (>=)
- Igual a (==)
- Diferente de (~=)

Operadores <, >, <=, >= são aplicáveis a number e string.

Operadores = = e ~ = compara os tipos dos dados (retorna *nil* caso falso) e, em seguida, os valores. Tabelas, funções e userdata são comparados por referência.

Retornam *nil* caso falso e 1 caso verdadeiro.

```
Ex:  a = 4 > 3
     -- armazena 1 em a
     b = (a = = 2)
     -- armazena nil em b
     c = (b = = {1})
     -- armazena nil em c
```

# Operadores Lógicos

São operadores lógicos de Lua:
- Conjunção (and)
- Disjunção (or)
- Negação (not)

Operadores *and* e *or* são avaliados da esquerda para a direita.

Avaliação do tipo *curto-circuito* (pára ao saber resultado).

```
Ex:  a = 10
     b = 20
     c = (a < 4) or (b < a)
     -- armazena nil em c avaliando toda a expressão
     d = (a < 4) and (b < a)
     -- armazena nil em d sem precisar avaliar o (b < a)
```

# Operador de Concatenação

É representado por dois caracteres ponto (..).

Aplicável a tipo *string*, convertendo valores do tipo *number* quando concatenados a *string*.

```
Ex:  a = "Linguagem"
     b = "Lua"
     c = 3.2
     c = a .. " " .. b .. c
     -- armazena "Linguagem Lua 3.2" em c
```

# Exercícios de Fixação II

**5) O que é impresso na tela após a execução do seguinte trecho de código Lua?**

```
x = 10
a, b, x = x, x*2, x*3
print(a)
print(b)
print(x)
```

**6) O que é impresso na tela após a execução do seguinte trecho de código Lua?**

```
a, b, c = 0, 1, "teste", 100
a, b, c = c, a
print(a)
print(b)
print(c)
```

**7) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
a = 4
b = 5
print(a and b)
print(c and b)
print(a or b)
print(c or b)
```

**8) O que é impresso na tela após a execução do seguinte trecho de código Lua?**

```
a = 10
b = 20
c = 30
print(((a = = 5) and b) or c)
print(((a = = 10) and b) or c)
```

**9) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
a = 10
b = 20
c = "10"
print(a = = b)
print(a < = b)
print(d = = e)
print(a = = c)
```

**10) O que é impresso na tela após a execução do seguinte trecho de código Lua?**

```
print(0 .. 1)
print(0 + 1)
print("Repita ".. 10 ..[[ vezes.]])
print(10 .. "" = = "10")
```

# Estruturas de Controle de Fluxo

## Expressões

- Expressões podem retornar qualquer valor de qualquer tipo.
- Sempre que retornar valores diferentes de *nil*, a expressão é considerada verdadeira.
- Sempre que retornar o valor *nil*, a expressão é considerada falsa.

## Tomada de Decisão

```
if expr then
   bloco
end

if expr then
   bloco1
else
   bloco2
end

if expr1 then
   bloco1
elseif expr2
   bloco2
…
elseif exprN
   blocoN
else
   blocoN + 1
end
```

*Ex:*
```
if (a = = 5) then
   print("a = 5")
end
```

*Ex:*
```
if b then
   print("b diferente de nil")
else
   print("b igual a nil")
end
```

*Ex:*
```
if (not a) then
   print("a igual a nil")
elseif (a = = 1) then
   print("a = 1")
else
   print("a é maior que 1")
end
```

# Laços Iterativos

Tomada de decisão no começo:

```
while expr do
  bloco
end

Ex:  i = 10
                while (i >= 0) do
                  i = i – 1
                end
```

Tomada de decisão no fim:

***repeat***
  bloco
***until*** expr

```
Ex:  i = 10
     repeat
       i = i – 1
     until (i == 0)
```

Não existem laços do tipo for em Lua 3.2.

Veremos ainda na parte de tabelas: foreach e foreachi.

# Exercícios de Fixação III

**11) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
a = 10
if (a) then
        a = a / 2
else
        a = a * 2
end
print(a)
b = not a
if (b) then
        a = a / 2
else
        a = a * 2
end
print(a)
```

**12) Qual é o problema com o seguinte trecho de código em Lua?**

```
i = 0
while (i < 5) do
print(i)
end
```

**13) O que é impresso na tela após a execução do seguinte trecho de código Lua?**

```
i = 0
while (i < = 4) do
        print(i)
        i = i + 1
end
```

**14) O que é impresso na tela após a execução do seguinte trecho de código Lua?**

```
i = 0
repeat
        print(i)
        i = i + 1
until (i > 4)
```

**15) Qual dos seguintes programas completos em Lua imprimem os números ímpares até 9?**

```
a) i = 0
repeat
        print(i)
        i = i + 2
until (i = = 9)
```

```
b)  i = 1
repeat
            print(i)
          i = i + 2
until (i < 11)

c) i = 1
while (i < 11) do
            print(i)
          i = i + 2
end

d) i = 1
while (i < = 9) do
          i = i + 2
          print(i)
end
```

# Tabelas

- Tabelas em Lua são do tipo de dados *table.*
- Implementam *vetores associativos* – valores indexados por qualquer outro tipo (exceto *nil*).
- Principal mecanismo de estrutura de dados de Lua.
- Uma variável contém uma referência a uma tabela, e não a própria tabela.
  - **Designação, passagem de parâmetros e retorno manipulam referências a tabelas, não implicando em cópia**
- Tabelas devem ser criadas antes de serem utilizadas

## Construtores

Construtores são expressões que criam tabelas.

Construtores podem criar tabelas:

- Vazias: utiliza-se abre e fecha chaves.
  *Ex: a = {}*
- Com elementos inicializados indexados numericamente consecutivamente.
- Valores separados por vírgula dentro de chaves.
- O primeiro índice é sempre 1 (e não 0).
  *Ex: b = {1, 3.8, "Teste"}*
  *-- nesse caso, b[1] = 1, b[2] = 3.8 e b[3] = "teste"*
- Com elementos inicializados indexados por qualquer outro tipo.
  *Ex: c = {c1 = "v1", c2 = "v2", [3] = "v3"}*
  *-- nesse caso, c["c1"] = "v1", c["c2"] = "v2" e c[3] = "v3"*
- Com elementos inicializados indexados das duas formas. Separa-se os dois tipos de indexação com um ponto-e-vírgula. Já os elementos com o mesmo tipo de indexação são separados por vírgula.
  *Ex: d = {1, 3.4; d1 = "x1", d2 = "x2", [3] = 8}*
  *-- nesse caso, d[1] = 1, d[2] = 3.4, d["d1"] = "x1", d["d2"] = "x2" e d[3] = 8*

Após criada a tabela, pode-se armazenar outros valores na tabela, com qualquer tipo de indexação.

## Acessando Dados

Para acessar índices numéricos em tabelas, utiliza-se o número entre colchetes.

