

Universidade Federal do Rio de Janeiro  
Departamento de Engenharia Eletrônica e de Computação

**Relatório II**  
**Análisor léxico da linguagem Lua**

Alunos: Bernardo Antonio Boechat Florencio  
João Pedro Soares Alves Fernandes  
Professor: Miguel Elias Mitre Campista

Outubro  
2019

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Implementação</b>	<b>2</b>
<b>3</b>	<b>Casos de Uso</b>	<b>3</b>
3.1	Entradas válidas . . . . .	3
3.2	Erros . . . . .	7
<b>4</b>	<b>Conclusão</b>	<b>8</b>
	<b>Referências</b>	<b>9</b>

# 1 Introdução

Neste texto, apresentamos o programa que foi desenvolvido pelos autores para o Trabalho II da disciplina Linguagens de Programação. Como previamente explicitado no Relatório I, esse programa consiste em um *lexer* para um interpretador da linguagem de programação Lua.

O *lexer* realiza a fase inicial da interpretação, denominada análise léxica. Nessa etapa, a sequência de caracteres contidas no código fonte é escaneada e são construídos *tokens* que representam os lexemas da linguagem, ou seja, agrupamentos de caracteres que representam um significado. Desta forma, as partes posteriores do interpretador podem operar sobre níveis de abstração maiores.

A Seção 2 deste relatório explicita os detalhes do programa escrito, incluindo as funcionalidades implementadas. Em seguida, a Seção 3 apresenta resultados da execução do programa para algumas entradas, destacando algumas informações pertinentes acerca da linguagem e das decisões de projeto. Por fim, a Seção 4 conclui o texto.

## 2 Implementação

O lexer foi escrito em Perl. Esta linguagem é apropriada para essa seção do interpretador devido ao seu alto potencial para processamento de texto por meio de expressões regulares, que possibilitam a síntese de lógicas de reconhecimento de padrões de lexemas.

A entrada do programa é o nome de um arquivo texto contendo o código fonte, em Lua, a ser analisado, enquanto a saída corresponde à sequência de *tokens* de Lua contidos no arquivo. Para a extração correta destes *tokens*, são realizadas cinco tipos de análises: leitura de *strings*, leitura de números, leitura de sequências especiais de símbolos, leitura de palavras-chave e reconhecimento de comentários. É válido notar também que o analisador léxico ignora espaços em branco no código, inclusive quebras de linha, característica que vai de acordo à especificação da linguagem Lua.<sup>1</sup>

Todas as funções foram implementadas com sucesso e, em maior parte, com fidelidade. Ainda assim, é importante indicarmos os desvios mais evidentes da especificação.

Primeiramente, não foram implementados níveis de aninhamento de comentários. Além disso, também não é possível escrever *strings* ao longo de múltiplas linhas (na especificação original, para este fim faz-se uso do caracter ”\” seguido da quebra de linha). A delimitação de *strings* a partir dos *long brackets* de ordem n diferente de 0 também não foi contemplada.<sup>2</sup> Também não foi incluída a especificação de caracteres, em *strings*, a partir de sequências unicode. Por fim, não foi implementada a especificação de números a partir de sequências em hexadecimal.

Dentro do escopo e do objetivo didático do trabalho, as limitações supracitadas foram consideradas aceitáveis pelos autores, devido a essas estruturas não estarem presentes na maioria dos códigos em Lua, e suas ausências não comprometerem a programação na linguagem.

---

<sup>1</sup>Linguagens como C e C++ utilizam o caracter ponto e vírgula como terminador de declarações, de modo que se torna fácil ignorar quebras de linha. Linguagens sem terminador explícito, em sua maioria, não são capazes de ignorá-las sem ambiguidade; este é o caso de, por exemplo, Python. Lua é especial nesse sentido pois, apesar de não possuir terminador explícito, consegue ignorar quebras de linha devido a uma gramática cuidadosamente controlada para evitar ambiguidades.

