Matheus Floriano Saito Da Silva

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

Relatório técnico de atividade prática solicitado pelo professor Rogerio Aparecido Gonçalves na disciplina de Teoria da computação do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR

Departamento Acadêmico de Computação – DACOM

Bacharelado em Ciência da Computação – BCC

Campo Mourão

Fevereiro / 2025

Resumo

Este relatório apresenta os desenvolvimentos e resultados da atividade prática realizada, cujo objetivo foi projetar e implementar um analisador léxico para a linguagem C- por meio de uma máquina de Moore e linguagem de programação (Python).

Palavras-chave: Moore. Python. analisador léxico. implementar.

Sumário

1 Introdução e objetivos

Neste relatório será descrito o trabalho 1 da disciplina de Teoria da computação, o qual visa construir um analisador léxico para a linguagem C- criado por meio de uma máquina de Moore e a linguagem Python. Este analisador léxico recebe por entrada um arquivo contendo um código C- e processa este código devolvendo tokens das operações.

2 Fundamentação

Para a realização deste trabalho foi utilizado como base as instruções, o autômato inicial e o código base inicial fornecido pelo professor, além do capitulo 2 do livro do (LOUDEN, 2004).

3 Materiais

Foi utilizado um Notebook Dell G15 com as especificações:

- Intel Core i5-10500H (@2.50GHz e 12 *thread*)
- 16GB RAM
- SSD NVMe ADATA 512GB
- Pop!_OS 22.04 LTS

  E ferramentas :

- JFLAP Versão 7.1
- Python

4 Projeto do Autômato

Foi construído e utilizado como base o seguinte autômato:

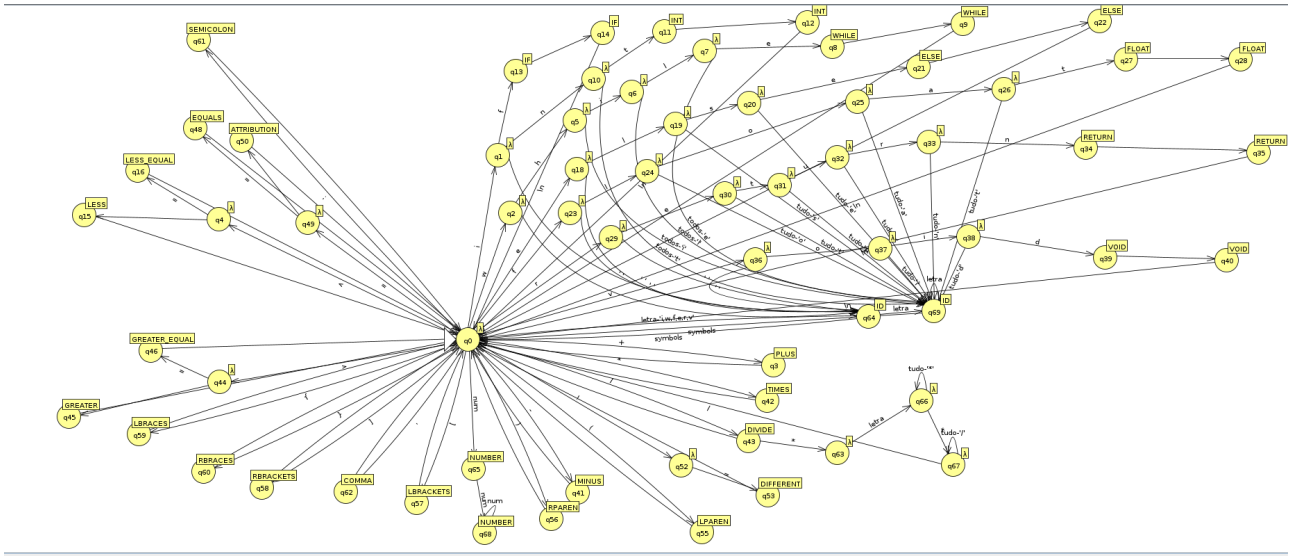

Figura 1 – Autômato Base

5 Descrição do funcionamento do Código

Esta seção está dividida em várias partes, cada parte contém um pedaço do programa final e uma devida explicação de como foi desenvolvida.