```
Ex: a = {1.3, 3.14}
    print(a[1])          -- imprime 1.3
    print(a[2])          -- imprime 3.14
    a[3] = 12            -- adiciona uma entrada na tabela
```

Para acessar índices string em tabela, é tanto válido utilizar a string entre aspas e colchetes quanto utilizar a notação com ponto. Desta forma, podem-se simular campos/registros.

*Ex:* *a = {campo1 = 1.3, campo2 = 3.14}*
*print(a["campo1")* *-- imprime 1.3*
*print(a["campo2"])* *-- imprime 3.14*
*print(a.campo1)* *-- imprime 1.3*
*print(a.campo2)* *-- imprime 3.14*
*a.campo3 = 12* *-- adiciona uma entrada na tabela*

Quando houver espaços na string de índice, a notação de ponto não é válida.
*Ex.: a["Isto é um teste"]* não tem equivalência com uso de ponto.

Para limpar uma entrada de uma tabela, basta igualá-la a *nil*.
*Ex:* *a.campo3 = nil*

# Exercícios de Fixação IV

**16) Represente em Lua a seguite matriz, em uma tabela t, utilizando somente um constructor.**

11 12 13 14

21 22 23 24

31 32 33 34

Imprima também os valores da terceira linha.

**17) Represente em Lua os seguinte registros de banco de dados. Primeiramente em uma tabela t1 utilizando somente um construtor, e em seguida em uma tabela t2, construindo uma tabela vazia e a preenchendo campo a campo.**

nome: João da Silva
endereco: Rua A, casa B
telefone: 2222-2222

nome: Maria Joaquina
endereco: Rua C, casa D
telefone: 3333-3333

nome: Pedro Joaquim
endereco: Rua E, casa F
telefone: 4444-4444

**18) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
a = { }
b = 1
c = "b"
a[1] = 100
a[c] = 10
a["b"] = a["b"] + 10
a[b] = a[b] + 100
print(a[b])
print(a[c])
```

# Funções

Funções em Lua são do tipo de dados *function.* É possível armazená-las em variáveis, em índices de tabelas, passá-las como parâmetros para outras funções ou retorná-las como resultados.

## Declaração

Duas sintaxes possíveis para declaração:

```
function f(lista-de-parâmetros)
        bloco
    end
```

```
f = function(lista-de-parâmetros)
        bloco
end
```

f representa a variável que armazenará a função.

```
Ex:  function soma(a, b)
       -- é o mesmo que: soma = function(a, b)
       return a + b
     end
```

Lua faz ajuste no número de parâmetros passados à função, completando com *nil* os parâmetros faltantes.

```
Ex:  function func1(a, b, c, d)
       bloco
     end

     -- chamada da função:
     func1(1, 2)
     -- dentro da função: a = 1, b = 2, c = d = nil

     -- chamada da função:
     func1(1, 2, 3, 4, 5, 6)
     -- dentro da função: a = 1, b = 2, c = 3, d = 4
     -- os valores 5 e 6 passados à função são ignorados
```

- **Número variável de argumentos:**

Ao final da lista de argumentos, adicionar … (três pontos seguidos)
Argumentos extras são colocados na tabela implícita arg

```
Ex:  function func1(a, b, …)
       bloco
     end
     -- chamada da função:
     func1(1, 2, 3, 4, 5)
     -- dentro da função: a = 1, b = 2, arg = {3, 4, 5}
     -- ou seja, arg[1] = 3, arg[2] = 4, arg[3] = 5
```

- **Passagem de parâmetros por valor**

Valores do tipo string e number são passados por valor, ou seja, uma cópia local dos mesmos é criada e as variáveis externas não tem seu valor alterado

```
Ex:  function func1(a)
         a = 1
     end
```

```
x = 10
func1(x)
-- o valor de x não é alterado
```

- **Passagem de parâmetros por referência**

Valores do tipo table, function e userdata são passados por referência, ou seja, é passado para a função um ponteiro para os mesmos na verdade

```
Ex: function func1(a)
        a[1] = 1
    a[2] = 2
end
x = {3, 4}
func1(x)
-- resulta em: x[1] = 1 e x[2] = 2
```

# Retorno

- **Comando return**

Efetua o retorno da função imediatamente.
Deve ser o último comando do bloco.
Pode retornar zero, um ou mais valores (múltiplo retorno).

```
Ex: function func1(param1)
      if (not param1) then
        return
      end
      print(param1)
    end
    -- somente imprimirá o valor de param1 se não for nil

Ex: function func1()
      return 1, 2, "teste"
    end
    x, y, z = func1()
    -- resulta em: x = 1, y = 2 e z = "teste"
```

Lua faz ajuste no número de parâmetros retornados da função, completando com *nil* os parâmetros faltantes.

```
Ex: function func1()
      return 1, 2
    end

    -- chamada da função:
    a, b, c, d = func1()
    -- resulta em: a = 1, b = 2, c = d = nil

    -- chamada da função:
    a = func1()
    -- resulta em: a = 1
    -- o valor 2 retornado é ignorado
```

# Chamada

As funções em Lua são chamadas pelo nome da função seguido dos argumentos entre parêntesis, separados por vírgulas:

*Ex: func1(1, 2, 3)*

Para funções que aceitam como parâmetro um único parâmetro que é uma tabela, é aceita a chamada sem parêntesis para uma nova tabela.

*Ex: func2 {10, "teste", 8.5}*

Para funções que aceitam como parâmetro um único parâmetro que é uma string, é aceita a chamada sem parêntesis para uma nova string.

*Ex: print "Isto é um teste"*

# Exercícios de Fixação V

**19) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
function func1(a, b, c)
        return c, b, a
end
local x, y = func1(1, 2)
print(x)
print(y)
```

**20) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
function somaum(a)
        a = a + 1
end
a = 10
somaum(a)
print(a)
```

**21) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
function somaum(a)
        a[1] = a[1] + 1
        a[2] = a[2] + 1
        a[3] = a[3] + 1
end
a = {10, 20, 30}
somaum(a)
print(a[1])
print(a[2])
print(a[3])
```

**22) Escreva uma função eq2 em Lua que retorne o resultado de uma equação de, no máximo, segundo grau (na forma y = ax2 + bx + c), dados a, b, c e x. Os parâmetros a, b e c devem ser opcionais, de forma a permitir a solução de equações de grau um e zero.**

**23) Escreva uma função fact em Lua, que retorne o fatorial de um dado n.**

# Funções Pré-Definidas

## call

call (func, arg [, mode])

Chama a função *func* passando a tabela *arg* como uma lista de argumentos. Equivale a: func(arg[1], arg[2], ... , arg[n]).

ARGUMENTOS
*func*: função a ser chamada.
*arg*: tabela que contém a lista de parâmetros.
**mode (opcional): se for "p", empacota resultados em uma única tabela**

RETORNO
O retorno da função *func* (caso mode seja diferente de "p") ou o retorno da função *func* empacotado em uma única tabela (caso mode seja "p").

