Analisador Léxico para a Linguagem C-: Projeto de Implementação utilizando Máquina de Moore

Lucas Dos Santos Vaz

¹Bacharelado em Ciências da Computção Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

This article presents the implementation of a lexical analyzer for the C- programming language using a Moore Machine. The lexical analyzer is responsible for reading source code and generating a list of tokens, which are essential for the subsequent phases of compilation. The project was developed in Python, utilizing the automata-lib library to simulate the Moore Machine. The results demonstrate the effectiveness of the approach in recognizing tokens and handling errors in the C- language. This work contributes to the understanding of lexical analysis and automata theory in the context of compiler construction.

Resumo

Este artigo apresenta a implementação de um analisador léxico para a linguagem de programação C- utilizando uma Máquina de Moore. O analisador léxico tem como função ler o código-fonte e gerar uma lista de tokens, que são essenciais para as fases subsequentes da compilação. O projeto foi desenvolvido em Python, utilizando a biblioteca automata-lib para simular a Máquina de Moore. Os resultados demonstram a eficácia da abordagem no reconhecimento de tokens e no tratamento de erros na linguagem C-. Este trabalho contribui para o entendimento da análise léxica e da teoria de autômatos no contexto da construção de compiladores.

1 Introdução

O objetivo deste trabalho é projetar e implementar um autômato com saída, do tipo Máquina de Moore, que funcione como um Analisador Léxico para a linguagem C-. Para compreender a relevância deste projeto, é essencial entender o funcionamento de uma Máquina de Moore, o conceito de Analisador Léxico e as características da linguagem C-.

A Máquina de Moore é um tipo de autômato finito determinístico (AFD) em que as saídas dependem exclusivamente do estado atual. Em outras palavras, cada estado está associado a uma saída fixa, e a transição entre os estados é determinada pela entrada recebida. Este modelo é amplamente utilizado em sistemas digitais e de computação devido à sua previsibilidade e facilidade de implementação. No contexto de um analisador léxico, a Máquina de Moore permite identificar padrões léxicos a partir de sequências de entrada, gerando saídas correspondentes a cada token reconhecido.

A linguagem C- é uma versão simplificada da linguagem de programação C, projetada geralmente para fins educacionais e de pesquisa em análise de compiladores. Ela preserva muitas das estruturas fundamentais do C, como expressões aritméticas, controle de fluxo e declaração de variáveis,

mas com um conjunto reduzido de recursos para facilitar a implementação de ferramentas de análise e compilação.

Um Analisador Léxico é a primeira fase de um compilador, responsável por examinar o códigofonte e dividir a entrada em unidades léxicas chamadas tokens. Cada token representa uma categoria sintática, como identificadores, palavras-chave, operadores ou delimitadores. A principal função do analisador léxico é simplificar a entrada para as etapas posteriores do processo de compilação, removendo espaços em branco e reconhecendo padrões predefinidos.

Neste trabalho, a implementação do Analisador Léxico utilizando uma Máquina de Moore para a linguagem C- visa demonstrar, na prática, como os conceitos teóricos de autômatos finitos podem ser aplicados na análise de léxica de linguagens de programação, proporcionando uma compreensão mais aprofundada sobre a interação entre a teoria dos autômatos e a compilação.

2 Metodologia

Para o desenvolvimento deste projeto, inicialmente utilizei o JFlap 7.1 para modelar o autômato de forma visual. O JFlap é uma ferramenta educacional amplamente utilizada no estudo de autômatos, gramáticas formais e linguagens formais. Ele permite a criação, simulação e análise de autômatos finitos, autômatos de pilha, autômatos de Turing, entre outros. Com ele, foi possível validar a estrutura e o comportamento do autômato antes da implementação do analisador léxico, garantindo que as transições entre os estados estavam corretas e que o autômato poderia reconhecer corretamente os padrões da linguagem C-.

