

# Analizador Léxico para a Linguagem C- utilizando Máquina de Moore

Relatório - Projeto de Implementação

**Reginaldo Gregório de Souza Neto - 2252813**

Universidade Tecnológica Federal do Paraná (UTFPR)  
Campo Mourão, PR – Brasil

[reginaldoneto@alunos.utfpr.edu.br](mailto:reginaldoneto@alunos.utfpr.edu.br)

**Abstract.** *This report describes the methodology of creation and execution of an algorithm made in Python language to represent a C-language lexical analyzer, applying state machine concepts to identify special tokens in text files.*

**Resumo.** *Este relatório descreve a metodologia de criação e execução de um algoritmo feito na linguagem Python para representar um analisador léxico da linguagem C-, aplicando os conceitos de máquina de estados para a identificação de tokens especiais em arquivos de texto.*

## 1. Informações Gerais

Este trabalho se trata da criação de um analisador léxico para a linguagem C- apresentada pelo professor no decorrer da disciplina. Para tal, foram estipulados algumas palavras reservadas da linguagem C- que representam tokens especiais a serem reconhecidos pelo analisador. Assim como retrata a figura a seguir:

<code>int</code>	INT
<code>return</code>	RETURN
<code>void</code>	VOID
<code>while</code>	WHILE
<code>+</code>	PLUS
<code>-</code>	MINUS
<code>*</code>	TIMES
<code>/</code>	DIVIDE
<code>&lt;</code>	LESS
<code>&lt;=</code>	LESS_EQUAL
<code>&gt;</code>	GREATER
<code>&gt;=</code>	GREATER_EQUAL
<code>==</code>	EQUALS
<code>!=</code>	DIFFERENT
<code>(</code>	LPAREN
<code>)</code>	RPAREN
<code>[</code>	LBRACKETS
<code>]</code>	RBRACKETS
<code>{</code>	LBRACES
<code>}</code>	RBRACES

**Figura 1. Lista de tokens**

Em um primeiro momento, foi cogitada a possibilidade de programar o analisador na linguagem C, porém, ao perceber que a linguagem Python se trata de uma linguagem de mais alto nível e que proporciona maior facilidade para trabalhar com strings sua escolha se tornou praticamente inevitável.

## 2. Tokens e números

Para a realização da implementação do algoritmo foi idealizada uma máquina de estados que fosse capaz de reconhecer certos padrões na linguagem. Para tal, foi criado um dicionário em python, que é uma espécie de banco de palavras, que armazena todas as palavras e símbolos reservados e seus respectivos tokens. Deste modo se torna mais simples e rápida a identificação de termos especiais.

Em seguida, também foi criada uma lista com todos os números naturais para que, deste modo, se o caractere de entrada for um número, o analisador possa reconhecer e exibir seu respectivo token. Entretanto, há um caso especial que ocorre quando existe uma variável que possua um número em seu nome, como por exemplo “var2”, neste caso o analisador deve reconhecê-la como um ID e não como um número.

Para que isso ocorra de maneira correta, foi criado um buffer que serve para armazenar todos os caracteres de entrada até que se encontre um espaço, ou outro caráter especial. Caso o analisador encontre um número e o buffer tenha conteúdo, então se trata de uma variável, caso contrário é um número.

```
numeros = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
palavra = ""
banco = {}
    "if": "IF",
    "else": "ELSE",
    "int": "INT",
    "return": "RETURN",
    "void": "VOID",
    "while": "WHILE",
    "+": "PLUS",
    "-": "MINUS",
    "*": "TIMES",
    "/": "DIVIDE",
    "<": "LESS",
    "<=": "LESS_EQUAL",
    ">": "GREATER",
    ">=": "GREATER_EQUAL",
    "==" : "EQUALS",
    "!=" : "DIFFERENT",
    "(": "LPAREN",
    ")": "RPAREN",
    "[": "LBRACKETS",
    "]": "RBRACKETS",
    "{": "LBRACES",
    "}": "RBRACES",
    "=": "ATTRIBUTION",
    ";": "SEMICOLON",
    ",": "COMMA",
```

Figura 2. Lista de números e dicionário de palavras reservadas

### 3. Comparadores lógicos

No caso dos comparadores lógicos é preciso que o analisador reconheça se há algum símbolo que possa ser composto como `<=` ou `!=` entre outros. Para suprir esses casos o algoritmo aguarda a próxima iteração de caractere para ter certeza de que se tratava de um símbolo único presente no dicionário, ou um símbolo composto que também está presente no dicionário.