5.1 Máquina de Moore

```
moore = Moore(
states=['q0', 'q1', 'q2', 'q3', 'q4', 'q5', 'q6', 'q7', 'q8', 'q9', 'q10', 'q11', 'q12', 'q13', 'q14',
↪  'q15', 'q16', 'q17', 'q18', 'q19', 'q20', 'q21', 'q22', 'q23',
        'q24', 'q25', 'q26', 'q27', 'q28', 'q29', 'q30', 'q31', 'q32', 'q33', 'q34', 'q35', 'q36',
        ↪  'q37', 'q38', 'q39', 'q40', 'q41', 'q42', 'q43', 'q44', 'q45', 'q46', 'q47', 'q48',
        'q49', 'q50', 'q51', 'q52', 'q53', 'q54', 'q55', 'q56', 'q57', 'q58', 'q59', 'q60', 'q61',
        ↪  'q62', 'q63', 'q64', 'q65', 'qID_START', 'qID_CONT', 'qNUM_START',
        ↪  'qNUM_CONT','qCOMS','qCOMB','qCOME'],

input_alphabet=list(string.ascii_letters) + list(string.digits) + ['+','!', '-', '*', '/', '<', '>',
↪  '=', '(', ')', '[', ']', '{', '}', ';', ',', '\n', ' '],

output_alphabet=['INT', 'ELSE', 'IF', 'WHILE', 'FLOAT', 'RETURN', 'VOID', 'MINUS', 'PLUS', 'TIMES',
↪  'DIVIDE', 'DIFFERENT', 'LPAREN', 'RPAREN', 'NUMBER', 'ID',
                'LBRACKETS', 'RBRACKETS', 'COMMA', 'LBRACES', 'RBRACES', 'GREATER', 'GREATER_EQUAL',
                ↪  'LESS', 'LESS_EQUAL', 'EQUALS', 'SEMICOLON', 'ATTRIBUTION'],

transitions={
    'q0': {
        'w': 'q2', 'i': 'q1', 'e': 'q18', 'f': 'q23', 'r': 'q29', 'v': 'q36', '-': 'q41', '+': 'q3',
        ↪  '*': 'q42', '/': 'q43',
        '!': 'q52', '(': 'q55', ')': 'q56', '[': 'q57', ']': 'q58', '{': 'q59', '}': 'q60', '<': 'q4',
        ↪  '>': 'q44', '=': 'q49',
        ';': 'q61', ',': 'q62', ' ': 'q0', '\n': 'q0',

        **{c: 'qID_START' for c in string.ascii_letters if c not in {'w', 'i', 'e', 'f', 'r', 'v'}},
```

```
    **{c: 'qNUM_START' for c in string.digits}
},

'qID_START': {
    **{c: 'qID_CONT' for c in string.ascii_letters + string.digits},  # Começa um ID
    **{c: 'q0' for c in [' ', '\n', '+', '-', '*', '/', '<', '>', '=', '(', ')', '[', ']', '{',
    ↪  '}', ';', ',']}  # Termina o ID
},

'qID_CONT': {
    **{c: 'qID_CONT' for c in string.ascii_letters + string.digits},  # Continua como ID
    **{c: 'q0' for c in [' ', '\n', '+', '-', '*', '/', '<', '>', '=', '(', ')', '[', ']', '{',
    ↪  '}', ';', ',']}  # Termina o ID
},

'qCOMS': {c: 'qCOMB' for c in string.printable},  # Consome caracteres dentro do comentário
'qCOMB': {'*': 'qCOME', **{c: 'qCOMB' for c in string.printable if c not in ["*"]}},  # Aguarda
↪  "*/"
'qCOME': {'/': 'q0', **{c: 'qCOMB' for c in string.printable if c not in "/"}},  # Fecha comentário

'qNUM_START': {c: 'qNUM_CONT' for c in string.digits},  # Começa a ler número
'qNUM_CONT': {c: 'qNUM_CONT' for c in string.digits},  # Continua lendo número
'qNUM_CONT': {c: 'q0' for c in [' ', '\n', '+', '-', '*', '/', '<', '>', '=', '(', ')', '[', ']',
↪  '{', '}', ';', ',']},  # Número termina

'q1': {' ' : 'qID_CONT', 'n': 'q10', 'f': 'q13'},
'q2': {' ' : 'qID_CONT','h': 'q5'},
'q3': {' ': 'q0'},
'q4': {'=': 'q16', ' ': 'q15'},
'q5': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'i'},'i': 'q6'},
'q6': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'l'},'l': 'q7'},
'q7': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'e'},'e': 'q8'},
'q8': {' ': 'q9'},
'q9': {'\n': 'q0'},
'q10': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 't'},'t': 'q11'},
'q11': {' ': 'q12'},
