Noções de Lua 3.1

Noções básicas da linguagem de programação Lua

ÍNDICE

1. Copyright

O texto aqui incluído foi extraído em parte dos documentos “Programando em Lua – Teoria e Prática (versão 2.1)” e “A Linguagem de Extensão Lua” escrito por Waldemar Celes Filho, Luiz Henrique de Figueiredo e Roberto Ierusalimschy. Atualizações para a versão 3.1 de Lua foram feitas por Roberto de Beauclair Seixas, baseado no texto original de Anna Magdalena Hester. Este documento está disponível em:

ftp://ftp.tecgraf.puc-rio.br/pub/lua/nocoes-3.1.pdf
http://www.tecgraf.puc-rio.br/lua/ftp/nocoes-3.1.pdf

Lua é uma linguagem de programação de distribuição aberta, gratuita. A implementação descrita aqui está disponível em:

```
ftp://ftp.tecgraf.puc-rio.br/pub/lua/lua-3.1.tar.gz
http://www.tecgraf.puc-rio.br/lua/
```

Copyright de Lua:

2. Visão Geral

Lua é uma linguagem de extensão projetada para ser usada como linguagem de configuração, acoplada a um programa hospedeiro (escrito na linguagem de programação C). Aplicações em geral podem acoplar códigos em Lua, permitindo prototipagem rápida e acesso programável pelo usuário à tecnologia implementada pela aplicação.

No entanto é bastante comum o uso de Lua como um interpretador *stand-alone*. Nesse enfoque o código Lua é a linha de execução principal, o programador escreve todo o código da aplicação em Lua e não precisa ter conhecimento da API em C. O ambiente de programação é formado pelas funções pré-definidas de Lua, pelas bibliotecas padrão da linguagem e por eventuais pacotes de extensão adicionados.

CGILua é essencialmente um interpretador Lua com uma série de pacotes de extensão, de forma que seu ambiente de programação é capaz de prover diversas funcionalidades úteis para o desenvolvimento de *scripts* CGI e páginas HTML dinâmicas.

Lua provê as construções fundamentais para definição de funções e controle de fluxo (*if-elseif-else-end*, *while-do-end*, *repeat-until*), um poderoso tipo para descrição de estruturas de dados, a *tabela*, e facilidades para manipulação de *strings* – em especial, os recursos de *pattern matching*.

Todos os mecanismos e recursos citados acima são facilmente acessados através de uma sintaxe simples, semelhante à de Pascal. Lua garante ainda uma programação com alto nível de abstração, já que o programador não precisa se preocupar com detalhes como declaração de variáveis e gerenciamento de memória. Com estruturas internas dinâmicas e coleta automática de lixo, o programador usa livremente as diversas opções de estruturação de dados, e deixa a cargo da própria linguagem as tarefas de re-alocação e liberação de memória.

Este texto apresenta, passo a passo, as principais características da linguagem de programação Lua. A abordagem é concentrada na utilização *stand-alone* da linguagem, ignorando a existência de um programa *hospedeiro* escrito em C. O texto pretende ter um caráter didático; alguns recursos da linguagem são omitidos, em especial o mecanismo de *tags methods*, os *contrutores de tipos*, a API C e a interface para *debug*. A definição oficial da linguagem está descrita no seu manual de referência, disponível *on-line* em `http://www.tecgraf.puc-rio.br/lua/manual`.

Nas seções subseqüentes, são apresentadas as diversas características da linguagem Lua, através de uma discussão detalhada e ilustrada com exemplos de programação.

3. Ambiente de Programação

Todos os comandos e construções de Lua são executados em um *único ambiente global*. Este ambiente, que guarda todas as variáveis globais e definições de funções, é automaticamente inicializado quando o interpretador é ativado e persiste enquanto o interpretador estiver em execução.

O ambiente global de Lua pode ser manipulado por códigos escritos em Lua ou por bibliotecas C que utilizem funções da interface C-Lua. Códigos Lua são compostos por comandos globais e definições de funções. Um módulo Lua é um arquivo ou uma cadeia de caracteres contendo códigos Lua. Quando se carrega um módulo Lua, os códigos são compilados para uma linguagem de máquina interna, para serem posteriormente executados. Os códigos globais (que não estão dentro de funções) são automaticamente executados ao fim da compilação do módulo. Os códigos de funções são armazenados e executados na chamada das respectivas funções. Em outras palavras, Lua usa uma estratégia híbrida de compilação e interpretação, que evita o custo de desempenho de interpretação textual direta, ao mesmo tempo que mantém a versatilidade de um ambiente interpretativo.

Pode-se carregar diversos módulos inicialmente independentes. Todos, no entanto, são carregados dentro do único ambiente global de Lua. Portanto, a ordem com que se carrega diversos módulos é importante, tendo em vista que os comandos globais de cada módulo são executados imediatamente após sua compilação.

Não existe uma formatação rígida para o código Lua, isto é, a linguagem permite comandos em qualquer coluna do texto, podendo ocupar diversas linhas. Os comandos de Lua podem opcionalmente ser separados por ponto e vírgula (`;`). Comentários iniciam-se com traços duplos (`--`) e persistem até o final da linha.

4. Variáveis e tipos

As variáveis globais[1] de Lua não precisam ser declaradas. É válido, por exemplo, escrever o comando

```
a = 2.3
```

sem necessidade prévia de declarar a variável global `a`. Uma característica fundamental de Lua é que as *variáveis não têm tipo*: os tipos estão associados aos dados armazenados na variável. Assim, o código acima armazena em `a` um dado do tipo numérico, com valor `2.3`, enquanto, após o comando

```
a = "Linguagem Lua"
```

a variável `a` passa a armazenar um dado do tipo cadeia de caracteres (*string*).

Os dados em Lua podem ser de sete tipos básicos: *nil*, *number*, *string*, *function*, *cfunction*, *userdata* e *table*, descritos a seguir.

Tipo *nil*

O tipo *nil* representa o *valor indefinido*. Todas as variáveis ainda não inicializadas assumem o valor *nil*. Assim, se o código:

```
a = b
```

for encontrado antes de qualquer atribuição à variável `b`, então esta é assumida como contendo o valor `nil`, o que significa que `a` também passa a armazenar `nil`, independentemente do valor anteriormente armazenado em `a`. A palavra reservada `nil` pode ser usada na programação para expressar o valor do tipo *nil*. Com isto, pode-se escrever:

```
a = nil
```

que atribui o valor `nil` à variável `a` (a partir deste ponto, é como se a variável `a` ainda não tivesse sido atribuída).

Pode-se testar se uma variável foi inicializada comparando o seu valor com `nil`:

[1] Lua também permite a definição de variáveis locais (veja pág. 16).

```
a == nil
```

Tipo *number*

O tipo *number* representa *valores numéricos*. Lua não faz distinção entre valores numéricos com valores inteiros e reais. Todos os valores numéricos são tratados como sendo do tipo *number*. Assim, o código

```
a = 4
b = 4.0
c = 0.4e1
d = 40e-1
```

armazena o valor numérico *quatro* nas variáveis a, b, c e d.

Tipo *string*

O tipo *string* representa *cadeia de caracteres*. Uma cadeia de caracateres em Lua é definida por uma seqüência de caracteres delimitadas por aspas simples (' ') ou duplas (" "). A seqüência de caracteres deve estar numa mesma linha de código. Dentro de um *string*, são interpretadas as seguintes seqüências de *escape*:

\n	*new line*
\t	*tab*
\r	*carriage return*
\v	*vertical tab*
\f	*form feed*
\xxx	*caracter com código decimal* xxx
\a	*bell*
\b	*backspace*
\"	*aspas duplas (“)*
\'	*aspas simples (‘)*
\\	*barra invertida (\)*

Por simplicidade, quando a cadeia de caracteres é delimitada por aspas duplas, pode-se usar aspas simples no seu conteúdo, sem necessidade de tratá-las como seqüências de *escape*. Entretanto, para reproduzir na cadeia de caracteres as aspas usadas como delimitadoras, é necessário usar os caracteres de *escape*. Assim, são válidas e equivalentes as seguintes atribuições:

```
s = "Olho d'agua"
s = 'Olho d\'agua'
```

Colchetes duplos ([[ ]]) também podem ser utilizados como delimitadores de *strings*. Diferentemente dos delimitadores aspas simples e aspas duplas, esse delimitador não interpreta sequências de escape e tolera que a *string* tenha “quebras de linhas” e pode conter *strings* com delimitadores aninhados. Exemplo:

```
s = [[Esse é um texto
que atravessa
mais de uma linha e contém uma
string aninhada: [[aninhada]] no final !]]
```

Tipo *function*

Funções em Lua são consideradas valores de primeira classe. Isto significa que funções podem ser armazenadas em variáveis, passadas como parâmetros para outras funções, ou retornadas como resultados. A definição de uma função equivale a atribuir a uma variável global o valor do código que executa a função (seção 7). Esta variável global passa a armazenar um dado do tipo *function*. Assim, adiantando um pouco a sintaxe de definição de funções, o trecho ilustrativo de código abaixo armazena na variável `func1` um valor do tipo *function*:

```
function func1 (...)
 ...
end
```

que pode posteriormente ser executada através de uma chamada de função:

```
func1(...)
```

Tipo *userdata*

O tipo *userdata* permite armazenar numa variável de Lua um ponteiro qualquer de C. Este tipo corresponde ao tipo `void*` de C e só pode ser atribuído ou comparado para igualdade a outro valor de mesmo tipo em Lua. Este tipo é muito útil para programadores de aplicação que fazem a ligação Lua-C, mas não é manipulado por usuários que programam somente em Lua, pois não é possível criar dados deste tipo diretamente em Lua.