```
EXEMPLO:
table1 = {91, 234}
f = function(a, b, c, d)
   return 12, {"um", "dois"}, {"três"}
end
result1 = call(f, table1)
-- result1 é 12 ( {"um", "dois"} e {"três"} são ignoradas)
-- (equivale a result1 = f(table1[1], table1[2]))

result 2 = call(f, table1, "p")
-- result2 é {12, "um", "dois", "três"}
-- (tudo empacotado em uma única tabela)
```

## dofile

dofile (filename)

Recebe um nome de arquivo, abre e executa o arquivo como um módulo de Lua dentro do código em execução.

Útil para separar os módulos de um programa ou isolar bibliotecas de funções.

ARGUMENTOS
*filename*: nome de um arquivo em Lua

RETORNO
Os valores retornados pelo próprio módulo ou *nil* em caso de erro.

```
EXEMPLO:
if (not dofile("teste.lua")) then
   print("Houve um erro ao executar teste.lua")
end
```

# dostring

dostring (string)

Recebe um valor do tipo string e o executa como um código Lua.

Útil para montar instruções a partir de valores de variáveis.

ARGUMENTOS
*string*: string a ser executada.

RETORNO
Os valores retornados pelo próprio código ou *nil* em caso de erro.

```
EXEMPLO:
local teste_1 = "aaaa",
 local teste_2 = "bbbb"
 local i = 1
 while (i < = 2) do
    dostring("print(teste_" .. i .. ")")
    -- executa print(teste_1) e print(teste_2)
   i = i + 1
 end
```

# next

next (table, index)

Retorna o próximo elemento da tabela *table* com relação ao índice index, ou retorna o primeiro elemento da tabela caso index seja *nil*.

Útil para percorrer todos os elementos de uma tabela.

ARGUMENTOS
*table:* tabela a ser percorrida.
*index*: indice da tabela, de qualquer tipo Lua, como referência para o próximo, *nil* para começar.

RETORNO
Retorna o índice na tabela correspondente ao próximo elemento e o valor do próximo elemento. Se não existir um próximo elemento (final da tabela), retorna *nil*.

```
EXEMPLO:
campo, valor = next (t, nil)
 while (campo) do
    print(campo .. " = " .. valor)
    campo, valor = next (t, campo)
  end
-- percorre e imprime todos os índices e valores da tabela t
```

# nextvar

nextvar( name )

Permite enumerar todas as variáveis globais cujo valor seja diferente de *nil*. Retorna a próxima variável global tendo a variável global name como referência, ou retorna a primeira variável global caso name seja *nil*.

ARGUMENTOS
*name:* nome da variável a ser pesquisada

*EXEMPLO:*
*O trecho de código abaixo imprime os nomes e valores de todas as variáveis globais:*

```
nome,valor = nextvar(nil)                -- captura primeira variável global
while nome do                        -- enquanto existir variável
  print(nome, "=", valor)                -- imprime valores
        nome,valor = nextvar(nome)  -- captura próxima variável global
end
```

# type

type( value )

Recebe como parâmetro uma expressão e informa o seu tipo.

ARGUMENTOS
*value:* expressão a ser pesquisada

RETORNO
Retorna uma string que descreve o tipo do valor resultante: "*nil*", "number", "string", "table", "function", ou "userdata".

*EXEMPLO:*
*O comando abaixo:*

```
    t = {}
    print(type(2.4), type("Alo"), type(t), type(t[1]), type(print))
```

*tem como saída:*

```
    number
string
table
nil
function
```

# tonumber

tonumber( e [,base] )

Esta função recebe um argumento e tenta convertê-lo para um valor numérico.

ARGUMENTOS
*e:* expressão a ser transformada em valor numérico

RETORNO
Se o argumento já for um valor numérico ou se for uma string para a qual é possível fazer a conversão, a função retorna o valor numérico correspondente. Se o conversão não for possível, retorna-se *nil*.

*EXEMPLO:*
*O comando:*

```
    print(tonumber("34.56 "), tonumber("3.2 X"), tonumber(2))
```

*imprime os valores:*

```
    34.56
    nil
    2
```

# tostring

tostring( e )

Recebe um argumento e o converte para uma string.

ARGUMENTOS
*e*: expressão a ser transformada em string.

RETORNO
Retorna a string correspondente.

*EXEMPLO:*
*O comando:*
*print(tostring(3.2), tostring({10,20,30}), tostring(print))*
*imprime os valores:*
*3.2*
*table: 0x324a43*
*function: 0x63ed21*

# print

print( expr1, expr2, ... )

Recebe uma lista de expressões e imprime seus resultados no dispositivo de saída padrão.

ARGUMENTOS
*expr1*, *expr2*, ...: expressões a serem impressas

*EXEMPLO:*
*print( 2*3+2 )*
*print( "valor = ", valor )*

OBSERVAÇÕES
Esta função não permite saídas formatadas, mas é de grande utilidade para consulta de valores, impressão de mensagens de erro e para teste e depuração de programas.
Para formatação, use format.

# assert

assert( value [,message] )

Certifica que a o valor value é diferente de *nil*. Gera um erro Lua com a mensagem "assertion failed" seguido possivelmente de *message* se *value* for igual a *nil*.
ARGUMENTOS
*value*: valor a ser testado

*EXEMPLO:*
*assert( readfrom(FILE),"cannot open" .. FILE )*

# getn

getn( table )

Retorna o "tamanho" de uma tabela quando vista como uma lista. Se a tabela tem um campo n com um valor numérico, este é o "tamanho" da tabela. Caso contrário, o "tamanho" é o maior índice numérico da tabela com um valor diferente de *nil*.

ARGUMENTOS
*table*: tabela da qual se quer avaliar o tamanho

```
EXEMPLO:
a = {10, 20, 30, 40, 50; campo1 = "a", campo2 = "b"}
print(getn(a))    -- imprime 5
b = {10, 20, 30, 40, 50; campo1 = "a", campo2 = "b", n = 8}
print(getn(b))    -- imprime 8
n = nil
b.n = getn(b)
print(getn(b))    -- imprime 5
```

# foreach

foreach( table, func )

Executa a função *func* sobre todos os elementos da tabela *table*. Para cada elemento, à função *func* são passados o índice e o valor respectivos como argumentos. Se a função retorna algum valor diferente de *nil*, o laço é quebrado e o este valor é retornado como o valor final do *foreach*.

ARGUMENTOS
*table*: tabela a ser percorrida.
*func*: função a ser executada sobre os elementos da tabela table.