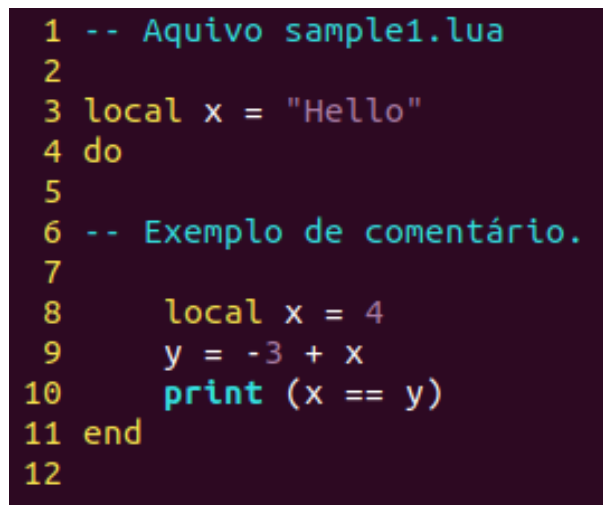
<sup>2</sup>Em Lua, um *long bracket* de ordem n é definido como a sequência de caracteres ”[X]”, onde X corresponde a n caracteres ”=”. Desse modo, o de ordem 0 é ”[[”, o de ordem 1 é ”[=” etc. A cada um destes está associado o *long bracket* terminador correspondente ”]X]”. Na especificação da linguagem, *long brackets* de quaisquer ordem podem delimitar uma *string*, enquanto o nosso trabalho se ateve aos de ordem 0.

## 3 Casos de Uso

Nessa seção, são apresentados os resultados da execução do programa para alguns arquivos de entrada, de modo a exemplificar o funcionamento do analisador léxico. Em seguida, mostramos alguns erros, tratados pelo programa, que podem ser obtidos pelo usuário.

### 3.1 Entradas válidas

A Figura 1 mostra o código do primeiro arquivo a ser utilizado, denominado *sample1.lua*, enquanto a Figura 2 apresenta a saída correspondente do programa.



```
1 -- Arquivo sample1.lua
2
3 local x = "Hello"
4 do
5
6 -- Exemplo de comentário.
7
8     local x = 4
9     y = -3 + x
10    print (x == y)
11 end
12
```

Figura 1: Código do arquivo *sample1.lua*

Este exemplo serve como ilustração para algumas características do analisador léxico.

Em primeiro lugar, observe-se que, por decisão de projeto, cada palavra-chave corresponde a um tipo de *token* diferente como, por exemplo, *KW\_LOCAL* e *KW\_DO*. A mesma regra se aplica a cada sequência de símbolos, como *SINGLE\_EQUAL* e *DOUBLE\_EQUAL*. Nesses casos, e nos demais em que o tipo do *token* é suficiente para determiná-lo, ao valor do *token* é atribuída a *strings* "None".

Além disso, nota-se que, além das informações léxicas captadas de cada *token*, também é obtida a linha em que esse *token* se encontra. Isto é para que, caso ocorra erro na interpretação, mensagens de erro melhores possam ser impressas na tela

Por fim, pode ser visto que o analisador acrescenta um *token* especial de tipo *EOS* (do inglês *end of string*) <sup>3</sup> ao final da sequência. Embora não estritamente necessária, essa

---

<sup>3</sup>Por estarmos lidando exclusivamente com códigos-fontes contidos em arquivos, pode-se julgar mais apropriado o uso da sigla *EOF* (do inglês *end of file*). Entretanto, *EOS* foi adotada por ser terminologia comum no contexto de interpretadores que, como no caso de interpretação interativa, muitas vezes não lidam com arquivos.

```

Line 3- Token {type: KW_LOCAL; value: None}
Line 3- Token {type: IDENTIFIER; value: x}
Line 3- Token {type: SINGLE_EQUAL; value: None}
Line 3- Token {type: STRING; value: Hello}
Line 4- Token {type: KW_DO; value: None}
Line 8- Token {type: KW_LOCAL; value: None}
Line 8- Token {type: IDENTIFIER; value: x}
Line 8- Token {type: SINGLE_EQUAL; value: None}
Line 8- Token {type: NUMBER; value: 4}
Line 9- Token {type: IDENTIFIER; value: y}
Line 9- Token {type: SINGLE_EQUAL; value: None}
Line 9- Token {type: MINUS; value: None}
Line 9- Token {type: NUMBER; value: 3}
Line 9- Token {type: PLUS; value: None}
Line 9- Token {type: IDENTIFIER; value: x}
Line 10- Token {type: IDENTIFIER; value: print}
Line 10- Token {type: PAR_OPEN; value: None}
Line 10- Token {type: IDENTIFIER; value: x}
Line 10- Token {type: DOUBLE_EQUAL; value: None}
Line 10- Token {type: IDENTIFIER; value: y}
Line 10- Token {type: PAR_CLOSE; value: None}
Line 11- Token {type: KW_END; value: None}
Line 13- Token {type: EOS; value: None}

```

Figura 2: Saída do arquivo *sample1.lua*

adição facilita o trabalho posterior do *parser* do interpretador, que será implementado em C++.