Tipo *table*

O tipo *table* (tabela) é o tipo mais expressivo da linguagem. Este tipo implementa os chamados vetores associativos, que permitem indexação por valores de qualquer outro tipo, com exceção do tipo *nil*. As tabelas em Lua permitem a construção de vetores convencionais, listas e *records* numa mesma estrutura. Tabelas devem ser explicitamente criadas. Adiantando a sintaxe de criação de uma tabela, o código

```
a = { }
```

cria uma tabela vazia e armazena em `a` este valor. Novos campos podem ser adicionados posteriormente a esta tabela. Tabelas são detalhadamente discutidas na seção 8.

5. Atribuição e operadores

Conforme mencionado, as variáveis em Lua não têm tipos; elas apenas armazenam valores de diferentes tipos. Diz-se que as variáveis são *tipadas* dinamicamente. Quando se atribui um valor a uma variável, é armazenado na variável um valor de um determinado tipo. Se é feita uma outra atribuição à mesma variável, ela passa a armazenar o valor correspondente à segunda atribuição. Portanto, antes da segunda atribuição, a variável armazenava um valor de um determinado tipo; após a segunda atribuição, a variável passa a armazenar um valor de um tipo possivelmente diferente.

Quando se efetua operações sobre os valores armazenados nas variáveis, a linguagem verifica, em tempo de execução, a validade dos tipos armazenados para a operação em questão. Por exemplo, ao tentar efetuar uma operação de soma sobre uma variável que armazena o valor de uma função, a linguagem reporta um erro de execução[2].

Atribuição simples e múltipla

Os exemplos das seções anteriores ilustraram a sintaxe de atribuição simples. Lua também permite atribuição múltipla. É possível atribuir diversas variáveis em um mesmo comando. Por exemplo, o código

```
s, v = "Linguagem Lua", 2
```

atribui a cadeia de caracteres `Linguagem Lua` à variável `s` e o valor numérico `2` à variável `v`. Quando o número de variáveis listadas à esquerda do sinal de igualdade é diferente do número de resultados à direita da igualdade, a linguagem automaticamente ajusta as listas: ou preenchendo as últimas variáveis da esquerda com valores `nil`s ou descartando os últimos resultados da direita. Assim, o código

```
a, b = 2
c, d = 2, 4, 6
```

atribui `2` à variável `a`, `nil` à variável `b`, `2` à variável `c`, `4` à variável `d`, e despreza o valor `6`.

A possibilidade de atribuição múltipla permite a troca de valores armazenados em duas variáveis com um único comando. Portanto

```
a, b = b, a
```

faz com que `a` receba o valor anteriormente armazenado por `b` e que `b` receba o valor anteriormente armazenado por `a`, sem necessidade de variáveis temporárias.

[2]Este comportamento pode ser alterado usando *tag methods* (veja o Manual de Referência de Lua).

Operadores

A linguagem Lua provê os operadores aritméticos, relacionais e lógicos usuais. Existe, ainda, um operador para concatenação de cadeias de caracteres. Lua possui também algumas regras de conversão automática de tipos.

Operadores aritméticos

Lua dispõe dos operadores aritméticos usuais: os operadores binários são `+` (adição), `-` (subtração), `*` (multiplicação) e `/` (divisão); e o operador unário `-`. É permitido o uso de parênteses para alterar a precedência dos operadores. Portanto, são válidas as expressões abaixo:

```
a = 2 * 3.4
b = (a + 1) / 3.4
c = 2 * (-3)
```

Os operadores aritméticos só podem ser aplicados sobre valores do tipo *number* (ou *strings* que possam ser convertidas para números; veja tópico "Coerção" nessa mesma seção).

O operador binário `^`, com precedência superior aos demais, pode ser implementado através do mecanismo de *tag methods* (veja o Manual de Referência de Lua). A biblioteca de funções matemáticas distribuída com a linguagem implementa a operação de exponenciação usual para este operador.

Operadores relacionais

Os operadores relacionais disponíveis em Lua são:

`<`	*menor que*
`>`	*maior que*
`<=`	*menor que ou igual a*
`>=`	*maior que ou igual a*
`==`	*igual a*
`~=`	*diferente de*

Os operadores relacionais retornam o valor `nil` quando o resultado é falso e o valor `1` quando o resultado é verdadeiro. Assim, as atribuições

```
a = 4 < 3
b = 4 > 3
```

armazenam em `a` o valor `nil` e em `b` o valor `1`.

Os operadores >, <, >=, <= só são aplicáveis em dados do tipo *number* ou *string*, e têm a interpretação usual. Este comportamento pode ser alterado usando *tag methods* (consultar o Manual de Referência de Lua para maiores informações).

O operador de igualdade (==) primeiro compara se os tipos dos dados comparados são iguais. Caso não sejam, retorna-se `nil`. Se os valores tiverem o mesmo tipo, compara-se a igualdade entre os valores. A igualdade de números e cadeias de caracteres é feita da forma usual. Tabelas, funções e *userdata* são comparados por referência, isto é, duas tabelas são consideradas iguais somente se as duas forem a *mesma* tabela, e não se as duas contêm armazenados os mesmos dados.

O operador de não-igualdade (~=) é a negação do operador de igualdade.

Operadores lógicos

Os três operadores lógicos disponíveis são:

`and`	*conjunção*
`or`	*disjunção*
`not`	*negação*

Conjunções ou disjunções múltiplas são avaliadas da esquerda para a direita; a avaliação pára assim que a veracidade ou falsidade do resultado for conhecida. Assim, o código

```
a = 23
b = 15
c = a < 20 or b > a
d = a < 20 and b > a
```

resulta no valor `nil` em `c` e `d`, sendo que, para avaliar `d`, não é necessário avaliar `b>a`, pois o resultado já era conhecido tendo em vista que a expressão era conectada com o operador lógico `and` e `a<20` já é falso.

Operador de concatenação

O operador para concatenação de cadeias de caracteres em Lua é representado por dois caracteres ponto (`..`). Aplicado a dois valores do tipo *string*, este operador retorna uma terceira cadeia de caracteres que corresponde à união das duas cadeias originais. Portanto, o código

```
a = "Linguagem"
b = "Lua"
c = a .. " " .. b
```

resulta no armazenamento da cadeia de caracteres `Linguagem Lua` na variável `c`, resultante da concatenação da variável `a` com uma cadeia composta por um caractere branco, seguida da variável `b`.

Coerção

Lua provê alguns mecanismos para conversão automática entre tipos. Qualquer operação aritmética aplicada sobre cadeias de caracteres tenta converter a cadeia de caracteres no valor numérico correspondente. Mais especificamente, se os caracteres da *string* representam um valor numérico, então este valor numérico é assumido na operação. Quando a conversão não é possível, a linguagem reporta um erro de execução. Portanto, o código

```
b = "53"
c = 2 + b
```

resulta no armazenamento do valor numérico `55` na variável `c`, tendo em vista que a cadeia de caracteres `53` foi convertida para o valor numérico `53` antes da operação (a variável `b` continua armazenando uma variável do tipo *string*).

Por outro lado, quando utilizamos um valor numérico onde espera-se uma cadeia de caracteres (e.g., numa concatenação), o valor numérico é convertido para a cadeia de caracteres correspondente. Se o valor numérico for inteiro, ele é escrito numa cadeia de caracteres sem ponto decimal ou expoente. Se o valor contiver parte fracionária, o resultado pode conter ponto decimal e expoente[3]. Assim, o código

```
print("resultado: " .. 23)
```

tem como saída a cadeia de caracteres

```
resultado: 23.
```

Existe uma função pré-definida, `tonumber`, que faz explicitamente a conversão de uma cadeia de caracteres para a forma numérica, se posssível. Para converter explicitamente um valor numérico numa cadeia de caracteres, existe a função `tostring` (seção 10).