```
EXEMPLO:
a = {10, 20, 30, 40, 50; campo1 = "a", campo2 = "b"}
foreach(a, function(i, v)
   print("Índice: ", i, " Valor: ", v)
end)
-- percorre a tabela a e imprime todos os pares índice-valor
```

# foreachi

foreach( table, func )

Executa a função *func* sobre todos os elementos da tabela *table* cujo índice é numérico, em ordem seqüêncial. Para cada elemento, à função *func* são passados o índice e o valor respectivos como argumentos. Se a função retorna algum valor diferente de *nil*, o laço é quebrado e o este valor é retornado como o valor final do *foreachi*.

ARGUMENTOS
*table*: tabela a ser percorrida.
*func*: função a ser executada sobre os elementos da tabela *table*.

```
EXEMPLO
a = {10, 20, 30, 40, 50; campo1 = "a", campo2 = "b"}
foreachi(a, function(i, v)
   print("Índice: ", i, " Valor: ", v)
end)
-- percorre a tabela a e imprime os pares índice-valor de índices numéricos
-- (no caso, a[1] a a[5])
```

# foreachvar

foreach( func )

Executa a função *func* sobre todos as variáveis globais. Para cada variável, à função *func* são passados o nome e o valor respectivos da variável como argumentos. Se a função retorna algum valor diferente de *nil*, o laço é quebrado e o este valor é retornado como o valor final do *foreachi*.

ARGUMENTOS
*func*: função a ser executada sobre as variáveis globais.

```
EXEMPLO
a =10; b = 20
foreachvar(function(n, v)
   print("Nome: ", n, " Valor: ", v)
end)
-- percorre a tabela a e imprime os pares nome-valor das globais.
-- No caso, a saída será:
-- Nome: a  Valor: 10
-- Nome: b  Valor: 20
```

# tinsert

tinsert ( table [, pos], value )

Insere o elemento *value* na tabela *table*, na posição de índice numérico *pos*, empurrando os outros elementos em índices numéricos superiores para frente. Caso *pos* não seja fornecido, insere ao final da tabela. Esta função incrementa o campo *n* da tabela em um.

ARGUMENTOS
*table*: tabela na qual o valor será inserido.
*pos* (opcional): posição na tabela (índice numérico) onde o valor será inserido. Caso seja *nil* ou não fornecido, insere no final da tabela.
*value*: valor a ser inserido.

```
EXEMPLO
a = {10, 20, 40, 50}
tinsert( a, 3, 30}
-- a tabela fica a = {10, 20, 30, 40, 50}
tinsert(a, 60)
-- a tabela fica a = {10, 20, 30, 40, 50, 60}
```

# tremove

tremove ( table [, pos] )

Remove da tabela *table* o elemento da posição de índice numérico *pos*. Caso *pos* não seja fornecido, remove ao final da tabela. Esta função decrementa o campo *n* da tabela em um.

ARGUMENTOS
*table*: tabela na qual o valor será removido.
*pos* (opcional): posição na tabela (índice numérico) de onde o valor será removido. Caso seja *nil* ou não fornecido, remove do final da tabela.
*value*: valor a ser inserido.

*EXEMPLO*
*a = {10, 20, 30, 40, 50}*
*tremove( a, 3 }*
*-- a tabela fica a = {10, 20, 40, 50}*
*tremove(a)*
*-- a tabela fica a = {10, 20, 40}*

# sort

sort ( table [, comp] )

Ordena os elementos de índice numérico da tabela *table* em uma dada ordem. Se *comp* é fornecido, deve ser uma função que recebe dois parâmetros e retorna verdadeiro quando o primeiro parâmetros é "menor" que o segundo (de forma que nenhum comp(table[i + 1], table[i]) será verdadeiro após a ordenação). Caso *comp* não seja dado, é utilizado o operador <.

ARGUMENTOS
*table*: tabela a ser ordenada.
*comp* (opcional): função de ordenação.

*EXEMPLO:*
*a = {"maçã", "abacaxi", "AMORA", "Morango"}*
*caseInsensSort = function(a1, a2)*
*  return (strlower(a1) < strlower(a2))*
*end*
*sort(a, caseInsensSort)*
*// ordena a tabela de strings sem considerar maiúsculas/minúsculas*
*// resultado: a = {"abacaxi", "AMORA", "maçã", "Morango"}*

# Exercícios de Fixação VI

**24) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
line = {0, 10, 100, 120; color = "blue", thickness = 2}
foreachi(line, function(i, v)
        print(i..": "..v)
end)
```

**25) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
line = {0, 10, 100, 120; color = "blue", thickness = 2}
foreach(line, function(i, v)
        print(i..": "..v)
end)
```

**26) O que é impresso na tela após a execução do seguinte programa completo em Lua?**