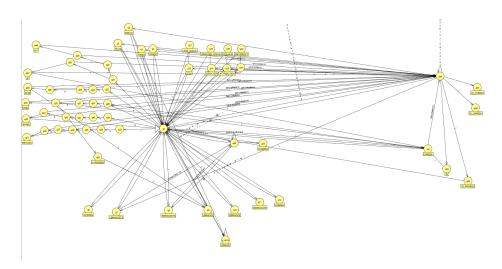


Figura 1: A Figura acima ilustra o autômato criado no JFLAP

A segunda etapa do desenvolvimento consistiu na implementação do analisador léxico utilizando a linguagem Python. Python foi escolhido devido à sua simplicidade, ampla comunidade de desenvolvedores e pela disponibilidade de bibliotecas especializadas. Em particular, utilizamos a biblioteca automata, que fornece suporte para a manipulação e simulação de autômatos finitos. A classe Moore da biblioteca automata.fa.Moore foi essencial para a implementação do autômato de Moore, facilitando a definição dos estados, transições e saídas.

Durante o desenvolvimento do analisador léxico, realizei diversos testes com diferentes entradas para garantir que os tokens eram corretamente reconhecidos e classificados. Implementamos rotinas para lidar com erros léxicos, como identificação de caracteres inválidos e tratamento de

Lucas Dos Santos Vaz

sequências malformadas.

3 Implementação

Para a implementação do autômato, utilizamos a linguagem Python devido à sua flexibilidade e à vasta disponibilidade de bibliotecas para manipulação de autômatos. O autômato foi implementado com a biblioteca automata-lib, que oferece suporte para a criação e manipulação de autômatos finitos, incluindo autômatos de Moore. A classe Moore, presente na biblioteca, permite definir estados, transições e saídas associadas de forma estruturada, facilitando a modelagem do analisador léxico.

A automata-lib é uma biblioteca projetada para auxiliar no estudo e na implementação de autômatos finitos, fornecendo ferramentas para modelagem, simulação e análise dessas estruturas. No contexto deste projeto, ela se encaixa perfeitamente ao permitir a representação formal do autômato de Moore, simplificando a implementação do analisador léxico e garantindo maior precisão na identificação de tokens.

A implementação está baseada na definição de estados, alfabeto de entrada e saídas correspondentes, conforme detalhado a seguir:

- Estados: Representam os diferentes estágios do processamento do analisador léxico. Cada estado é identificado por um nome único, como (q0), (q1), ..., (q61). além do estado especial (q_error), que indica um erro léxico quando um padrão inválido é encontrado. O estado inicial é q0, e os estados finais estão associados aos diferentes tokens reconhecidos pelo analisador.
- Alfabeto de entrada: Conjunto de caracteres reconhecidos pelo autômato.
 - Letras: a-z, utilizadas para formar identificadores e palavras-chave.
 - Dígitos: 0-9, usados na construção de números.
 - Operadores: +, -, *, /, empregados em expressões aritméticas.
 - Delimitadores: (), , [], ,, ;, =, usados para estruturar a sintaxe da linguagem.
 - Espaços e caracteres de controle: (espaço), \n (nova linha), \t (tabulação), utilizados para separar tokens e estruturar o código-fonte.
 - Caracteres especiais: Outros símbolos relevantes para a linguagem C-.
- Saídas (Tokens reconhecidos): Cada estado final do autômato está associado a um token específico da linguagem C-, como:
 - Palavras-chave: INT, IF, ELSE, VOID, RETURN, WHILE, FOR, FLOAT.
 - Operadores: PLUS, MINUS, TIMES, DIVIDE.
 - Operadores relacionais: LESS, LESS_EQUAL, GREATER_EQUAL, GREATER, EQUALS, DIFFERENT.
 - Delimitadores: LPAREN ((), RPAREN ()), LBRACKETS ([), RBRACKETS (]), LBRACES ().
 - Outros símbolos: ATTRIBUTION (=), SEMICOLON (;), COMMA (,).
 - Identificadores e números: ID, NUMBERS.

```
moore = Moore(['ge', 'q1', 'q2', 'q3', 'q4', 'q5', 'q6', 'q6', 'q6', 'q16', 'q12', 'q13', 'q33', 'q3
```

Figura 2: A figura é o trecho do código que realiza as definições dos estados, alfabeto e saídas do automato de Moore

A implementação do autômato de Moore envolve a definição dos estados, do alfabeto de entrada e das saídas, bem como a especificação das transições entre os estados. Cada estado do autômato representa uma fase do processamento de tokens no analisador léxico.

As transições do autômato são implementadas utilizando um dicionário, onde cada chave representa um estado e seu valor é outro dicionário contendo os possíveis caracteres de entrada e seus respectivos estados de destino. Essa abordagem permite uma organização clara e eficiente do autômato.