Ordem de precedência dos operadores

Os operadores têm a ordem de precedência usual. Conforme mencionado, é possível o uso de parênteses para mudar a ordem natural de avaliação de uma expressão. A lista abaixo apresenta os operadores, em ordem decrescente de precedência:

```
^
not   -(unário)
*     /
+     -
..
<     >     <=     ~=     ==
and   or
```

[3]Mais precisamente, utiliza-se `tostring`, que por *default* usa a função `sprintf` com `%.16g`.

6. Controle de fluxo e variáveis locais

Lua possui os mecanismos usuais para controle de fluxo. Estes controladores agrupam diversos comandos em blocos que podem ou não ser executados uma ou mais vezes. Lua também permite a declaração de variáveis locais. Estas variáveis deixam de existir ao final do bloco onde foram declaradas.

Tomadas de decisão com *if*

O comando de decisão básico de Lua é o *if*. Sua forma pode ser:

```
if expr then
 bloco
end
```

ou

```
if expr then
 bloco1...
else
 bloco2...
end
```

ou ainda

```
if expr1 then
 bloco1
elseif expr2 then
 bloco2
 ...
elseif expr N then
 bloco N
else
 bloco N+1
end
```

Na primeira forma, o bloco de comandos representado por *bloco* é executado se a expressão *expr* produzir um valor diferente de `nil`. Na segunda forma, *bloco2* será executado se *expr* produzir o valor `nil`. Caso contrário, *bloco1* será executado. A terceira forma ilustra a possibilidade de tomada de decisão entre diversas alternativas usando-se o comando *if-then-elseif-then-...-else-end*. No código ilustrado, *bloco1* é executado se *expr1* produzir um valor verdadeiro (diferente de `nil`), *bloco2* é executado de *expr2* for verdadeiro, ..., e *bloco n+1* é executado se todas as expressões forem falsas (i.e., iguais a `nil`).

Laços iterativos com tomada de decisão no início (*while*)

Lua possui duas alternativas para construção de laços iterativos: *while* e *repeat*. A primeira delas permite que a tomada de decisão (se os comandos do laço devem ou não ser executados) seja feita no início do bloco. Sua forma geral é:

```
while expr do
 bloco
end
```

Isto é, enquanto a expressão *expr* produzir um valor verdadeiro (diferente de `nil`), os comandos do *bloco* são executados.

Para exemplificar, considere o código abaixo que calcula o fatorial de um número (assumido como inteiro positivo) armazenado na variável `n`:

```
f = 1                          valor do fatorial
i = n                          controle do laço
while i > 0 do
 f = f * i
 i = i – 1
end
```

Ao final da execução do código acima, `f` armazena o valor do fatorial de `n` e `i` armazena o valor zero (condição de fim de execução do laço).

Laços iterativos com tomada de decisão no fim (*repeat*)

A construção de um laço com tomada de decisão no fim pode ser feita através do comando *repeat*. Sua forma geral é:

```
repeat
 bloco
until expr
```

Nesta construção, o bloco de comandos é executado *pelo menos uma vez* (pois o teste de decisão só ocorre no final do laço) e é repetido até se alcançar o valor verdadeiro para a expressão. O mesmo exemplo de cálculo de fatorial pode ser escrito:

```
f = 1          valor do fatorial
i = 1          controle do laço
repeat
 f = f * i
 i = i + 1
until i > n
```

Declaração de variáveis locais

Lua permite que se defina explicitamente variáveis de escopo local. A declaração de uma variável local pode ocorrer em *qualquer* lugar dentro de um bloco de comandos, e seu escopo termina quando termina o bloco no qual foi declarada. A declaração de uma variável local com mesmo nome de uma variável global obscurece temporariamente (i.e., dentro do bloco da declaração local) o acesso à variável global. Quando o programador escrever o nome da variável, estará se referindo à variável local.

Variáveis locais podem ser inicializadas na declaração seguindo a sintaxe de atribuições. Para exemplificar, considere o código abaixo:

```
a = 2                 variável global assumindo o valor 2
if a > 0 then
 local b = a          declara-se uma variável local que recebe o valor de a (2)
 a = a + 1            incrementa a variável global a de uma unidade
 local a = b          declara-se uma variável local a que recebe o valor de b
 print(a)             a  refere-se a variável local, logo imprime-se o valor 2
end                   fim do bloco e do escopo de a e b locais

print(a)              a refere-se à variável global, logo imprime-se o valor 3
```

Pode-se ainda declarar e inicializar várias variáveis locais num mesmo comando:

```
local a, b, c = 2, 5+6, -3
```

Neste caso, `a` recebe `2`, `b` recebe `11` e `c` recebe `-3`. Variáveis locais não inicializadas assumem o valor `nil`.

7. Funções

A linguagem Lua trata funções como valores de primeira classe. Isto quer dizer que funções podem ser armazenadas em variáveis, podem ser passadas como parâmetros para outras funções e podem ser retornadas como resultados de funções. Quando definimos uma função em Lua, estamos armazenando na variável global cujo nome corresponde ao nome da função o código "de máquina" interno da função, que pode posteriormente ser executado através de chamadas para a função.

Funções em Lua podem ser definidas em qualquer lugar do ambiente global. A forma básica para definição de funções é

```
function nome ( [lista-de-parâmetros] )
 bloco de comandos
end
```

onde *nome* representa a variável global que armazenará a função. Pode-se fornecer uma lista de parâmetros que são tratados como variáveis locais da função e inicializados pelos valores dos argumentos na chamada da função.

A chamada da função segue a forma básica

```
nome ( [lista-de-argumentos] )
```

isto é, o nome da função é seguido de abre e fecha parênteses, que pode ou não conter uma lista de argumentos. Se *nome* não for uma variável que armazena um valor de função (i.e., do tipo *function*), Lua reporta um erro de execução (TAG METHODS). Se presente, a lista de argumentos é avaliada antes da chamada da função. Posteriormente, os valores dos argumentos são ajustados para a atribuição dos parâmetros (a regra de ajuste é análoga a uma atribuição múltipla onde os parâmetros recebem os valores dos argumentos).

Lua passa parâmetros por valor. Isto quer dizer que quando se altera o valor de um parâmetro dentro de uma função, altera-se apenas o valor da variável local correspondente ao parâmetro. O valor da variável passada como argumento na chamada da função permanece inalterado.

Funções em Lua podem retornar zero, um ou mais valores através do comando `return`. A possibilidade de retorno múltiplo evita a necessidade de passagem de parâmetros por referência. Quando, durante a execução de uma função, encontra-se o comando `return`, a execução da função é terminada e o controle volta para o ponto imediatamente posterior à chamada da função. O comando `return` pode vir seguido de uma lista de expressões; sua forma geral é

```
return [lista-de-expressões]
```

Por razões sintáticas, o comando `return` deve sempre ser o último comando de um bloco de comandos. Deste modo, evita-se a ocorrência de comandos inalcançáveis (tendo em vista que comandos após `return` nunca serão executados). Se `return` não for o último comando do bloco, Lua reporta um erro de sintaxe.

Os valores retornados por uma função são ajustados para a atribuição na linha que faz a chamada. Para exemplificar, supõe-se uma função que incrementa os valores de um ponto cartesiano (`x, y`):

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

Considera-se, agora, que a chamada da função é feita por:

```
a, b = translada(20, 30, 1, 2)
```

Assim, `a` recebe o valor `21` (=`20+1`) e `b` recebe o valor `32` (=`30+2`). Se a chamada fosse

```
a = translada(20, 30, 1, 2)
```

então o segundo valor retornado seria desprezado, e `a` receberia o valor `21`. Por outro lado, a atribuição:

```
a, b, c = translada(20, 30, 1, 2)
```

faria `a= 21`, `b = 32` e `c = nil`.

É importante notar que uma chamada de função que retorna múltiplos valores não pode estar no meio de uma lista de expressões. Por razões de ajustes de valores em atribuições múltiplas, se uma função estiver sendo chamada no meio de uma lista de expressões, só se considera o primeiro valor retornado. Para exemplificar, duas situações sutilmente diferentes são analisadas; considera-se inicialmente a chamada de função abaixo:

```
a, b, c = 10, translada(20, 30, 1, 2)
```

que faz com que `a` receba o valor `10`, `b` o valor `21` e `c` o valor `32`. Neste caso, o funcionamento é o esperado, tendo em vista que a chamada da função é a última expressão numa lista de expressões. No entanto, se o código fosse

```
a, b, c = translada(20, 30, 1, 2), 10
```

então teria-se resultados surpreendentes, já que a chamada da função está no meio de uma lista de expressões. Conforme mencionado, independentemente do número de valores retornados pela função, considera-se apenas o primeiro (no caso, o valor `21`). O valor `32` retornado pela função é perdido durante o ajuste. Assim, teríamos como

resultado o valor `21` armazenado em `a` e o valor `10` armazenado em `b`. A variável `c`, então, receberia o valor `nil`, pois não se tem outra expressão após a constante `10`.