'q12': {'\n': 'q0'},
'q13': {' ': 'q14'},
'q14': {'\n': 'q0'},
'q15': {'\n': 'q0'},
'q16': {' ': 'q0'},
'q18': {' ': 'qID_CONT','l': 'q19'},
'q19': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 's'},'s': 'q20'},
'q20': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'e'},'e': 'q21'},
'q21': {' ': 'q22'},
'q22': {'\n': 'q0'},
'q23': {' ' : 'qID_CONT','l': 'q24'},
'q24': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'o'},'o': 'q25'},
'q25': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'a'},'a': 'q26'},
'q26': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 't'},'t': 'q27'},
'q27': {' ': 'q28'},
'q28': {'\n': 'q0'},
'q29': {' ' : 'qID_CONT', 'e': 'q30'},
'q30': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 't'},'t': 'q31'},
'q31': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'u'},'u': 'q32'},
'q32': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'r'},'r': 'q33'},
'q33': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'n'},'n': 'q34'},
'q34': {' ': 'q35'},
'q35': {'\n': 'q0'},
'q36': {' ' : 'qID_CONT','o': 'q37'},
```

```
    'q37': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'i'},'i': 'q38'},
    'q38': {**{c: 'qID_CONT' for c in string.ascii_letters if c != 'd'},'d': 'q39'},
    'q39': {' ': 'q40'},
    'q40': {'\n': 'q0'},
    'q41' : {' ': 'q0'},
    'q42' : {' ': 'q0'},
    'q43' : {'*': 'qCOMS',' ': 'q0'},
    'q44' : {'=' : 'q46', ' ' : 'q45'},
    'q45' : {'\n': 'q0'},
    'q46' : {' ' : 'q0'},
    'q48' : {' ': 'q0'},
    'q49' : {'=' : 'q48', ' ' : 'q50'},
    'q50' : {'\n': 'q0'},
    'q52' : {'=' : 'q53'},
    'q53' : {' ': 'q0'},
    'q55' : {' ': 'q0'},
    'q56' : {' ': 'q0'},
    'q57' : {' ': 'q0'},
    'q58' : {' ': 'q0'},
    'q59' : {' ': 'q0'},
    'q60' : {' ': 'q0'},
    'q61' : {' ': 'q0'},
    'q62' : {' ': 'q0'},

},
initial_state='q0',
output_table={
    'q0': '',
    'q1': '',
    'q2': '',
    'q3': 'PLUS',
    'q4': '',
    'q5': '',
    'q6': '',
    'q7': '',
    'q8': 'WHILE',
    'q9': 'WHILE',
    'q10': '',
    'q11': 'INT',
    'q12': 'INT',
    'q13': '',
    'q14': 'IF',
    'q15': 'LESS',
    'q16': 'LESS_EQUAL',
    'q18': '',
    'q19': '',
    'q20': '',
    'q21': 'ELSE',
    'q22': 'ELSE',
    'q23': '',
    'q24': '',
    'q25': '',
    'q26': '',
    'q27': 'FLOAT',
    'q28': 'FLOAT',
    'q29': '',
    'q30': '',
    'q31': '',
    'q32': '',
    'q33': '',
```

```
        'q34': 'RETURN',
        'q35': 'RETURN',
        'q36': '',
        'q37': '',
        'q38': '',
        'q39': 'VOID',
        'q40': 'VOID',
        'q41': 'MINUS',
        'q42': 'TIMES',
        'q43': 'DIVIDE',
        'q44': '',
        'q45': 'GREATER',
        'q46': 'GREATER_EQUAL',
        'q48' : 'EQUALS',
        'q49': '',
        'q50': 'ATTRIBUTION',
        'q52': '',
        'q53': 'DIFFERENT',
        'q55': 'LPAREN',
        'q56': 'RPAREN',
        'q57': 'LBRACKETS',
        'q58': 'RBRACKETS',
        'q59': 'LBRACES',
        'q60': 'RBRACES',
        'q61': 'SEMICOLON',
        'q62': 'COMMA',
        'qID_START': 'ID',
        'qID_CONT': 'ID',
        'qNUM_START': 'NUMBER',
        'qNUM_CONT': 'NUMBER',
        'qCOMS' : '',
        'qCOMB' : '',
        'qCOME' : ''
    }
)
```