O uso de uma flag foi primordial para que o autômato reconheça a passagem do estado de símbolo único para composto, assim como quando o próximo caractere não complete uma combinação, neste caso ele armazena qual foi o primeiro símbolo lido e identifica seu token individualmente.

```
elif(flag == 1):
    if(letra == "="):
        esp += letra
        proviosria = banco.get(esp)
        print(proviosria)
        esp = ''
        flag = 0
    elif(letra == '!' or letra == '=' or letra == '>' or letra == '<'):
        flag = 1
        esp += letra
    elif(letra != " "): # LETRA OU CARACTER ESPECIAL
        token = banco.get(letra)
```

Figura 3. Comparações para símbolos especiais

### 4. Buffer

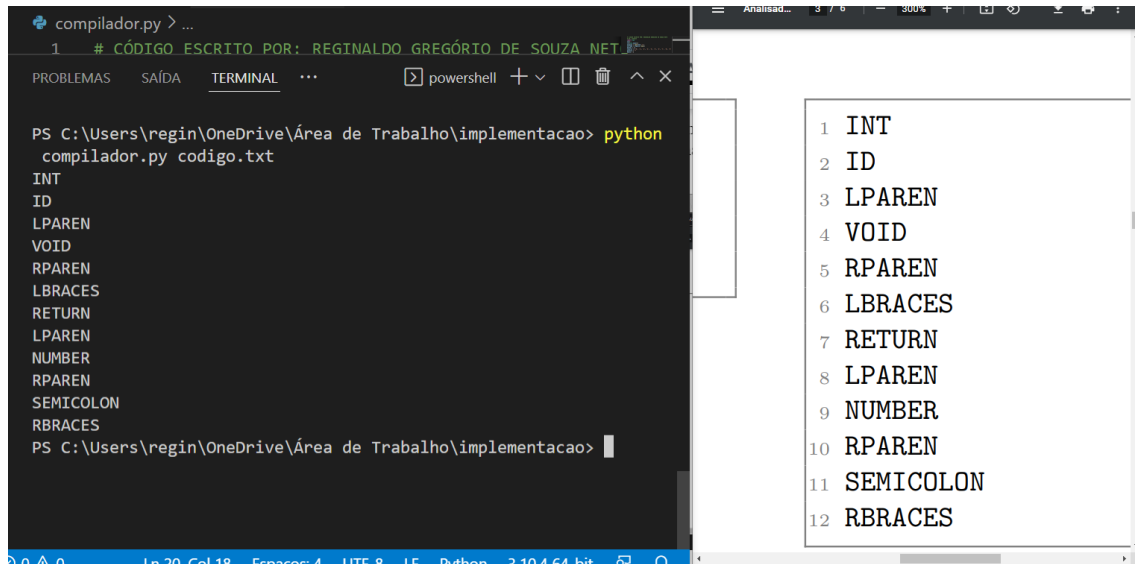
O conceito de buffer foi amplamente utilizado neste código, pois facilita a manipulação da entrada do algoritmo. A principal ideia para identificação das expressões contidas no arquivo foi abertura do mesmo através do seu nome passado por parâmetro na execução do código.

Por meio de uma estrutura de repetição, foi possível ler caractere por caractere do arquivo, inclusive os espaçamentos, deste modo, o analisador (autômato) foi capaz de receber entradas (cada caractere por iteração), comparando e identificando símbolos especiais, caso contrário ele adiciona no buffer cada letra para formar uma palavra reservada ou ID.

### 5. Execução

Para executar o código do analisador, é preciso que o arquivo de entrada esteja no mesmo diretório que o código fonte em python. Em seguida digite o comando no cmd: *“python compilador.py codigo.txt”* em que o primeiro argumento se trata de qual compilador irá atuar sobre o código, o segundo argumento se trata do nome do arquivo de código fonte (que está contido o analisador) e o terceiro e último argumento se trata do nome do arquivo que servirá de entrada para o autômato junto de sua extensão.

Vale ressaltar que toda a programação, execução e testes foram realizados em um computador com o Windows 10. Por conta disso, é possível que haja alguns problemas de execução através do terminal do linux.



The image shows a Windows 10 desktop environment. On the left, a PowerShell terminal window titled 'compilador.py > ...' displays the command 'python compilador.py codigo.txt' and its output, which lists 12 tokens: INT, ID, LPAREN, VOID, RPAREN, LBRACES, RETURN, LPAREN, NUMBER, RPAREN, SEMICOLON, and RBRACES. On the right, a text editor window titled 'Analisad...' shows the same list of tokens, each preceded by a line number from 1 to 12.

```
1 INT
2 ID
3 LPAREN
4 VOID
5 RPAREN
6 LBRACES
7 RETURN
8 LPAREN
9 NUMBER
10 RPAREN
11 SEMICOLON
12 RBRACES
```

Figure 4. Caso de teste disponibilizado no enunciado do trabalho

## 6. Conclusão

Por fim, podemos concluir que é possível criar um analisador léxico utilizando os conceitos aprendidos na disciplina de Linguagens Formais Autômatos e Computabilidade em conjunto com a aplicação de lógica de programação e outras disciplinas do curso.

Nota-se também que em um código relativamente simples (pouco mais de 80 linhas), é possível cobrir uma quantidade razoável de casos e tokens de uma linguagem de programação. Portanto, é nítido que tais conhecimentos adquiridos e aplicados neste trabalho serão de suma importância para o desenvolvimento de outros trabalhos assim como outras disciplinas, como por exemplo a de Compiladores.