```
a = {{1, 2, 3}, 4, "5", function () end}
i = 1
while (i <= getn(a)) do
        print(type(a[i]))
        i = i + 1
end
print(type(a))
```

# Biblioteca: Manipulação de Strings

## strlen

strlen( *str* )

Informa o tamanho de uma *string*.

ARGUMENTOS
*str*: string a ser medida

RETORNO
Retorna o número de caracteres presentes na cadeia de caracteres.

*EXEMPLO*
```
print(strlen("Linguagem Lua"))
-- imprime o valor 13.
```

## strsub

strsub( str, I [, j ] )

Cria uma cadeia de caracteres que é uma subcadeia de str, começando na posição i e indo até a posição j (inclusive). Se i ou j tiverem valor negativo, eles são considerados relativos ao final de str. Assim, -1 aponta para o último caracter de str e –2 para o penúltimo, etc. Se j não for especificado, é considerado como sendo equivalente à posição do último caracter. Como particularidade, strsub(str,1,j) retorna um prefixo de str com j caracteres; e strsub(str,i) retorna um sufixo de str começando na posição i.

ARGUMENTOS
*str*: string de onde vai ser extraída a substring.
*i*: posição de início da substring.
*j* (opcional): posição de término da substring.

RETORNO
Retorna a subcadeia.

*EXEMPLO*
```
a = "Linguagem Lua“
print(strsub(a, 11))
-- imprime a string Lua.
```

## strlower

strlower( str )

Cria uma cópia da string passada como parâmetro, onde todas as letras maiúsculas são trocadas pelas minúsculas correspondentes. Os demais caracteres permanecem inalterados.

ARGUMENTOS
*str*: string a ser transformada

RETORNO
Retorna a string resultante.

*EXEMPLO*
*print(strlower("Linguagem Lua 3.2"))*
*-- imprime: linguagem lua 3.2*

# strupper

strupper( str )

Cria uma cópia da string passada como parâmetro, onde todas as letras minúsculas são trocadas pelas maiúsculas correspondentes. Os demais caracteres permanecem inalterados.

ARGUMENTOS
*str*: string a ser transformada.

RETORNO
Retorna a string resultante.

*EXEMPLO*
*print(strlower("Linguagem Lua 3.2"))*
*-- imprime: LINGUAGEM LUA 3.2*

# strrep

strrep( str, n )

Cria uma string que é a concatenação de n cópias da *string str*.

ARGUMENTOS
*str*: string a ser replicada.
*n*: número de vezes que deverá replicar a string.

RETORNO
Retorna a string criada.

*EXEMPLO*
*print(strrep( "abc", 4 ))*
*-- imprime: abcabcabcabc*

# ascii

ascii( str [, i] )

Informa o código ASCII do caracter *str[i]*.

ARGUMENTOS
*str*: string a ser consultada.
*i* (opcional): posição do caracter na string *str*.

RETORNO
Retorna o código correspondente.

*EXEMPLO*
*print(ascii("abc",2))*
*--imprime 42, que é o código ASCII de 'b'*

# format

format( formatstring, exp1, exp2, ... )

Cria uma string com expressões *exp1, exp2 etc*. formatadas de acordo com formatstring. Cada expressão deve ter um código embutido em formatstring, que especificica como a formatação é feita. Os códigos consistem do caracter % seguido de uma letra que denota o tipo da expressão sendo formatada, da mesma forma que na linguagem C.

ARGUMENTOS
*formatstring*: formato a ser usado
*exp1, exp2, ...:* expressões a serem formatadas

RETORNO
Retorna uma string no formato formatstring, com os códigos substituídos pelas expressões correspondentes.

*EXEMPLO*
```
nome = "Carla"
id = 123
print(format( "insert into tabela (nome, id) values ('%s', %d)" , nome, id)
-- imprime: insert into tabela (nome, id) values ('Carla', 123)
a = 123.456
print( format( "%+010.2f", a ) )
-- imprime +000123.46
```

# format – Tipos Aceitos

Tipos aceitos no comando format são os mesmos que na linguagem C.

| | |
|---|---|
| %s | String |
| %q | string com delimitadores, num formato que possa ser lido por Lua |
| %c | caracter |
| %d | inteiro com sinal |
| %i | igual a %d |
| %u | inteiro sem sinal |
| %o | inteiro octal |
| %x | hexadecimal usando letras minúsculas (abcdef) |
| %X | hexadecimal usando letras maiúsculas (ABCDEF) |
| %f | real no formato [-]ddd.ddd |
| %e | real no formato [-]d.ddd e[+/-]ddd |
| %g | real na forma %f ou %e |
| %E | igual a %e, usando o caracter E para o expoente no lugar de e |
| %% | caracter % |

# Expressões Regulares

*Classes de caracteres* são utilizadas para representar um conjunto de caracteres em funções de busca e substituição dentro de strings em Lua.

A tabela abaixo lista as classes de caracteres aceitas em Lua:

| | |
|---|---|
| x | o próprio caracter x, exceto ()%.[*-? |
| %x | (x é um caracter não alfanumérico) representa o caracter x |
| . | qualquer caracter |
| %a | Letras |
| %A | tudo exceto letras |
| %d | Dígitos decimais |
| %D | tudo exceto dígitos |
| %l | letras minúsculas |
| %L | tudo exceto letras minúsculas |
| %s | caracteres brancos (espaços, tabs, quebras de linha) |
| %S | tudo exceto caracteres brancos |
| %u | letras maiúsculas |
| %U | tudo exceto letras maiúsculas |
| %w | caracteres alfanuméricos |
| %W | tudo exceto caracteres alfanuméricos |
| [char-set] | união de todos os caracteres de char-set |
| [^char-set] | complemento de char-set |

Um item de pattern pode ser:
* uma classe de caracteres, que casa com qualquer caracter da classe.
* uma classe de caracteres seguida de *, que casa com 0 ou mais repetições dos caracteres da classe. O casamento é sempre feito com a seqüência mais longa possível.
* uma classe de caracteres seguida de +, que casa com 1 ou mais repetições dos caracteres da classe. O casamento é sempre feito com a seqüência mais longa possível.
* uma classe de caracteres seguida de -, que casa com 0 ou mais repetições dos caracteres da classe. Ao contrário do * e do +, o casamento é sempre feito com a seqüência mais curta possível.
* uma classe de caracteres seguida de ?, que casa com 0 ou 1 ocorrência de um caracter da classe
* %n, para n entre 1 e 9; este item casa com a substring igual à n-ésima string capturada.

Um pattern é uma seqüência de itens de pattern. Patterns compõem as expressões regulares de Lua e serão utilizados nas funções strfind e gsub.

O caracter ^ no início do pattern obriga o casamento no início da string procurada. Um $ no final do padrão obriga o casamento no final da string

Capturas: um pattern pode conter sub-patterns entre parênteses, que descrevem capturas. Quando o casamento é feito, as substrings que casam com as capturas são guardados (capturados) para uso futuro. As capturas são numeradas da esquerda para a direita.

> *Exemplo: no pattern "(a*(.)%w(%s*))", a parte da string que casa com a*(.)%w(%s*) é armazenada como a primeira captura (número 1); o caracter que casou com . é capturado com o número 2 e a parte %s* tem número 3*

## strfind

strfind( str, pattern [, init [, plain ]] )

Procura um padrão dentro de uma cadeia de caracteres. A busca é feita na cadeia str, procurando o padrão *pattern* a partir do caracter *init* (opcional). Se não for especificado o parâmetro *init*, a busca é feita na cadeia toda. Caso o parâmetro *plain* (opcional) seja diferente de *nil*, não é utilizado *pattern matching*, e nesse caso é feita uma busca simples de subcadeia.

ARGUMENTOS
*str*: cadeia de caracteres na qual será feita a busca
*pattern*: padrão procurado
*init* (opcional): índice de str para o início da busca
*plain* (opcional): indica se deve ser usado pattern matching ou não.

RETORNO
São retornados os índices inicial e final da primeira ocorrência do padrão na cadeia *str*. Caso *str* não contenha o padrão, retorna *nil*. Se o padrão for encontrado e este contiver capturas, é retornado um valor a mais para cada captura.

*EXEMPLO*
```
i, f = strfind("Linguagem Lua 3.0", "Lua")
print( i, f )
-- imprime os valores 11 e 13, correspondentes à posição da subcadeia Lua na cadeia pesquisada
O código abaixo utiliza capturas para extrair informações de uma string:
data = "13/4/1997“
i, f, dia, mes, ano = strfind(data, "(%d*)/(%d*)/(%d*)")
print( dia, mes, ano )
-- imprime os valores 13, 4, e 1997
```

# gsub

gsub( str, patt, repl [, n ] )

Retorna uma cópia da string *str*, onde todas as ocorrências do pattern patt são substituídas utilizando-se um comando de substituição especificado por *repl*. Retorna ainda o número total de substituições feitas.

Se *repl* é uma string, então seu valor é utilizado para a substituição. Nessa string, para se referir às capturas feitas no *pattern*, utilize %n, onde *n* é o número da captura (de 1 a 9).

Se *repl* é uma função, então essa função é chamada a cada vez que o pattern for encontrado na string *str*, com todas as substrings capturadas sendo passadas como parâmetro. Se a função retornar uma string, essa string será usada para a substituição. Caso contrário, substitui por uma string vazia.

O parâmetro opcional *n* limita o número de substituições.

ARGUMENTOS
*str*: string onde será feita a busca.
*pattern*: padrão a ser procurado e substituído.
*repl*: string para substituir cada padrão encontrado (ou função).
*table* (opcional): parâmetro a ser passado a repl quando este é uma funçãol
*n* (opcional): número máximo de substituições a serem feitas. Se omitido, faz todas as substituições.

RETORNO
Retorna a string *str* com todas as substituições feitas. Em particular, se o padrão não for encontrado, o resultado será a própria string *str*.

*EXEMPLOS*
```
x = gsub("hello world", "(%w+)", "%1 %1")
-- x="hello hello world world"
x = gsub("hello world", "(%w+)", "%1 %1", 1)
-- x="hello hello world"
x = gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
-- x="world hello Lua from"
```