Esta é a representação direta da máquina de Moore do autômato apresentado adaptado ao código por meio da automata-python. O comportamento base é: todos os símbolos lidos retornam a respectiva saída, quando um caractere da tabela ascii não representa a inicial de uma palavra reservada ele é imediatamente considerado ID, quando o caractere faz parte de uma palavra reservada ele só se torna um ID se não completar a palavra reservada, dígitos são reconhecidos e quando detectam um símbolo retornam a q0, comentários são reconhecidos após ter um caractere "*"depois de "/"e param de ser reconhecidos quando é detectado "*"seguido de "/", vale ressaltar também que todas as transições retornam para q0 quando terminam de processar uma entrada.

5.2 Pré-processamento

```
def preprocess_input(input_string):
    formatted_input = ""
```

```
i = 0

while i < len(input_string):
    char = input_string[i]

    # Verifica se é um operador composto
    if i + 1 < len(input_string) and char in ['<', '>', '!', '=']:
        next_char = input_string[i + 1]
        if next_char == '=':
            formatted_input += f"\n{char}{next_char}\n"
            i += 2  # Pula os dois caracteres
            continue

    #verifica se e comentario
    #abertura de comentário
    if i + 1 < len(input_string) and char in ['/']:
        next_char = input_string[i + 1]
        if next_char == '*':
            formatted_input += f"\n{char}{next_char}\n"
            i += 2
            continue

    #fechamento de comentário
    if i + 1 < len(input_string) and char in ['*']:
        next_char = input_string[i + 1]
        if next_char == '/':
            formatted_input += f"\n{char}{next_char}\n"
            i += 2
            continue

    # Se for um delimitador isolado, adiciona espaçamento
    if char in ' (){};,+-*/<>=![]':
        formatted_input += f" \n{char} \n"
    elif char == ' ':
        formatted_input += ' '  # Mantém espaços simples
    elif char == '\n':
        formatted_input += '\n'  # Mantém novas linhas
    else:
        formatted_input += char  # Mantém o caractere

    i += 1  # Avança para o próximo caractere
```

```
    formatted_input += '\n'
    # print (formatted_input)
    return formatted_input.strip()
```

Esta função é responsável por tratar o código diretamente fornecido pelo arquivo, a ideia aqui é simplificar os dados passados para o autômato, neste caso o principal seria passar cada seção do código original em formato de: operação <quebra de linha> como por exemplo um "return(0);"seria transformado em :

return (0) ;

Esta função também foi utilizada para tratar um problema com operadores compostos, onde ao separar != por exemplo, teríamos um problema de identificação.

5.3 Processamento

```
def process_input(input_string):
    tokens = []
    current_state = moore.initial_state
    token = ""

    #formatando tokens

    for char in input_string:
        if char in moore.input_alphabet:
            next_state = moore.transitions[current_state].get(char, 'q0')

            if next_state == 'q0':  # Finalizou um token
                if moore.output_table.get(current_state):
                    tokens.append(moore.output_table[current_state])  # Acessa
                    ↪  e guarda o output da máquina

                    token = ""  # Reinicia o token
                    current_state = moore.initial_state
            else:
                token += char  # Continua construindo o token
                current_state = next_state
        else:
            error_handler.handle_error(f"Unexpected character: {char}")
            return tokens
```

```
    # Garante que o último token seja adicionado
    if moore.output_table.get(current_state):
        tokens.append(moore.output_table[current_state])
    # print(tokens)
    return tokens
```

Esta função é responsável por lidar com os tokens sem si, obtendo as saídas da máquina de Moore e as colocando em uma lista de strings onde cada string representa um token.

6 Testes e Resultados

6.1 Correção dos testes

Vale a pena ressaltar que alguns testes estavam errados, e um apenas foi ajustado para funcionar de acordo com a minha implementação como demonstrado a seguir:

Figura 2 – Teste 004

aqui a linha 32 descreve como INT quando na verdade é um ID.

Figura 3 – Teste 005

aqui faltou um RPAREN.