É possível definir funções em Lua que aceitem número variável de argumentos. Para tanto é necessário acrescentar à lista de argumentos da definição a indicação "`...`" (três pontos seguidos). Uma função definida dessa forma não faz ajuste do número de parâmetros em relação ao número de argumentos. Ao invés disso, quando a função é chamada os argumentos extras são colocados em um parâmetro implícito de nome `arg`. Esse parâmetro é sempre inicializado como uma tabela, onde o campo `n` contém o número de argumentos extras e os campos `1`, `2`, `...` os valores dos argumentos, sequencialmente.

Para exemplificar, duas diferentes definições de funções são apresentadas:

```
function f(a, b) end
function g(a, b, ...) end
```

Os seguintes mapeamentos de argumentos a parâmetros seriam efetuados na chamada dessas funções:

CHAMADA	PARÂMETROS
`f(3)`	`a=3, b=nil`
`f(3, 4)`	`a=3, b=4`
`f(3, 4, 5)`	`a=3, b=4`
`g(3)`	`a=3, b=nil, arg={ n=0 }`
`g(3, 4)`	`a=3, b=4, arg={ n=0 }`
`g(3, 4, 5, 8)`	`a=3, b=4, arg={ 5,8; n=2 }`

8. Tabelas (vetores associativos)

Tabelas em Lua são vetores associativos, isto é, vetores que podem ser indexados por valores numéricos, por cadeias de caracteres ou por qualquer valor dos demais tipos da linguagem. Vetores só não podem ser indexados pelo valor `nil`. Esta flexibilidade na indexação de tabelas é a base do mecanismo existente em Lua para descrição de objetos.

Criação de tabelas

Uma tabela deve ser explicitamente criada antes de ser usada. A expressão `{ }` cria uma tabela. Assim, o código

```
t = { }
```

cria uma tabela vazia e a armazena na variável de nome `t`. A partir deste ponto, a variável `t` armazena um valor do tipo *table* e pode então ser indexada. É válido escrever, por exemplo:

```
t[1] = 13
t[45] = 56
```

Observa-se que não há limite para a indexação da tabela. Se necessário, Lua automaticamente redimensiona a tabela. Além disso, também é válido escrever:

```
t["nome"] = t[1] + 2
```

Isto é, a tabela `t` indexada pela valor cadeia de caracateres `nome` recebe o valor armazenado na tabela `t` indexada pelo valor `1` (que é `13`) somado com o valor `2`. Isto é, `t` indexado por `"nome"` vale `15`.

Também seria válido escrever

```
t[t] = 5
```

que significa que a tabela `t` indexada pelo valor de `t` (que é a própria tabela) armazena o valor numérico `5`. (Parece confuso, mas é válido!)

Quando indexamos uma tabela por uma cadeia de caracteres, podemos usar uma notação com ponto (`.`). Por exemplo, escrever:

```
t["versao"] = 3.1
```

é equivalente a escrever:

```
t.versao = 3.1
```

Esta notação é mais clara; seu uso é recomendado sempre que possível. Se a cadeia de caracteres for um nome composto, esta simplificação não é possível. Por exemplo:

```
t["Rio de Janeiro"] = 1994
```

não tem equivalência com uso de ponto para indexação.

Da mesma forma que armazenamos valores numéricos em tabelas, poderíamos ter armazenado outros valores. Todas as atribuições abaixo são válidas para a tabela `t`:

```
t[10] = "exemplificando"
t[2.34] = 12
t.tabela, t[0] = 3, 12
t[-234] = 0
a = "Lua"
t[a] = 5
```

Esta última atribuição merece uma análise cuidadosa. Considerando que `a` é uma variável que armazena o valor `Lua`, a última atribuição armazena `5` no campo (ou índice) `Lua` da tabela. Escrever `t[a]` é diferente de `t.a`, que é equivalente a `t["a"]`! Assim, no exemplo, `t[a]` é equivalente a `t.Lua`.

O valor de uma tabela para qualquer índice cuja indexação ainda não foi inicializada é `nil`. Assim, considerando somente as atribuições acima, o comando

```
print(t[999])
```

imprimirá o valor `nil`, pois o campo `999` de `t` não foi inicializado.

É possível armazenarmos outras tabelas em tabelas, criando conjuntos multidimensionais. Por exemplo:

```
m = { }
m[1] = { }
m[2] = { }
m[1][1] = 1.0
m[2][1] = 0.0
```

Também é válido escrever:

```
s = { }
s.dados = { }
s.dados.nome = "Linguagem Lua"
s.dados.versao = 3.0
```

Inicialização de tabelas via indexação consecutiva

Lua permite inicialização de tabelas na criação. Esta seção discute a inicialização de tabelas por indexação.

Uma forma possível para inicialização de tabelas é

{ *expr1*, *expr2*, ..., *exprN* **}**

Isto é, escreve-se uma lista de expressões entre as chaves que criam a tabela. Cada expressão listada é avaliada e seu valor armazenado no índice correspondente: o valor da primeira expressão é armazenado no índice 1 da tabela, o valor da segunda expressão no índice 2, e assim por diante. Portanto, a inicialização abaixo:

```
t = {23, 45, -7, "Lua", 6+4}
```

é equivalente a

```
t = { }
t[1] = 23
t[2] = 45
t[3] = -7
t[4] = "Lua"
t[5] = 6+4
```

Após a inicialização, a tabela criada pode armazenar outros valores. Isto é, continua sendo possível armazenar qualquer valor indexado por qualquer valor (exceto `nil`). Assim, seria possível, após a inicialização, ter:

```
t[67] = 0
t["Lua"] = "exemplo"
```

Inicialização de tabelas via atribuição de campos

Além da forma de inicialização mostrada na seção anterior, Lua permite a inicialização de campos da tabela. A forma geral é:

{ [*exprA***] =** *expr1*, **[***exprB***] =** *expr2*, ... **}**

Isto é, na criação de tabelas pode-se inicializar qualquer tipo de campo que uma tabela pode armazenar. Se o índice do campo da tabela for uma *string*, o par de valores pode ser escrito simplesmente

```
string1 = expr1
```

Assim, o comando

```
t = { nome = "Linguagem Lua", [1] = 3.0, [f(x)] = g(y) }
```

é equivalente a

```
t = { }
t.nome = "Linguagem Lua"
t[1] = 3.0
t.[f(x)] = g(y)
```

Mais uma vez, continua sendo possível armazenar qualquer valor indexado por qualquer valor (exceto `nil`). Assim, seria possível, após a inicialização, ter:

```
t[67] = 0
t["Lua"] = "exemplo"
```

Inicialização mista

Lua permite combinar as duas formas de inicialização mostradas nas seções anteriores. A forma geral é:

```
tabela = { lista-de-expressões ; lista-de-atribuições-de-campos
}
```

Assim, a lista de expressões de uma inicialização por indexação consecutiva pode vir seguida do caractere ponto e vírgula (`;`) e da lista de atribuições de campos por nomes. Portanto, escrever:

```
t = {23, 45, -7 ;
     nome="Linguagem Lua",
     versao=3.0,
     [4]=80
}
```

é equivalente a:

```
t = { }
t[1] = 23
t[2] = 45
t[3] = -7
t.nome = "Linguagem Lua"
t.versao = 3.0
t[4] = 80
```

9. Tratamento de erros

Todos os erros que podem ser gerados em Lua são "bem comportados" e podem ser controlados pelo programador. Quando ocorre um erro na compilação ou execução de códigos Lua, a função cadastrada como *error method* é executada e a a função da biblioteca Lua responsável pela execução do trecho (`lua_dofile`, `lua_dostring`, `lua_dobuffer` ou `lua_callfunction`) é cancelada, retornando uma condição de erro.

Lua passa para a função de *error method* um único parâmetro que é a cadeia de caracteres que descreve o erro ocorrido. A função *default* para *error method* simplesmente imprime essa cadeia de caracteres na saída padrão de erro (`stderr`). É possível mudar a função de *error method* através da função `seterrormethod` (veja seção 10). A biblioteca padrão de Lua para funções de entrada e saída usa essa facilidade para redefinir a função de tratamento de erros de forma a exibir algumas informações adicionais, como a pilha de chamada de funções de Lua.

Para se ter uma descrição completa dos erros de execução, incluindo a pilha de chamada de funções, deve-se incluir o pragma `$debug` no código Lua. Este pragma deve ser escrito na primeira coluna de uma linha do módulo.