```
x = gsub("home = $HOME, user = $USER", "%$(%w+)", getenv)
-- x = "home = /home/roberto, user = roberto" (for instance)
x = gsub("4+5 = $return 4+5$", "%$(.-)%$", dostring)
-- x = "4+5 = 9"
local t = {name = "lua", version = "3.2"}
x = gsub("$name - $version", "%$(%w+)", function (v) return %t[v] end)
-- x = "lua - 3.2"
t = {n = 0}
gsub("first second word", "(%w+)", function (w) tinsert(%t, w) end)
-- t = {"first", "second", "word"; n = 3}
```

# Exercícios de Fixação VII

**27) Escreva uma função func que receba uma string e imprima cada caracter desta string na ordem correta e em seguida na ordem inversa.**

**28) Dada uma tabela com elementos em seqüência contendo nomes de arquivos. Qual das linhas a seguir deve ser inserida no programa, no local indicado, para imprimir os nomes de arquivos com a extensão trocada para .lua? Repare que somente é considerado extensão o que vem depois do último ponto, inclusive.**

```
function trocaExtLua(table)
        foreachi(table, function(i, v)
                -- INSIRA O CÓDIGO AQUI
        end)
end

print(gsub(v, ".[.*]",  ".lua"))
print(gsub(v, "%..*",  ".lua"))
print(gsub(v, "%.[^%.]*$", ".lua"))
print(gsub(v, "%.[^%.]*", ".lua"))
```

**29) Escreva uma função trim, utilizando gsub, que remova os espaços no começo e no final de uma string dada.**

**30) Escreva uma função func1, que receba uma string como parâmetro e imprima todos os números, com ou sem casas decimais, contidos nessa string separados por letras, espaços ou quebras. Imprima os números com três casas decimais.**

# Biblioteca: Funções Matemáticas

| | |
|---|---|
| log(value) | logaritmo de value na base e |
| log10(value) | logaritmo de value da base 10 |
| cos(angle) | cosseno de angle (especificado em graus) |
| sin(angle) | seno de angle (especificado em graus) |
| tan (angle) | tangente de angle (especificado em graus) |
| acos(value) | arco cosseno, em graus, de value |
| asin(value) | arco seno, em graus, de value |
| atan(value) | arco tangente, em graus, de value |
| atan2(y,x) | arco tangente, em graus, de y/x |
| deg(angle) | converte angle (especificado em radianos) para graus |
| rad(angle) | converte angle (especificado em graus) para radianos |
| abs(value) | valor absoluto de value |
| sqrt(value) | raiz quadrada de value |
| ceil(value) | inteiro imediatamente inferior a value |
| floor(value) | inteiro imediatamente superior a value |
| mod(value,div) | resto da divisão inteira de value por div |
| min(expr1, expr2, ...) | o menor dos valores |
| max(expr1, expr2, ...) | o maior dos valores |
| random() | número real pseudo-randômico entre 0 e 1 |
| random(n) | número real pseudo-randômico entre 0 e n |
| random(l, u) | número real pseudo-randômico entre l e u |
| randomseed(n) | utiliza n como início para geração da série de pseudo-randômicos |
| value1^value2 | value1 elevado a value2 (exponenciação) |
| PI | global com o valor de PI |

# Biblioteca: Funções de E/S

## readfrom

readfrom( [filename] )

Abre ou fecha um arquivo para leitura. Se o parâmetro for uma string, a função abre o arquivo nomeado *filename*, colocando-o como arquivo de entrada corrente, e retorna uma referência para o arquivo aberto. Se a função for chamada sem parâmetros, o arquivo de entrada corrente é fechado e a entrada padrão é restaurada como arquivo de entrada corrente.

ARGUMENTOS
*filename*: nome do arquivo a ser aberto.

RETORNO
Retorna uma referência para o arquivo aberto (userdata com o FILE* de C). Em caso de erro, retorna nil e uma string descrevendo o erro.

```
EXEMPLO
readfrom( "c:\\txt\\b.txt" )
readfrom()
```

## writeto

writeto( [filename] )

Abre ou fecha um arquivo para escrita. Se o parâmetro for uma string, a função cria um arquivo nomeado filename, colocando-o como arquivo de saída corrente, e retorna uma referência para o arquivo aberto (caso já exista um arquivo com este nome, o seu conteúdo é perdido). Se a função for chamada sem parâmetros, o arquivo de saída corrente é fechado e a saída padrão é definida novamente como arquivo de saída corrente.

ARGUMENTOS
*filename*: nome do arquivo a ser aberto.

RETORNO
Retorna uma referência para o arquivo aberto (userdata com o FILE* de C). Em caso de erro, retorna *nil* e uma string descrevendo o erro.