As Figuras 3 e 4 apresentam, respectivamente, o código-fonte e a saída do segundo arquivo de exemplo.

```

1 -- Arquivo sample2.lua
2
3 function multiplicar_por_2000(n)
4     dois_mil = 2e3
5     for i = 1, 19 do
6         b = "Esse loop só existe\nPara fins didáticos"
7         len_b = #b
8         print (len_b)
9     end
10    --[[ Comentário
11
12    de múltiplas
13
14    linhas ]]
15    return dois_mil * n
16 end

```

Figura 3: Código do arquivo *sample2.lua*

Este exemplo reforça as observações já feitas em relação ao primeiro arquivo. São dignos de nota, porém, o uso da notação para expoente em números, a presença do caracter de controle "\n" na *string* e a escrita de um comentário em múltiplas linhas.

```

Line 3- Token {type: KW_FUNCTION; value: None}
Line 3- Token {type: IDENTIFIER; value: multiplicar_por_2000}
Line 3- Token {type: PAR_OPEN; value: None}
Line 3- Token {type: IDENTIFIER; value: n}
Line 3- Token {type: PAR_CLOSE; value: None}
Line 4- Token {type: IDENTIFIER; value: mil}
Line 4- Token {type: SINGLE_EQUAL; value: None}
Line 4- Token {type: NUMBER; value: 2000}
Line 5- Token {type: KW_FOR; value: None}
Line 5- Token {type: IDENTIFIER; value: i}
Line 5- Token {type: SINGLE_EQUAL; value: None}
Line 5- Token {type: NUMBER; value: 1}
Line 5- Token {type: COMMA; value: None}
Line 5- Token {type: NUMBER; value: 19}
Line 5- Token {type: KW_DO; value: None}
Line 6- Token {type: IDENTIFIER; value: b}
Line 6- Token {type: SINGLE_EQUAL; value: None}
Line 6- Token {type: STRING; value: Esse loop só existe
Para fins didáticos}
Line 7- Token {type: IDENTIFIER; value: len_b}
Line 7- Token {type: SINGLE_EQUAL; value: None}
Line 7- Token {type: HASH; value: None}
Line 7- Token {type: IDENTIFIER; value: b}
Line 8- Token {type: IDENTIFIER; value: print}
Line 8- Token {type: PAR_OPEN; value: None}
Line 8- Token {type: IDENTIFIER; value: len_b}
Line 8- Token {type: PAR_CLOSE; value: None}
Line 9- Token {type: KW_END; value: None}
Line 15- Token {type: KW_RETURN; value: None}
Line 15- Token {type: IDENTIFIER; value: mil}
Line 15- Token {type: STAR; value: None}
Line 15- Token {type: IDENTIFIER; value: n}
Line 16- Token {type: KW_END; value: None}
Line 17- Token {type: EOS; value: None}

```

Figura 4: Saída do arquivo *sample2.lua*

O terceiro arquivo é apresentado na Figura 5. Nota-se que esse código não corresponde a um programa válido dentro da sintaxe de Lua; uma tentativa de execução, por um interpretador completo, resultaria em um erro de *parser*. Ainda assim, o programa é lexicamente correto, de modo que é uma entrada válida para o analisador léxico aqui apresentado. A saída corresponde é mostrada na figura 6.

```

1 -- Strings
2 "String com \n caractere de controle."
3 'Outra string \n mas com apóstrofo.'
4 [[Aqui, \n é interpretado literalmente]]
5 -- Números
6 232.5
7 3e-2
8 -4
9 -- Palavras chave e identificadores
10 local
11 LOCAL
12 locale
13 alocal
14 -- Símbolos
15 = == ~= ~
16 // /
17 >>>