10. Funções pré-definidas

Existem algumas funções pré-definidas que estão sempre disponíveis para programas Lua. O conjunto de funções pré-definidas em Lua é pequeno, mas provê poderosos recursos de programação. Como será discutido, estas funções possibilitam escrever funções para gerar códigos Lua que, se executados, restauram o ambiente de execução. Isto significa que objetos em Lua podem persistir entre diferentes sessões de execução.

dofile(*filename*)

DESCRIÇÃO
Esta função recebe como parâmetro o nome de um arquivo (*filename*), compila e executa seu conteúdo como um módulo da linguagem Lua.

ARGUMENTOS

filename	arquivo a ser executado

RETORNO
Retorna os valores retornados pelo módulo se a execução for bem sucedida ou `nil` se ocorrer erro.

EXEMPLO

```
if not dofile("bib.lua") then
  print("Não é possível abrir a biblioteca de funções")
end
```

OBSERVAÇÕES
Controlar o código de retorno da função pode ser útil para controlar erros que ocorrem em tempo de execução.

dostring(*string* [, *name*])

DESCRIÇÃO
Esta função recebe como parâmetro um valor do tipo *string*, compila e executa seu conteúdo como um módulo da linguagem Lua.

ARGUMENTOS

string	string a ser executada
name	opcional; usado em mensagens de erro.

RETORNO
Retorna os valores retornados pelo código se a execução for bem sucedida ou `nil` se ocorrer erro.

EXEMPLO
Esse exemplo transfere valores armazenados em variáveis da forma botaox (onde x é um número inteiro) para a tabela botao:

```
i = 1; botao = {}
b = dostring("return botao"..i )
while b do
  botao[i] = b
  i = i + 1
  b = dostring("return botao"..i )
end
```

next(*table*, *index*)

DESCRIÇÃO
Fornece um elemento da tabela *table* o próximo em relação ao elemento de índice *index*. Esta função permite enumerar todos os campos de uma tabela. O primeiro argumento da função é uma variável que armazena um valor do tipo *table*; o segundo argumento é uma variável que armazena um dos valores que indexa esta tabela. Este índice pode ser de qualquer tipo, já que uma tabela pode ser indexada por um valor de qualquer tipo (exceto `nil`). Quando a função `next` é chamada passando-se como segundo argumento o valor `nil`, retorna-se um primeiro índice da tabela e seu respectivo valor associado.

ARGUMENTOS

table	tabela Lua a ser examinada
index	índice da tabela considerado como referência; `nil` para começar.

RETORNO
A função retorna dois valores: o índice na tabela correspondente ao "próximo elemento" e o valor associado a esse índice. Se não existir o "próximo elemento", a função retorna `nil`.

EXEMPLO
O trecho de código abaixo imprime todos os índices e valores associados de uma dada tabela `t`:

```
campo,valor = next(t, nil)        captura primeiro campo da tabela
while campo do                    enquanto existir campo
 print(campo, "=", valor)         imprime valores
 campo,valor = next(t, campo)     captura próximo campo da tabela
end
```

OBSERVAÇÕES

Em Lua, os campos (índices) de uma tabela não precisam ser criados; eles são incorporados na tabela à medida em que são atribuídos. Para um campo ainda não atribuído, assume-se o valor `nil`. Portanto, não há diferença semântica entre campos realmente inexistentes na tabela e campos que armazenam o valor `nil`. Por isso, a função `next` apenas considera campos que armazenam valores diferentes de `nil`. A ordem em que os índices aparecem não é especificada.

nextvar(*name*)

DESCRIÇÃO

Esta função permite enumerar todas as variáveis globais cujo valor seja diferente de `nil`. A função recebe um único parâmetro que representa o nome de uma variável global. No caso de ser passado o valor `nil` para a função, retorna-se o nome e o valor de uma primeira variável global. A função sempre retorna dois valores: nome da variável e valor armazenado. O nome retornado por uma chamada da funcão pode ser usado para acessar uma próxima variável global, de modo similar à função `next`. A ordem que as variáveis são retornadas não é especificada.

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(*value*)

DESCRIÇÃO

Esta função 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(*e* [,*base*])

DESCRIÇÃO
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(*e*)

DESCRIÇÃO
Esta função 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(*expr1*, *expr2*, ...)

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

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

RETORNO
Nenhum.

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.

error(*msg*)

DESCRIÇÃO
Esta função gera uma condição de erro de execução, finalizando a função chamada de C (`lua_dofile`, `lua_dostring`, ...). Esta função recebe uma cadeia de caracteres para descrever o erro ocorrido.

ARGUMENTOS

msg	texto descritor do erro

RETORNO

Nenhum.

OBSERVAÇÕES

A função interrompe a execução do programa; ela nunca retorna.

call(*func*, *arg* [,*retmode*])

DESCRIÇÃO

Chama a função *func* com os argumentos contidos na tabela *arg*, percorrendo sequencialmente os índices numéricos até o índice *arg*.n. Se *arg*.n não for definido, então a lista de argumentos é considerada terminada ao ser encontrado o primeiro elemento de *arg* com valor `nil`.

ARGUMENTOS

func	função a ser chamada
arg	tabela contendo todos os argumentos para a função
retmode	opcional; só o valor `"pack"` é admitido

RETORNO

Retorna todos os valores que a função *func* retornar. Se o argumento *retmode* tiver recebido o valor `"pack"`, então o retorno é uma tabela contendo os valores resultantes nos índices inteiros e o número total de resultados no índice `n`.

EXEMPLO

Considere as seguintes chamadas:

```
a = call( sin, {5} )                  a = 0.871557
a = call( max, {1,4,5;n=2})           a = 4
t = { x=1 }
a = call( next,{t,nil;n=2},"pack" )   a={"x",1;n=2}
```

assert(*value* [,*message*])

DESCRIÇÃO

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 à `nil`.

ARGUMENTOS

value	valor a ser testado

RETORNO

Nenhum.

EXEMPLO

```
assert( readfrom(FILE),"cannot open" .. FILE )
```

11. Bibliotecas de funções

Existem três bibliotecas já implementadas provendo funções C que podem ser acessadas de Lua: manipulação de *strings*, funções matemáticas e funções de entrada e saída. Estas bibliotecas são distribuídas com a biblioteca básica da linguagem.

Biblioteca de manipulação de *strings*

Esta biblioteca provê funções genéricas para manipulação de cadeias de caracteres, tais como determinar o número de caracteres de uma *string* e capturar ou localizar uma *substring*.

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

DESCRIÇÃO

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

O código

```
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 )
```

Este código imprime os valores 13, 4, e 1997.

OBSERVAÇÕES

Padrões operam sobre classes de caracteres, que são conjuntos de caracteres. A tabela abaixo lista as classes que podem ser utilizadas para compor os padrões:

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 um padrão 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 sequência mais longa possível;
- uma classe de caracteres seguida de `-`, que casa com 0 ou mais repetições dos caracteres da classe. Ao contrário do `*`, o casamento é sempre feito com a sequência 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 (ver abaixo);

- %b*xy*, onde *x* e *y* são dois caracteres diferentes; este item casa com strings que começam com *x* e terminam com *y*, onde *x* e *y* são balanceados. Por exemplo, o item `%b()` casa com expressões com parênteses balanceados.

Um padrão é uma sequência de itens de padrão. O caracter `^` no início do padrão obriga o casamento no início da string procurada. Um `$` no final do padrão obriga o casamento no final da string.

Um padrão pode conter sub-padrões 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. Por exemplo, no padrão `"(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.

EXEMPLO
O código

```
function name (arg1,arg2)
(function %s*(%a%a*)(%b()))
```

```
captura as seguintes strings: declarações de funções Lua ; nome
da função (%a%a*); captura lista de parâmteros (%b()).
```

strlen(*str*)

DESCRIÇÃO
Informa o tamanho de uma *string*.

ARGUMENTOS
str string a ser medida

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

EXEMPLO
O código

```
print(strlen("Linguagem Lua"))
```

imprime o valor `13`.

strsub(*str*, *I* [, *j*])

DESCRIÇÃO
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

O código:

```
a = "Linguagem Lua"
print(strsub(a, 11))
```

imprime a *string* `Lua`.

strlower(*str*)

DESCRIÇÃO

Cria uma cópia da cadeia 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

O código

```
print(strlower("Linguagem Lua"))
```

tem como saída a cadeia `linguagem lua`.

strupper(*str*)

DESCRIÇÃO

Cria uma cópia da cadeia 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 cadeia resultante.

EXEMPLO
O código

```
print(strupper("Linguagem Lua"))
```

tem como saída a cadeia `LINGUAGEM LUA`.

strrep(*str*, *n*)

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

ARGUMENTOS
str *string* a ser replicada

RETORNO
Retorna a *string* criada.