```
EXEMPLO
if writeto( "a.txt") then
  write( "conteúdo do arquivo" )
  writeto()
end
```

## appendto

appendto( [filename] )

Abre um arquivo para escrita nomeado *filename* e o define como arquivo de saída corrente. Ao contrário da função writeto, caso já exista um arquivo com este nome o seu conteúdo não é perdido; novas escritas são acrescentadas aos dados já existentes. Quando chamada sem parâmetros, tem o mesmo comportamento da função *writeto*: fecha o arquivo de saída corrente e restaura a saída padrão como corrente.

ARGUMENTOS
*filename*: nome do arquivo a ser aberto.

RETORNO
Retorna uma referência para o arquivo aberto (userdata com o FILE* de C). Em caso de erro, retorna nil e uma string descrevendo o erro.

```
EXEMPLO
if appendto( "a.txt") then
  write( "conteúdo do arquivo" )
  appendto()
end
```

# read

read( [readpattern] )

Lê uma string do dispositivo de entrada corrente (arquivo, tela etc) de acordo com o *pattern readpattern*. O arquivo é lido até que o pattern termine ou falhe. A função retorna os caracteres lidos, mesmo que o *pattern* não tenha sido completamente lido. Quando chamada sem parâmetros, é usado o pattern [^\n]*{\n}, que lê uma linha do arquivo. A descrição do padrão é a mesma utilizada pela função *strfind.*

O padrão pode conter sub-padrões entre chaves, que definem skips. Os skips são lidos do arquivo mas não são incluídos na string resultante.

O comportamento dos padrões usados nesta função é diferente do *pattern matching* regular, onde um * expande para o maior comprimento tal que o resto do padrão não falhe. Nesta função, uma classe de caracteres seguida de * lê o arquivo até encontrar um caracter que não pertence à classe, ou até final do arquivo.

Existem alguns padrões pré-definidos:
*n" lê um número
*l" retorna a próxima linha pulando ofim da linha), ou nil no final do arquivo. Este é o pattern default, equivalente ao pattern "[^\n]*{\n}".
*a" lê o arquivo inteiro. Equivalente ao pattern ".*".
*w" lê a próxima palavra (seqüência máxima de caracteres não-espaço), pulando espaços caso necessário, ou nil no final do arquivo. Equivalente ao pattern "{%s*}%S+".

ARGUMENTOS
*readpattern*: padrão a ser lido.

RETORNO
Retorna uma string com o conteúdo do dispositivo de entrada que casou com o *pattern readpattern* (mesmo parcialmente) ou *nil* se os dados do arquivo não casaram com nada do *pattern* (ou se o arquivo chegou ao fim).

```
EXEMPLO
data = {}
i = 0
readfrom( "datas.txt" )
repeat
 i = i + 1
          data[i] = read( "%d/%d/%d{\n}" )
until not data[i]
readfrom()
-- lê todas as datas contidas no arquivo datas.txt, colocando-as na tabela data.
```

# write

write( *value1*, *value2*, ...)

Recebe uma lista de valores e os escreve no dispositivo de saída corrente (arquivo, tela etc). Os valores devem ser números ou strings. Diferentemente da função print, não é gerada uma quebra de linha após cada valor escrito.

ARGUMENTOS
*value1, value2, …* : valores a serem escritos.

RETORNO
Nenhum.

*EXEMPLO*
```
write( “a = ”, a )
```

# remove

remove*( filename )*

Apaga o arquivo nomeado *filename*.

ARGUMENTOS
*filename*: path físico do arquivo a ser apagado.

RETORNO
Se a função não conseguir apagar o arquivo, ela retorna *nil* e uma string descrevendo o erro.

*EXEMPLO*
```
a, error = remove( "c:\doc\arq.txt" )
if not a then
   print( error )
end
-- tenta apagar o arquivo c:\doc\arq.txt.
-- Em caso de erro, é impressa uma mensagem explicando a causa.
```

# rename

rename( name1, name2 )

Renomeia o arquivo nomeado *name1* para *name2*.

ARGUMENTOS
*name1*: nome do arquivo a ser renomeado.
*name2*: novo nome do arquivo.

RETORNO
Se a função não conseguir renomear o arquivo, ela retorna *nil* e uma string descrevendo o erro.

*EXEMPLO*
```
a, error = rename( "arq.txt", "arquivo.txt" )
if not a then
   print( error )
end
-- tenta renomear o arquivo arq.txt para arquivo.txt.
-- Em caso de erro, é impressa uma mensagem explicando a causa.
```

# tmpname

tmpname( )

Obtém um nome de arquivo que pode ser usado para criar um arquivo temporário. O arquivo precisa ser explicitamente apagado quando não for mais necessário.

ARGUMENTOS
Nenhum.

RETORNO
Retorna uma string com o nome do arquivo.

*EXEMPLO*
*filename = tmpname()*
*writeto( filename )*
*--cria um arquivo temporário para escrita.*

# date

date( [format] )

Consulta a data atual, retornando-a formatada de acordo com o parâmetro *format* (opcional). O formato é uma string com códigos na forma %c, que especificam os componentes da data (mês, dia, ano, hora etc.). A funcão retorna a string *format* com os códigos substituídos pelos valores correspondentes à data no momento da chamada. O formato usado quando o parâmetro não é especificado é dependente do sistema.

ARGUMENTOS
*format* (opcional): descrição de como a data deve ser formatada.

RETORNO
Uma string com a data atual.

*EXEMPLO*
*print( date( "hoje é dia %d do mês %B" ) )*
*-- imprime a string: hoje é dia 14 do mês Agosto*
*-- (considerando a execução no dia 14/8 em um sistema que utiliza a língua portuguesa)*

# date - Formatos

A tabela abaixo lista os códigos aceitos no formato:

| | |
|---|---|
| %a | nome do dia da semana, abreviado |
| %A | nome do dia da semana, completo |
| %b | nome do mês, abreviado |
| %B | nome do mês, completo |
| %c | data e hora, no formato usado pelo sistema |
| %d | dia do mês, de 01 a 31 |
| %H | hora, de 00 a 23 |
| %I | hora, de 01 a 12 |
| %j | dia do ano, de 001 a 366 |
| %m | número do mês, de 01 a 12 |
| %M | minuto, de 00 a 59 |
| %P | indicação am/pm |
| %S | segundo, de 00 a 60 |

%U semana do ano
(considerando domingo como o primeiro dia da semana), de 00 a 53
%w número do dia da semana, de 0 a 6 (0 é domingo)
%W semana do ano
(considerando segunda-feira como o primeiro dia da semana), de 00 a 53
%x data no formato usado pelo sistema
%X hora no formato usado pelo sistema
%y ano sem o século, de 00 a 99
%Y ano (completo) com o século
%z time zone
%% caracter %

# exit

exit( [code] )

Termina a execução do programa, retornando *code* a quem o executou. Caso o parâmetro *code* não seja especificado, é usado o valor 1.

ARGUMENTOS
*code* (opcional): o código a ser retornado.

RETORNO
Esta função não retorna.

*EXEMPLO*
*exit(2)*
*-- termina a execução do programa retornando 2 a quem o executou*

# getenv

getenv( varname )

Consulta o valor da variável de ambiente do sistema nomeada *varname*.

ARGUMENTOS
*varname*: nome da variável de ambiente a ser consultada.

RETORNO
Retorna uma string com o valor da variável *varname*, ou *nil* se esta não estiver definida.

*EXEMPLO*
*print( getenv( "REMOTE_HOST" ) )*
*-- imprime o valor da variável de ambiente REMOTE_HOST*

# execute

execute( command )

Executa o comando command no sistema operacional.

ARGUMENTOS
*command*: comando a ser executado.

RETORNO
O valor de retorno desta função é dependente do sistema operacional. Normalmente é um valor retornado pelo comando executado.

*EXEMPLO*
*execute( "mkdir c:\data" )*
*-- executa um comando no sistema operacional*
*-- para criar o diretório c:\data.*

# Exemplo Completo