```

Figura 5: Código do arquivo *sample3.lua*

```

Line 2- Token {type: STRING; value: String com
caracter de controle.}
Line 3- Token {type: STRING; value: Outra string
mas com apóstrofo.}
Line 4- Token {type: STRING; value: Aqui, \n é interpretado literalmente}
Line 6- Token {type: NUMBER; value: 232.5}
Line 7- Token {type: NUMBER; value: 0.03}
Line 8- Token {type: MINUS; value: None}
Line 8- Token {type: NUMBER; value: 4}
Line 10- Token {type: KW_LOCAL; value: None}
Line 11- Token {type: IDENTIFIER; value: LOCAL}
Line 12- Token {type: IDENTIFIER; value: locale}
Line 13- Token {type: IDENTIFIER; value: alocal}
Line 15- Token {type: SINGLE_EQUAL; value: None}
Line 15- Token {type: DOUBLE_EQUAL; value: None}
Line 15- Token {type: NOT_EQUAL; value: None}
Line 15- Token {type: TIL; value: None}
Line 16- Token {type: DOUBLE_SLASH; value: None}
Line 16- Token {type: SLASH; value: None}
Line 17- Token {type: DOUBLE_GREATER; value: None}
Line 17- Token {type: GREATER; value: None}
Line 18- Token {type: EOS; value: None}

Scanning of 'sample_lua_source/sample3.lua' sucessfully completed.

```

Figura 6: Saída do arquivo *sample3.lua*

O início do código mostra exemplos referentes às três formas de delimitar *strings*. Nota-se que, como explicitado na especificação da linguagem Lua, a delimitação com *long brackets* não interpreta sequências de caracteres de controle como “\n”.

Em seguida, são apresentados três exemplos de números. O primeiro introduz o uso do caracter “.” para indicar parte fracionária.<sup>4</sup> O segundo usa a notação para expoente, que já foi mencionada no segundo exemplo. O terceiro mostra que números negativos não são implementados diretamente no analisador léxico. Na verdade, é escaneada a sequência de um símbolo *MINUS* seguido do módulo do número. Posteriormente, o *parser*, a ser implementado em C++, reconhecerá esta construção como uma operação unária.

No próximo trecho, observa-se que o analisador léxico consegue distinguir corretamente palavras-chave de identificadores. É digno de nota que a linguagem Lua é *case-sensitive* inclusive para palavras-chave.

A última parte do código mostra como o programa diferencia sequências especiais de símbolos, mesmo quando estas apresentam caracteres em comum. A preferência é sempre pelo escaneamento da maior sequência. Este fato fica bem explicitado nos resultados da linha 17, na qual o analisador reconheceu um *token DOUBLE\_GREATER* (>>) seguido de um token *GREATER* (>).

<sup>4</sup>Em Lua, o tipo *number* é usado tanto para números inteiros quanto para números em ponto flutuante.



## 3.2 Erros

Discutimos aqui alguns erros tratados pelo analisador léxico.

Primeiramente, o programa pode não conseguir ler o arquivo especificado. Algumas razões para isso são: a inexistência do mesmo, a falta de permissões para abertura, erro na leitura etc. Em todo caso, o programa exibe um *Fatal Error* e uma mensagem apropriada. A ausência de um arquivo de entrada na chamada do programa também incutirá em um *Fatal Error*, seguido de uma mensagem indicando o uso correto do programa.

Caso o código-fonte contenha um lexema inválido na linguagem, uma mensagem é impressa na tela e a execução do analisador léxico é encerrada. Um exemplo dessa ocorrência é apresentado na Figura. Por fim, uma mensagem de erro mais específica ocorre caso o usuário tente escrever uma *string* ao longo de múltiplas linhas.

## 4 Conclusão

Como vimos nas seções anteriores, o programa foi implementado com sucesso e atende satisfatoriamente ao especificado no Relatório I, funcionando como um robusto *lexer* para a linguagem Lua. Contudo, há possibilidade de melhora do programa, por meio da implementação de algumas especificidades que foram deixadas de lado, como as discutidas na Seção 2. Esperamos implementar, no Trabalho III, as seções restantes do interpretador para o subconjunto de Lua.

## Referências

- [1] CAMPISTA, M. Slides de aula: Linguagens de programação.
- [2] IERUSALIMSKY, R., DE FIGUEIREDO, L. H., AND CELES, W. Lua 5.3 reference manual, 2015.
- [3] NYSTROM, B. *Crafting Interpreters*. 2015.
- [4] SKUD, K. Perl 5 version 30.0 documentation: Perl introduction.