A biblioteca automata organiza essas informações estruturando os estados e transições de maneira eficiente. Durante a execução, a entrada é processada caractere por caractere, determinando a transição de estado de acordo com o símbolo atual. Caso a sequência reconhecida corresponda a um padrão válido, o estado final gerará o token correspondente.

```
{
  'q0': [[
  '!': 'q61',
  'x': 'q52',
  'y': 'q52',
  's': 'q52',
  '+': 'q31',
```

Figura 3: A figura é o um trecho das transições que o estado q0 realiza quando recebe caractes como (!,x,y,s,+)

Cada estado no autômato possui transições definidas para caracteres válidos. No entanto, quando um caractere inesperado é encontrado, a transição ocorre para o estado (q_error). Esse estado é utilizado para indicar que um erro léxico foi detectado, permitindo que o analisador identifique e trate essas situações de maneira apropriada.

Após a definição das transições, é necessário definir o estado inicial da Máquina de Moore. O estado inicial representa o ponto de partida do autômato e, no caso da implementação realizada, esse estado é denominado q0.

Lucas Dos Santos Vaz

Figura 4: A figura é o trecho do código que define o estado inicial (q0)

Caso um caractere ou sequência inválida seja encontrada, a execução é desviada para o estado (q_error), indicando um erro léxico. Esse estado é responsável por lidar com entradas não reconhecidas e fornecer feedback adequado ao usuário sobre a ocorrência de um erro.

O autômato possui 63 estados (q0 a q62), cada um representando uma etapa no reconhecimento de um token. O estado inicial é q0, e a partir dele, o autômato realiza transições com base nos caracteres lidos do código-fonte. Exemplo de transições:

- IF (q2): Reconhece a palavra-chave if.
- INT (q4): Reconhece a palavra-chave int.
- ELSE (q8): Reconhece a palavra-chave else.
- RETURN (q14): Reconhece a palavra-chave return.
- VOID (q18): Reconhece a palavra-chave void.
- WHILE (q23): Reconhece a palavra-chave while.
- FOR (q26): Reconhece a palavra-chave for.
- FLOAT (q30): Reconhece a palavra-chave float.
- PLUS (q31): Reconhece o operador +.
- MINUS (q32): Reconhece o operador -.
- TIMES (q33): Reconhece o operador *.
- DIVIDE (q34): Reconhece o operador /.
- LESS_EQUAL (q36): Reconhece o operador <=.
- GREATER_EQUAL (q38): Reconhece o operador >=.
- EQUALS (q39): Reconhece o operador ==.
- DIFFERENT (q40): Reconhece o operador !=.
- LESS (q58): Reconhece o operador <.
- **GREATER** (q59): Reconhece o operador >.
- ATTRIBUTION (q60): Reconhece o operador =.
- LPAREN (q41): Reconhece o delimitador (.
- RPAREN (q42): Reconhece o delimitador).
- LBRACES (q43): Reconhece o delimitador {.
- RBRACES (q44): Reconhece o delimitador }.
- LBRACKETS (q45): Reconhece o delimitador [.

- RBRACKETS (q46): Reconhece o delimitador].
- COMMA (q47): Reconhece o delimitador,.
- SEMICOLON (q48): Reconhece o delimitador;
- NUMBER (q50): Reconhece números.
- POINT (q51): Reconhece o caractere ..
- ID LPAREN (q53): Reconhece a sequência id(.
- ID RPAREN (q54): Reconhece a sequência id).
- ID COMMA (q55): Reconhece a sequência id,.
- ID SEMICOLON (q57): Reconhece a sequência id;.
- ERR-LEX-USE (q_error): Erro léxico.

A função principal para processar o arquivo é:

Figura 5: A figura é o trecho onde realizamos o processamento do arquivo caso nenhum erro seja encontrado

Aqui, o código abre o arquivo e lê cada linha, chamando o autômato para processá-la. O método moore.get_output_from_string(linha.strip()) analisa a linha e retorna a sequência de tokens identificados. Se a saída contém o estado (q_error), significa que um caractere inválido foi encontrado. O código então percorre a linha para identificar exatamente onde ocorreu o erro e retorna as posições e o caracter.

```
if 'q_error' in output:

# Encontra o caractere inválido
for i, char in enumerate(linha.strip(), start=1):

# Verifica se o caractere leva ao estado de erro
next_state = moore.transitions[moore.current_state].get(char, 'q_error')
if next_state == 'q_error':
    print(f"Erro[{num_linha:02}][{i:02}]: Caracter inválido: {char}")
    break
```

Figura 6: A acima mostra como é feito o tratamento dos erros de acordo com o que foi pedido no enunciado do projeto devolvendo a posição e o caracter que está causando erro.