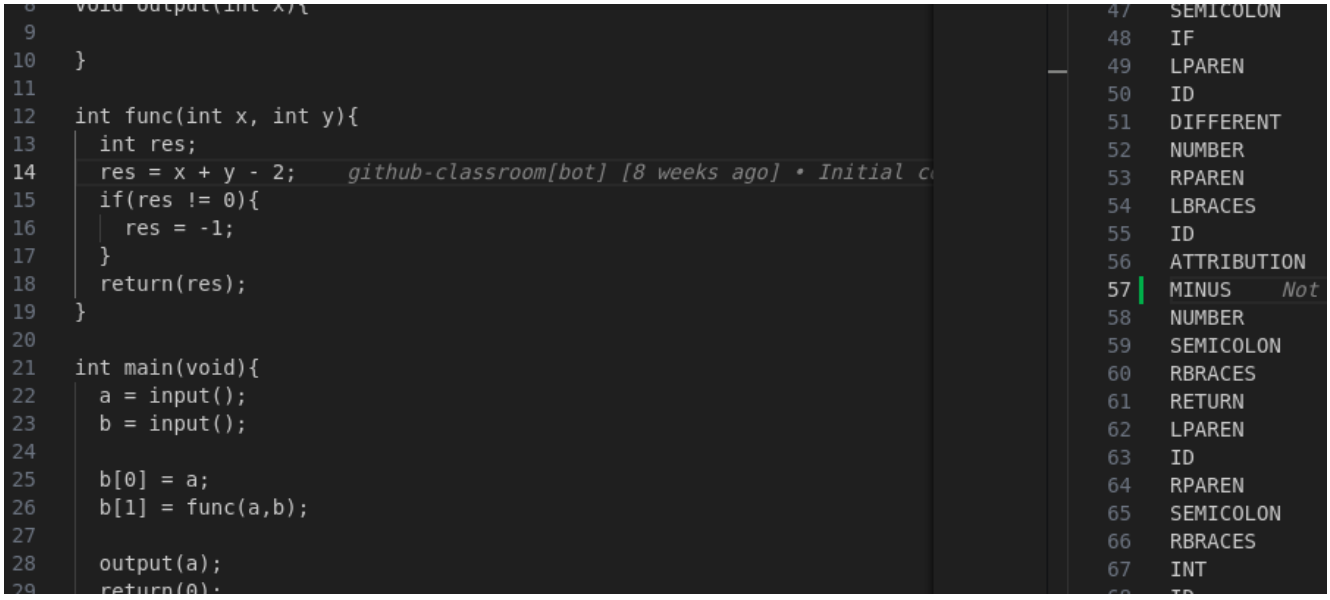

Figura 4 – Teste 006

aqui ocorre uma diferença entre implementações, no meu caso está sendo considerado MINUS, NUMBER, enquanto o teste original considera um número negativo como apenas NUMBER.

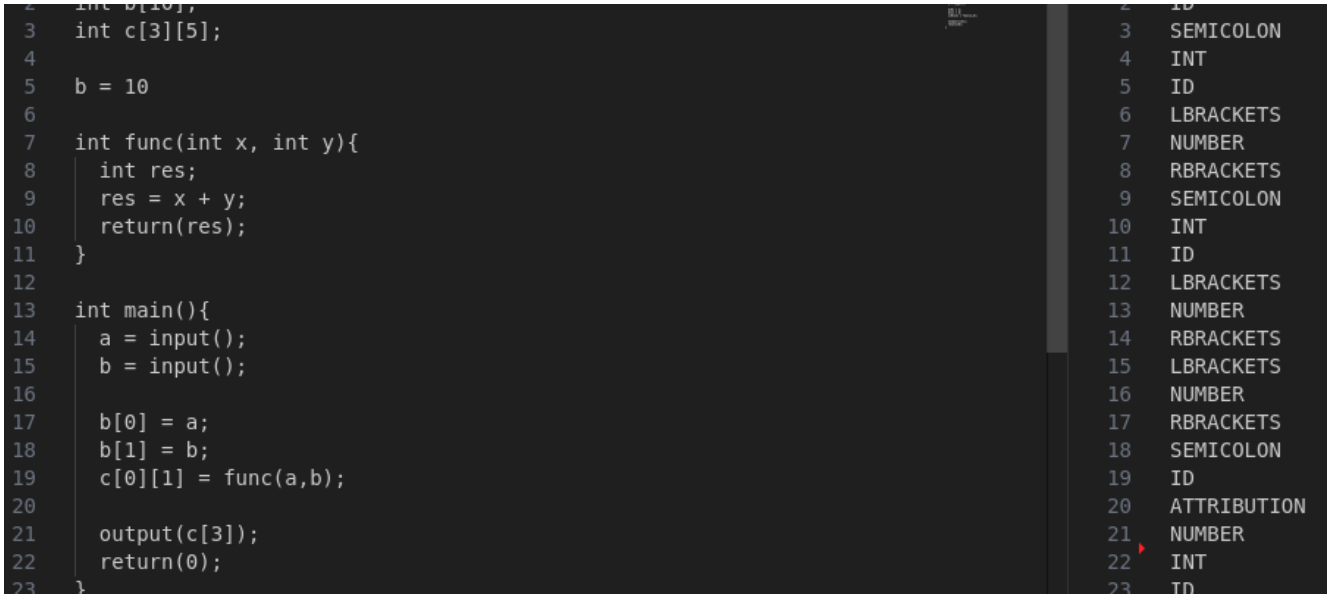

Figura 5 – Teste 007

aqui após o NUMBER tínhamos um SEMICOLON adicional após b = 10.