EXEMPLO
O código

```
print(strrep( "0123456789", 2 ))
```

tem como saída

```
01234567890123456789
```

ascii(*str* [, *i*])

DESCRIÇÃO
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
O código

```
print(ascii("abc",2))
```

tem como saída 42.

format(*formatstring*, *exp1*, *exp2*, …)

DESCRIÇÃO
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.

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
O código abaixo:

```
nome = "Carla"
id = 123
cmd = format( "insert into tabela (nome, id)
               values ('%s', %d)" , nome, id )
```

cria uma string com um comando SQL para inserção dos valores da variáveis nome e id em uma tabela. Já o código abaixo:

```
print( format( "%c", 65 ) )
```

mostra como a função pode ser usada para converter um código ASCII númerico no caracter correspondente; nesse caso resultando na impressão do caracter `A`.

O exemplo abaixo mostra o uso de modificadores:

```
a = 123.456
print( format( "%+010.2f", a ) )
```

imprime `+000123.46`.

OBSERVAÇÕES
A tabela abaixo lista os tipos aceitos no formato. Os tipos `%s` e `%q` aceitam como valor uma string, enquanto que os outros aceitam um número.

%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 %

A formatação de cada um dos tipos pode ser alterada incluindo-se modificadores entre os dois caracteres. Um número define o tamanho mínimo que o campo ocupa (ex: %10d). Um número seguido de um ponto define a precisão do campo (ex: %.2f). Um sinal de - após o % significa que o campo deve ser alinhado pela esquerda, dentro do tamanho do campo. Um sinal de + faz com que seja mostrado este sinal caso o valor correspondente seja positivo. O caracter 0 faz com que o número seja preenchido com zeros à esquerda para completar o tamanho mínimo.

gsub(*str, pattern*, *repl* [, *n*])

DESCRIÇÃO

Procura e substitui padrões em uma cadeia de caracteres. O padrão *pattern* é procurado na *string str*. Cada padrão encontrado é substituído por *repl*, que pode ser uma *string* ou uma função. Caso *repl* seja uma função, o padrão encontrado é substituído pelo resultado da execução da função *repl*. As substituições são feitas para cada ocorrência do padrão *pattern*, a não ser que seja especificado o parâmetro *n*, que define o número máximo de substituições a serem feitas. Se o padrão especifica capturas, estas podem ser usadas na *string repl* na forma %1, %2 etc; se *repl* for uma função, as capturas são passadas como parâmetro para a função. A descrição do padrão é a mesma utilizada pela função *strfind*.

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 função
n	opcional; número máximo de substituições a serem feitas

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*.

EXEMPLO

O código

```
print( gsub( "Linguagem Lua 3.0", " ", "+" ) )
```

imprime `Linguagem+Lua+3.0`.

O código

```
texto = "    Linguagem      Lua      3.0   "
print(  gsub( texto, "  *", " " ) )
```

substitui toda sequência de um ou mais brancos por um único branco, imprimindo "`Linguagem Lua 3.0 `".

Capturas também podem ser utilizadas com a função gsub:

```
lista = [[
arq.txt
texto.txt
r.txt
]]
print( gsub( lista, "(.*)%.txt\n", "move %1.txt %1.bak\n" ))
```

imprime:

```
move arq.txt arq.bak
move texto.txt texto.bak
move r.txt r.bak
```

Biblioteca de funções matemáticas

Existem implementadas funções para retornar o valor máximo ou mínimo de uma série de expressões. As funções recebem uma lista variável de parâmetros. A forma geral das funções é:

```
min( expr1, expr2, ..., exprN)
max( expr1, expr2, ..., exprN)
```

Assim, o código abaixo:

```
print(min(23, 5, 123, 3))
print(max(23, 5, 123, 3))
```

imprime os valores `3` e `123`.

Existem ainda funções que implementam as funcionalidades de funções homônimas da biblioteca padrão da linguagem C. São elas:

`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`

Biblioteca de funções de entrada e saída

Esta biblioteca implementa funções adicionais para execução de operações de entrada e saída. Todas as operações de entrada e saída desta biblioteca são feitas em dois arquivos correntes, um de entrada (`_INPUT`) e um de saída (`_OUTPUT`). Os dispositivos padrões da linguagem C: `stdin`, `stdout` e `stderr` estão disponíveis através das variáveis globais `_STDIN`, `_STDOUT` e `_STDERR`. Inicialmente, `_INPUT` tem o mesmo valor de `_STDIN` e `_OUTPUT` tem o mesmo valor que `_STDOUT`.

readfrom(*filename*)

DESCRIÇÃO

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" )
```

writeto([*filename*])

DESCRIÇÃO

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

O código abaixo:

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

cria o arquivo `a.txt`, escrevendo a string `conteúdo do arquivo` nele.

appendto([*filename]*)

DESCRIÇÃO

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
O código

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

acrescenta a string `conteúdo do arquivo` no arquivo `a.txt`.

read([*readpattern]*)

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

ARGUMENTOS
`readpattern` padrão a ser lido

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

EXEMPLO
O codigo

```
data = {}
i = 1
readfrom( "datas.txt" )
repeat
  data[i] = read(  "%d/%d/%d{\n}" )
  i = i + 1
until data == nil
readfrom()
```

lê todas as datas contidas no arquivo `datas.txt`, colocando-as na tabela `data`.

OBSERVAÇÕES
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.

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

DESCRIÇÃO
Recebe uma lista de valores e os escreve no arquivo de saída corrente (`_OUTPUT`). 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(*filename*)

DESCRIÇÃO
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
O código

```
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(*name1*, *name2*)

DESCRIÇÃO
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
O código

```
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( )

DESCRIÇÃO
Obtem 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
O código

```
filename = tmpname()
writeto( filename )
```

cria um arquivo temporário para escrita.

date([*format*])

DESCRIÇÃO
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
O código

```
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).

OBSERVAÇÕES
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
`%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
`%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([*code*])

DESCRIÇÃO
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
O código

```
if termina_programa then
  exit()
end
```

termina a execução do programa caso a variável *termina_programa* contenha um valor diferente de `nil`..

getenv(*varname*)

DESCRIÇÃO
Consulta o valor da variável de ambiente nomeada *varname*.

ARGUMENTOS
varname nome da variável a ser consultada.

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

EXEMPLO
O código

```
print( getenv( "REMOTE_HOST" ) )
```

imprime o valor da variável de ambiente `REMOTE_HOST`.

execute(*command*)

DESCRIÇÃO
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
O código

```
execute( "mkdir c:\data" )
```

executa um comando no sistema operacional para criar o diretório `c:\data`.

Apêndice. Relevância de Lua

Um dos aspectos mais marcantes de Lua é sua dualidade acadêmico-industrial. No aspecto industrial, Lua vem sendo usada em dezenas de produtos, desenvolvidos tanto na PUC-Rio quanto fora, inclusive no exterior. No aspecto acadêmico, Lua deu origem e influenciou diversos trabalhos, refletidos em dezenas de dissertações, teses, e publicações.

Relevância Tecnológica

Lua vem sendo disponibilizada na Internet desde fevereiro de 1994. Esse fato, aliado à qualidade da linguagem, vem tornando Lua uma linguagem amplamente utilizada em diversos tipos de aplicação, tanto no Brasil quanto no exterior. Abaixo relatamos alguns usos de Lua em produtos tecnológicos. Essa lista está longe de ser exaustiva, e procura apenas exemplificar a diversidade de projetos industriais em que Lua vem sendo utilizada.

É importante frizar que, devido à distribuição pela Internet, os próprios autores não tem controle do número total de usuários ou produtos/protótipos que utilizam a linguagem.

TeCGraf

O TeCGraf é uma parceria entre a PUC-Rio e o Centro de Pesquisas da PETROBRÁS (CENPES) com o objetivo de desenvolver, implantar e manter software de computação gráfica e de interface com o usuário para aplicações técnico-científicas. Como a linguagem Lua foi desenvolvida no TeCGraf, é bastante natural que o grupo tenha adotado Lua em diversos projetos. Atualmente, mais de vinte pessoas programam regularmente em Lua no TeCGraf, e em torno de mais dez pessoas usam Lua eventualmente. No total, os diversos sistemas produzidos no TeCGraf com tecnologia Lua incluem mais de cem mil linhas de código Lua, em dezenas de produtos finais. O artigo da "Software: Practice & Experience" também descreve algumas aplicaçõe de Lua no TeCGraf. (Maiores informações sobre o TeCGraf e seus produtos podem ser obtidas em `http://www.tecgraf.puc-rio.br/publico/prodtec.html`.)