O módulo principal do programa é responsável por gerenciar a execução do analisador léxico. Ele garante que o código seja executado corretamente quando o script for chamado diretamente pelo usuário. A variável args recebe a lista de argumentos da linha de comando, excluindo o nome do script (sys.argv[0]). Caso nenhum argumento seja fornecido, o programa exibe uma mensagem de erro e encerra a execução imediatamente Esse mecanismo evita que o analisador seja executado sem um arquivo de entrada válido, garantindo a integridade do processamento. Podemos a aplicação no trecho de código abaixo.

```
if __name__ == "__main__":
    args = sys.argv[1:] # Captura os argumentos (ignorando o nome do script)

# Verifica se há pelo menos um argumento
    if len(args) < 1:
        print("ERR-LEX-USE")
        sys.exit(1)</pre>
```

Figura 7: ativação do modulo principal do codigo.

O programa suporta um modo de execução especial identificado pela flag -k. Caso essa opção seja utilizada, o usuário deve fornecer obrigatoriamente o nome do arquivo a ser processado como segundo argumento. O tratamento dessa situação é feito da seguinte maneira:

Lucas Dos Santos Vaz 7

```
# Verifica se há uma flag -k
if args[0] == "-k":
    if len(args) < 2:
        print("ERR-LEX-USE")
        sys.exit(1)
        arquivo = args[1]
else:
    arquivo = args[0]</pre>
```

Figura 8: Tratamento da flag -k.

Se o primeiro argumento for -k", mas nenhum arquivo for especificado após a flag, o programa informa um erro (ERR-LEX-USE) e encerra a execução. Por outro lado, se a flag não for utilizada, o primeiro argumento fornecido é tratado diretamente como o nome do arquivo a ser analisado.

O analisador léxico foi projetado para processar exclusivamente arquivos com a extensão .cm. Para garantir que apenas arquivos desse tipo sejam utilizados, o seguinte bloco de código é executado:

```
# Verifica a extensão do arquivo
if not arquivo.endswith(".cm"):
    print("ERR-LEX-NOT-CM")
    sys.exit(1)
```

Figura 9: Tratamento da para verificação da extensão .cm.

Além da verificação da extensão, o programa também checa se o arquivo informado realmente existe no diretório especificado. Se o arquivo não for encontrado, o programa emite a mensagem de erro ERR-LEX-FILE-NOT-EXISTS e encerra sua execução, evitando que o analisador tente processar um arquivo inexistente.

Após passar por todas as verificações necessárias, o programa inicia a análise léxica do arquivo chamando a função principal de processamento. Essa chamada encaminha o arquivo validado para a função responsável pela leitura e análise do conteúdo, garantindo que apenas arquivos corretamente especificados e existentes sejam submetidos ao processamento pelo autômato de Moore.

```
# Verifica se o arquivo existe
if not os.path.exists(arquivo):
    print("ERR-LEX-FILE-NOT-EXISTS")
    sys.exit(1)

# Processa o arquivo
processar_arquivo(arquivo)
```

Figura 10: Tratamento da para verificação da extensão .cm.

Durante o desenvolvimento tive que fazer algumas alterações para corrijir erros de como as

quebras de linha são tratadas no Windows no arquivo de teste. O código original comparava diretamente as saídas após remover espaços extras (strip()), mas não tratava possíveis diferenças nas quebras de linha entre sistemas operacionais (Windows usa \r\n, enquanto Linux e macOS usam \n). A versão modificada substitui \r\n por \n para garantir que a comparação funcione corretamente em diferentes ambientes.

```
# Normaliza as quebras de linha antes de comparar
generated_output = stdout.decode("utf-8").strip().replace("\r\n", "\n").replace("\r", "\n")
expected_output_normalized = expected_output.strip().replace("\r\n", "\n").replace("\r", "\n")
# Compara as saidas normalizadas
assert generated_output == expected_output_normalized
```

Figura 11: Tratamento da para verificação das quebreas de linha no Windows.