6.2 Teste Final e Meu teste

Após todas as mudanças nos testes como descrito na seção 6.1 o resultado do teste final:

Figura 6 – Testes

Também fiz um arquivo de teste que demonstra todas as funções do lexer e a comparação entre codigo e saída de tokens pode ser vista abaixo:

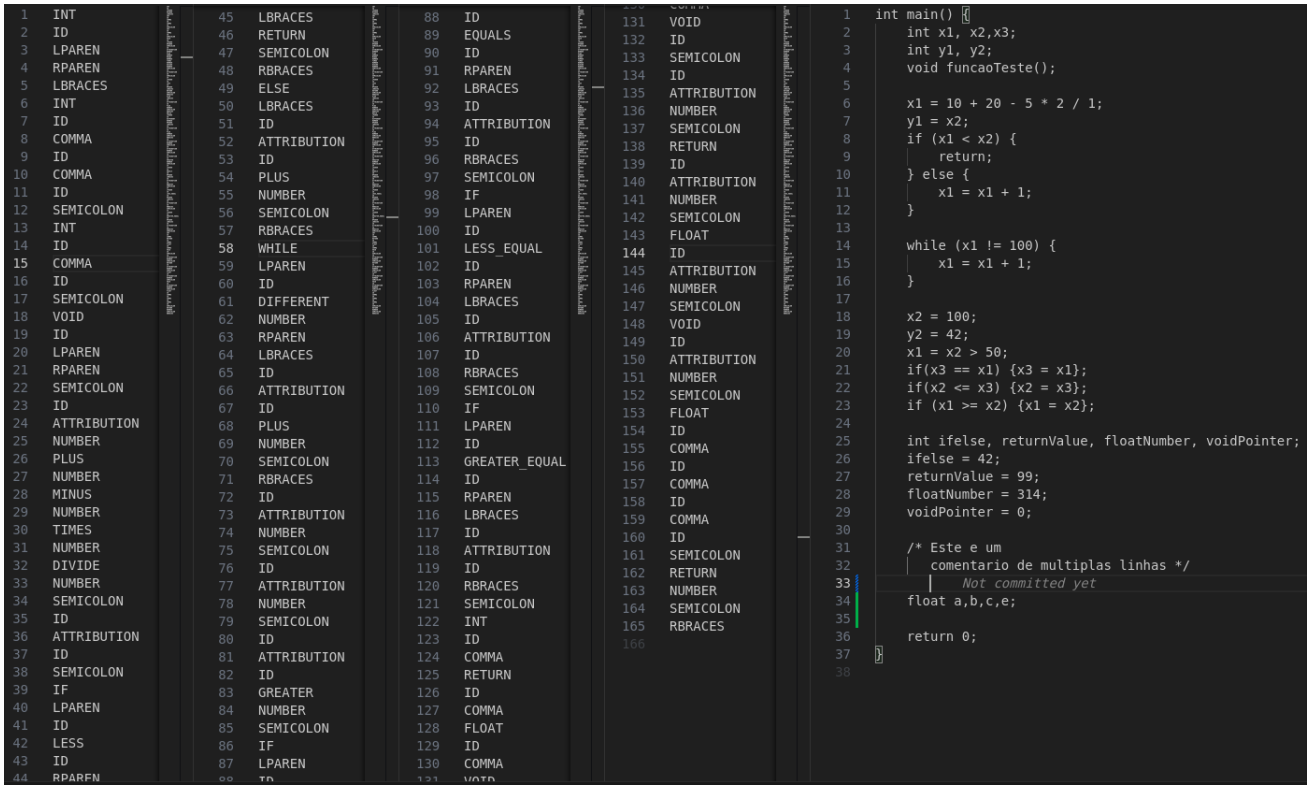

Figura 7 – MyTest

7 Conclusão

O projeto em si foi realizado com sucesso e os resultados finais foram favoráveis, durante a implementação do projeto foi possível obter mais conhecimento sobre um analisador léxico e sobre como lidar com autômatos em Python.

8 Referências

LOUDEN, K. C. *Compiladores: princípios e práticas*. 2004. [Accessed 27-01-2025]. Citado na página 4.