Sistema RPA@PUC

O RPA@PUC (Rede de Perfis Acadêmicos da PUC-Rio) é um sistema Intranet desenvolvido pela Vice-Reitoria Acadêmica da PUC-Rio, para acompanhamento da produção acadêmica do corpo docente da Universidade. Para suportar mais de cinquenta diferentes tipos de produção ("Artigos de Periódicos", "Capítulos de

Livros", "Resenhas", "Traduções e Versões", "Material Didático e Instrucional", "Participação em Bancas", etc.) o sistema utiliza tabelas Lua para descrição abstrata de cada tipo de item. Baseados nessas tabelas, programas escritos em Lua são acionados, através do protocolo CGI, para gerar dinamicamente formulários, relatórios, crítica de dados, e outros documentos WWW. Com isso, a inclusão ou alteração de um tipo de produção implica apenas em modificações na descrição abstrata de itens, sem envolver nenhum tipo de programação. Atualmente, o sistema está sendo usado regularmente por aproximadamente cento e cinquenta docentes e funcionários de oito Departamentos da PUC-Rio, e em breve deverá estar sendo usado por toda a Universidade. (Outras informações sobre o projeto RPA@PUC podem ser obtidas em `http://www.inf.puc-rio.br/~info-rpa/projeto/`).

Um sistema semelhante está sendo usado para manutenção do "site" do PESC (Programa de Engenharia de Sistemas e Computação) da COPPE/UFRJ (maiores informações em `http://www.cos.ufrj.br/equipe/webmaster.html`).

Medialab

A Medialab, empresa responsável pelos "sites" brasileiros na Internet de empresas como Shell e Microsoft, tem usado Lua na construção de diversas páginas dinâmicas (protocolo CGI). Ao todo, mais de dez "sites" já estão em funcionamento com partes dinâmicas escritas em Lua, entre eles o sistema de busca de endereços e telefones da Volkswagen do Brasil (`http://www.volkswagen.com.br/indbusca.htm`) e o sistema de busca do Guia de Restaurantes Danusia Bárbara (`http://www.brazilweb.com/rio/`).[4]

Non-Linear Control Consultants Ltd

O uso de Lua relatado abaixo é de Mark Ian Barlow, que trabalha na empresa Non-Linear Control Consultants Ltd., na Inglaterra. Em uma mensagem, ele relata seu primeiro contato com a linguagem; aparentemente, ele soube de Lua através do artigo na Dr. Dobb's:

[4] Os URLs fornecidos não são de páginas criadas com Lua, mas sim páginas estáticas que acessam as páginas dinâmicas criadas com a linguagem. O uso de Lua pode ser conferido, se necessário, vendo-se as fontes dessas páginas, onde as ações de botões referem-se a roteiros Lua (terminação `.lua` via programa CGILua). Maiores informações sobre a biblioteca CGILua podem ser obtidas em `http://www.tecgraf.puc-rio.br/manuais/cgilua`.

```
From Mark@nlcc.demon.co.uk Mon Jan 13 12:59:13 1997
Date: Mon, 13 Jan 1997 01:24:47 GMT
From: Mark@nlcc.demon.co.uk (Mark Ian Barlow)
Reply-To: Mark@nlcc.demon.co.uk
Message-Id: !532@nlcc.demon.co.uk?
To: lua@icad.puc-rio.br
Subject: Thanks!

Hi, For some time now I've been looking for a portable,
compact interpreted or incrementally compiled language to
use in various robotic applications. […] I think lua is
going to be just what I need! Writing to hardware is of
course simple with CFunctions, but the debug interface
(omitted from almost all the embedded languages I've seen
before) allows me to arrange for incoming data to appear
"magically" in reserved global variables. […] Thanks again
for so accurately spotting the need for lua. Is there a
mailing list for it, and have you heard of anyone else
considering using it as a front-end for operating custom
hardware?
```

Uma segunda mensagem, a pedido dos autores, relata seu uso atual:

```
From Mark@nlcc.demon.co.uk Sat Apr 5 09:56:02 1997
From: Mark@nlcc.demon.co.uk (Mark Ian Barlow)
To: lhf@server02.lncc.br
Subject: Re: applications that use Lua

I'm using Lua to sequence high-level commands to robots,
connected to the host either via a serial line or transputer
link; perhaps in future via USB or FireWire. Lua is
particularly valuable here because the debugger hooks give
me a means to make external asynchronous events (eg: touch-
sensors) look like atomic actions as far as the Lua program
is concerned, and allow users to assign call-back functions
to them. Essentially the robot is a "manipulation server"
and the Lua program it's client. The second major advantage
is the portability; I can easily compile a stripped-down Lua
client for an embedded processor on the robot's motherboard.
The user can then download their final debugged application
into non-volatile memory, disconnect the host and leave the
robot to get on with the job.
```

Relevância Acadêmica

Apesar de sua motivação tecnológica, Lua tem tido uma grande relevância acadêmica desde o início de seu projeto. As sub-áreas da computação nas quais o trabalho com a linguagem Lua se mostrou mais relevante foram Linguagens de Programação e Engenharia de Software.

Na linha de Linguagens, Lua oferece diversos mecanismos inovadores; porém, o mais importante é sua economia de conceitos, que a torna significativamente mais simples e menor que outras linguagens com recursos semelhantes. Entre outras

estruturas que contribuem para isso, podemos destacar o mecanismo de tabelas e sua identificação com o conceito de objetos, e o mecanismo de fallbacks (tag method).

Na linha de Engenharia de Software, o principal impacto de Lua foi promover o método "bottom-up" de desenvolvimento, com uso intensivo de componentes. O uso de Lua como "glue-language" permite que partes significativas de um sistema possam ser desenvolvidas como bibliotecas, com um pequeno roteiro em Lua definindo a arquitetura final de uma aplicação.

Publicações

Uma das maneiras de mostrar a relevância acadêmica de Lua é através das publicações geradas em torno da linguagem. Abaixo, apresentamos a lista dessas publicações. Essa lista é dividida em duas partes: a primeira contém artigos diretamente relacionados com a linguagem. A segunda contém trabalhos baseados em Lua. Vale frizar que os trabalhos baseados em Lua, de maneira geral, não só utilizaram a linguagem em sua implementação, mas também sofreram influência de Lua em suas arquiteturas, sendo desenvolvidos segundo a linha "bottom-up" promovida pela linguagem.

Trabalhos Sobre Lua

- R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. "Lua – an extensible extension language." Software: Practice and Experience, 26(6):635–652, 1996.

- Luiz Henrique de Figueiredo, Roberto Ierusalimschy, and Waldemar Celes. "Lua: An extensible, portable embedded language." Dr. Dobb's Journal, 21(12):26–33, December 1996.

- Tomás G. Gorham and Roberto Ierusalimschy. "Um sistema de depuração reflexivo para uma linguagem de extensão." I Simpósio Brasileiro de Linguagens de Programação, páginas 103–114, Belo Horizonte, setembro 1996.

- Noemi Rodriguez, Cristina Ururahy, Roberto Ierusalimschy, and Renato Cerqueira. "The use of interpreted languages for implementing parallel algorithms on distributed systems." In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, Euro-Par'96 Parallel Processing – Second International Euro-Par Conference, páginas 597–600, Volume I, Lyon, France, August 1996. Springer-Verlag. (LNCS 1123).

- Tomás Guisasola Gorham. "Um sistema de depuração para a linguagem de extensão Lua." Dissertação de Mestrado, Dep. Informática, PUC-Rio, março 1996.

- L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. "The design and implementation of a language for extending applications." XXI Semish, páginas 273–284, Caxambu, 1994.

- Roberto Ierusalimschy, Waldemar Celes Filho, Luiz Henrique de Figueiredo, Roberto Murilo de Sousa. “Lua: uma linguagem para customização de aplicações.” Caderno de Ferramentas do VII Simpósio Brasileiro de Engenharia de Software, página 55, 1993.

Trabalhos Baseados em Lua

- André Oliveira Costa. “TkVIX – um toolkit para construção de interfaces gráficas em Lua.” Dissertação de Mestrado, Dep. Informática, PUC-Rio, abril 1997.

- Paulo Henrique Mascarenhas Sant’Anna. “Modelos de extensão de agentes para gerenciamento de redes.” Dissertação de Mestrado, Dep. Informática, PUC-Rio, abril 1997.

- R. O. Prates, C. S. de Souza, A. C. B. Garcia. “A Semiotic Framework for Multi-User Interfaces.” ACM SIGCHI Bulletin 29(2):28–39, 1997.

- Nick Lehtola. “Ambiente para o desenvolvimento de sistemas CAD integrado em edificações de aço.” Tese de doutorado, Dep. Engenharia Civil, PUC-Rio, Abril 1996.

- André Luiz Clínio dos Santos. “VIX – um framework para suporte a objetos visuais interativos.” Dissertação de Mestrado, Dep. Informática, PUC-Rio, dezembro 1996.