```
write("\nComandos:\n\n")
write("R: renomear arquivo\n")
write("D: remover arquivo\n")
write("X: executar comando DOS\n")
write("A: mostrar valor de variavel de ambiente\n")
write("E: sair so programa\n\n")

while(1) do
   write("? ")
   local cmd = strupper(read())
   if (cmd == "R") then
      write("Nome atual do arquivo: ")
      local oldname = read()
      write("Novo nome para o arquivo: ")
      local newname = read()
      local ret, error = rename(oldname, newname)
      if (not ret) then
         write(error.."\n")
      else
         write("Arquivo renomeado!\n")
      end
   elseif (cmd == "D") then
      write("Nome do arquivo a apagar: ")
      local filename = read()
      write("Apagar o arquivo '"..filename.."'? (S/N)")
      local answer = strupper(read())

 if (answer == "S") then
         local ret, error = remove(filename)
         if (not ret) then
            write(error.."\n")
         else
            write("Arquivo apagado!\n")
         end
      end
   elseif (cmd == "X") then
      write("Comando a executar: ")
      local command = read()
      local ret = execute(command)
      write("\nO comando retornou '"..ret.."'.\n")
   elseif (cmd == "A") then
      write("Nome da variavel de ambiente: ")
      local envVar = read()
      local valEnvVar = getenv(envVar)
      write("Valor de '"..envVar.."': ")
      if (valEnvVar) then
         write(valEnvVar.."\n")
      else
         write("Variavel inexistente.\n")
      end
   elseif (cmd == "E") then
      exit()
   else
      write("Comando nao reconhecido.\n")
   end
end
```

# Exercícios de Fixação VIII

**31) Escreva uma função *latex2XML* em Lua que receba dois nomes de arquivos como entrada. Ela deverá ler o primeiro arquivo linha a linha, convertê-lo do formato *comando*{*texto*} para o formato <*comando*>*texto*</*comando*>, e gravar o resultado no segundo arquivo.**

# Respostas dos Exercícios de Fixação

**1)** 20
12
12

**2)** 20
nil
4
10

**3)** Isto é um 'teste'!!!
Isto é um
"teste"!!!
Isto é um
"teste"!!!

**4)** letra e

**5)** 10
20
30

**6)** teste
0
nil

**7)** 5
nil
4
5

**8)** 30
20

**9)** nil
1
1
nil

**10)** 01
1
Repita 10 vezes.
1

**11)** 5
10

**12)** Não incrementa o i. Portanto o laço (loop) é infinito.

**13)** 0
1
2
3
4

**14)** 0
1
2
3
4

**15)** letra c

**16)** t = {{11, 12, 13 ,14}, {21, 22, 24, 24}, {31, 32, 33, 34}}

print(t[3][1], t[3][2], t[3][3], t[3][4]}

**17)** t1 = {{nome = "João da Silva", endereco = "Rua A, casa B", telefone = "2222-2222"},
{nome = "Maria Joaquina", endereco = "Rua C, casa D", telefone = "3333-3333"},
{nome = "Pedro Joaquim", endereco = "Rua E, casa F", telefone = "4444-4444"}}

t2 = {}
t2[1].nome = "João da Silva"
t2[1].endereco = "Rua A, casa B"
t2[1].telefone = "2222-2222"

t2[2].nome = "Maria Joaquina"
t2[2].endereco = "Rua C, casa D"
t2[2].telefone = "3333-3333"

t2[3].nome = "Pedro Joaquim"
t2[3].endereco = "Rua E, casa F"
t2[3].telefone = "4444-4444"

**18)**
200
20

**19)** nil
2

**20)** 10

**21)** 11
21
31

**22)** function eq2(x, c, b, a)
a = a or 0
b = b or 0
c = c or 0
return (a * x^2 + b * x + c)
end

**23)** function fact (n)
if (n == 0) then
return 1
else
return n * fact(n - 1)
end
end

**24)** 1: 0
2: 10
3: 100
4: 120

**25)** 1: 0
2: 10
3: 100
4: 120
color: blue
thickness: 2

**26)** table
number
string
function
table

**27)** function func(str)
local i = 1
while(i <= strlen(str)) do
print(strsub(str, i, i))
i = i + 1
end
i = 1
while(i <= strlen(str)) do
print(strsub(str, -i, -i))
i = i + 1
end
end

**28)** letra c

**29)** function trim(str)
str = gsub(str, "^%s*", "")
str = gsub(str, "%s*$", "")
return str
end

ou

function trim (str)
return (gsub(str, "^%s*(.-)%s*$", "%1"))
end

30) function func1(str)
gsub(str, "([%d%.]+)", function (v)
print(format("%.3f", v))
end)
end

**31)** function latex2XML(fileIn, fileOut)
readfrom(fileIn)
writeto(fileOut)
local line
repeat
line = read()
if (line) then
line = gsub(line, "\\(%a+){(.*)}",
"<%1><%2></%1>")
write(line.."\n")
end
until not line
writeto()
readfrom()
end

# Bibliografia

1) Noções de Lua 3.1 - Noções básicas da linguagem de programação Lua - **Roberto de Beauclair Seixas**

2) Reference Manual of the Programming Language Lua 3.2 - **Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes**

3) A Linguagem LUA (PowerPoint) - **Alexandra Barros**

4) Lua and the Web (PowerPoint) - **Roberto Ierusalimsch**