- Neil Armstrong Rezende. “GLB: Uma ferramenta para manipulação de objetos gráficos procedurais.” Dissertação de Mestrado, Dep. Informática, PUC-Rio, março 1996.

- Carlos Roberto Serra Pinto Cassino. “Uma ferramenta para programação literária modular.” Dissertação de Mestrado, Dep. Informática, PUC-Rio, agosto 1996.

- Waldemar Celes Filho. “Modelagem configurável de subdivisões planares hierárquicas.” Tese de doutorado, Dep. Informática, PUC-Rio, julho 1995.

- Marcelo Tílio Monteiro de Carvalho. “Uma estratégia para o desenvolvimento de aplicações configuráveis em mecânica computacional.” Tese de doutorado, Dep. Engenharia Civil, PUC-Rio, junho 1995.

- A. Carregal, R. Ierusalimschy. “Tche – a visual environment for the Lua language.” VIII Simpósio Brasileiro de Computação Gráfica, páginas 227–232, São Carlos, 1995.

- R. Cerqueira, N. Rodriguez, R. Ierusalimschy. “Uma experiência em programação distribuída dirigida por eventos.” PANEL95 – XXI Conferência Latino Americana de Informática, páginas 225–236, 1995.

- Waldemar Celes Filho, Luiz Henrique de Figueiredo, Marcelo Gattass. "EDG: uma ferramenta para criação de interfaces gráficas interativas." VIII Simpósio Brasileiro de Computação Gráfica, páginas 241–248, São Carlos, 1995.

- M. T. de Carvalho, L. F. Martha. "Uma arquitetura para configuração de modeladores geométricos: aplicação a mecânica computacional." PANEL95 – XXI Conferência Latino Americana de Informática, páginas 123–134, 1995.

- Neil Armstrong Rezende, Waldemar Celes Filho, Marcelo Gattass, Ênio E. Ramos Russo. "PRE-GO: Um editor de objetos gráficos procedurais." VIII Simpósio Brasileiro de Computação Gráfica, páginas 309–310, São Carlos, 1995.

- Siome Klein Goldenstein, Paulo Cezar Pinto Carvalho, Luiz Henrique de Figueiredo. "IG: um sistema de construções geométricas com vínculos." VIII Simpósio Brasileiro de Computação Gráfica, páginas 311–312, São Carlos, 1995.

- André Filipe Lessa Carregal. "Tche: Um ambiente visual Lua." Dissertação de Mestrado, Dep. Informática, PUC-Rio, março 1995.

- Eduardo Setton Sampaio da Silveira. "Um sistema de modelagem bidimensional configurável para simulação adaptativa em mecânica computacional." Dissertação de Mestrado, Dep. Engenharia Civil, PUC-Rio, agosto 1995.

Lua na Internet

Lua vem sendo disponibilizada na Internet desde fevereiro de 1994. (sua *home-page* pode ser acessada em `http://www.inf.puc-rio.br/~roberto/lua.html`). Após a publicação de um artigo de divulgação na Dr. Dobb's, no final de 1996, o interesse externo pela linguagem tem crescido constantemente. O pacote está disponível em quatro locais, no Brasil, Canadá, Alemanha e Grécia, sendo os dois últimos mantidos por grupos totalmente independentes.

Atualmente, a linguagem conta com uma lista de discussões (lua-l@tecgraf.puc-rio.br) com mais de setenta participantes, sendo a maioria deles de fora do país (a lista de participantes pode ser obtida enviando-se `review lua-l` para listproc@tecgraf.puc-rio.br).

Várias coleções de linguagens e/ou ferramentas na Internet já incluem Lua. Entre outras, citamos as seguintes:

- `http://www.KachinaTech.COM/~hjjou/new-SAL/F/1/LUA.html`. Página mantida pela SAL: "SAL (Scientific Applications on Linux) is a collection of information and links of software that scientists and engineers will be interested in. The broad coverage on Linux applications will also benefit the whole Linux/Unix community."

- `http://www.idiom.com/free-compilers/TOOL/lua-1.html`. Parte do Catalog of Free Compilers and Interpreters: "This list is primarily aimed at developers

rather than researchers, and consists mainly of citations for production quality systems."

Abaixo listamos alguns comentários sobre Lua que circularam pela rede. Alguns foram mandados diretamente para os autores, outros apareceram em grupos de discussão ("newsgroups") sobre linguagens. É interessante observar a diversidade das instituições, através dos endereços dos remetentes.

- From: W.Boeke, Lucent Technologies, the Netherlands

```
From wboeke@hzsda01.nl.lucent.com Thu Jul 18 04:31:22 1996

After the article in 'Software, P.&E.' you must have had a
lot of responses. Everyone with some knowledge of computer
languages will have been struck by the original concepts of
the language.

[…]
```

- From: "Francis L.Burton" !gpaa29@udcf.gla.ac.uk?

```
From udcf.gla.ac.uk!gpaa29 Fri Jan 26 14:20:34 1996

[…] Lua is a splendid piece of software, imho -- very clean
language design, powerful through its reflexivity, nice API.
The documentation is a pleasure to read. Would that all
software were as neat as Lua! […] Frankly, I was amazed and
delighted. […]
```

- From: Joshua Jensen !joshj@doc.sunrem.com?

```
[…] Congratulations on Lua. It was exactly what I was
looking for, and I've been using it as my primary script
language for a month and a half now. It's powerful, small,
and very easy to use (and expand upon). […] Good luck with
it. I'd recommend the language to anyone.
```

- From: "Rodney A. Ludwig"!rludwig@rrnet.com?

```
[…] I have been using interpreters in my code since the
early 80s for testing and prototyping and to add a macro
language […] so far this looks like the best I have seen and
I have tried most everything around during that time. […]
```

- From: fburton@nyx10.cs.du.edu (Francis Burton)

```
Article: 8741 of comp.compilers
Message-ID: !96-11-066@comp.compilers?
Organization: University of Denver, Math/CS Dept.

[…] If you really want to learn about elegance and power in
the design of little languages, take a look at the papers
published about Lua (metapage at
http://csg.uwaterloo.ca/~lhf/lua/) and study the source
code. Neat!
```

- From: Bret Mogilefsky !mogul@lucasarts.com?

```
After reading the Dr. Dobbs article on Lua I was very eager
to check it out, and so far it has exceeded my expectations
in every way! It's elegance and simplicity astound me.
Congratulations on developing such a well-thought out
language. […]
```

- Newsgroups: comp.lang.ml

```
Message Id: !5p6bbi$8cd$1@goldenapple.srv.cs.cmu.edu?

[…] I'm not so happy with Tcl as language design, and for C
I think Lua does a better job than Tcl. […]

Norman Ramsey -- moderator, comp.programming.literate
http://www.cs.virginia.edu/~nr
```

- From: jack@cwi.nl (Jack Jansen)

```
Message-Id: !jack.860671284@news.cwi.nl?
Newsgroups: comp.lang.python

[…] From a glance at
http://csg.uwaterloo.ca/~lhf/lua/home.html it looks like a
pretty decent and small language, "Tcl done right" is what
immedeately came to mind […]
```

- From: jamesl@netcom.com (James Logajan)

Message-Id: !jameslE8EL5A.vF@netcom.com?
Newsgroups: comp.lang.scheme, comp.lang.scheme.scsh, comp.lang.lisp,comp.lang.tcl,comp.lang.functional,comp.lang.c++,comp.lang.perl.misc, comp.lang.python,comp.lang.eiffel

John Ousterhout (ouster@tcl.eng.sun.com) wrote: "Every single programmer who ever wrote a program in Tcl, Perl, C++, Visual Basic, or even C could have chosen Lisp, Scheme, or Smalltalk. But they didn't."

Others have responded to this point adequately; however, I just happen to be working on a product that is going to require a scripting or extension language. We get to pick our own language (for once). Our short list includes:

Lua
Python
Perl
Tcl

I have not used any of these before, so have "fresh eyes" on our team. We need a language that is easy for beginners to pick up, that is not too slow, handles string processing well, has a small memory footprint, and is of course extensible. OOP support was considered important, but not vital.

So far, Lua is winning, with Python still in the running because of its stronger OO support. Tcl is horribly slow with no particular feature that compensates for its sluggishness. Perl seems too syntactically unruly for people to get a good grip on how best to compose their work but has about the same speed as Python.

Lua is about 1/10 the size of the others (70k versus 1000k for all the others on a Sparc) and is several times faster than Python and Perl. Tcl is an order of magnitude (10, not 2!) slower than Python/Perl.

It is too bad that Lua doesn't have its own newsgroup (yet?). I was pleasantly surprised by the small size, fast speed, simple syntax, and powerful semantics of the language.