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

Paulo Henrique Pereira Da Silva[1]
[1]Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

This paper aims to present the implementation of an automaton with output, of Moore type, that works as a lexical analyzer of the C- programming language, utilizing the Python language.

Resumo

Este trabalho tem o objetivo de apresentar a implementação de um autômato com saída, do tipo Máquina de Moore, que funcione como um Analisador Léxico para a linguagem de programação C- utilizando a linguagem Python.

1 Introdução

A análise léxica desempenha um papel essencial no processo de compilação de programas de computador, convertendo o código-fonte em uma sequência de tokens significativos. Essa fase inicial envolve a identificação e classificação de lexemas, que são sequências de caracteres com significado léxico, como identificadores, palavras-chave, operadores e delimitadores.

Este artigo apresenta uma visão sobre a análise léxica da linguagem C-, uma versão simplificada da linguagem C, amplamente utilizada em cursos de computação e disciplinas introdutórias de programação. Dessa forma, a linguagem é bem conhecida no meio acadêmico. O artigo detalha o projeto e a implementação de um analisador léxico para C-, que exige a construção de um autômato capaz de reconhecer os tokens presentes em um código-fonte e, posteriormente, o desenvolvimento de um programa que transforma esse autômato em um código funcional para listar os tokens identificados.

Serão abordados os fundamentos da análise léxica, incluindo conceitos como lexemas, tokens e a estrutura da tabela de símbolos, além de uma introdução à linguagem C-, a representação do autômato utilizado, a implementação do código e exemplos de entradas e saídas geradas pelo analisador.

2 Desenvolvimento

A análise léxica é a etapa inicial do processamento de linguagens de programação, responsável por identificar e classificar os componentes básicos do código-fonte. Esse processo converte a sequência de caracteres em unidades significativas chamadas tokens, que representam elementos como identificadores, palavras-chave, números e símbolos.

Para realizar essa conversão, são utilizados conceitos como expressões regulares, que definem padrões para o reconhecimento de lexemas, e autômatos finitos determinísticos, modelos matemáticos que processam a entrada e determinam a classificação de cada lexema. Além disso, a

Corresponding author(s): Paulo Henrique Pereira Da Silva, Universidade Tecnológica Federal do Paraná, paulo.2004@alunos.utfpr.edu.br

tabela de símbolos armazena informações essenciais sobre identificadores, como tipos e escopos, auxiliando no processo de compilação.

Um aspecto importante da análise léxica é a detecção e manipulação de erros, garantindo que caracteres inválidos ou sequências desconhecidas sejam identificados e tratados adequadamente. Esse mecanismo é essencial para manter a integridade do analisador léxico e assegurar a correta interpretação do código.

2.1 Linguagem C-

A linguagem C- é uma versão simplificada da linguagem C, frequentemente utilizada em cursos de computação e disciplinas introdutórias de compiladores. Ela mantém a estrutura e a sintaxe básicas do C, mas elimina características mais complexas, tornando-se ideal para o ensino de conceitos fundamentais de análise léxica, sintática e semântica.

Ela foi projetada para facilitar a construção de compiladores e ferramentas de análise, incluindo elementos essenciais como variáveis, operadores, estruturas de controle e funções, mas omitindo funcionalidades avançadas, como ponteiros e alocação dinâmica de memória. Seu uso permite o desenvolvimento de analisadores léxicos e sintáticos mais acessíveis, preparando estudantes para trabalhar com linguagens mais complexas no futuro.

2.2 Diagrama do Autômato

O diagrama do autômato representa visualmente os estados e transições de um autômato usado na análise léxica. Cada estado simboliza uma condição específica do reconhecimento de lexemas, enquanto as transições, indicadas por setas, mostram como o autômato muda de estado ao ler diferentes caracteres de entrada.

O autômato inicia em um estado inicial e, conforme processa a entrada, pode avançar para novos estados até alcançar um estado de aceitação, que indica o reconhecimento de um token válido. No contexto da linguagem C-, um autômato pode ser projetado para identificar identificadores, números, operadores e símbolos especiais, garantindo uma classificação eficiente dos tokens.

Utilizando o programa JFLAP, um diagrama foi criado representando os estados e transições do autômato implementado neste trabalho, apresentado pela Figura 1

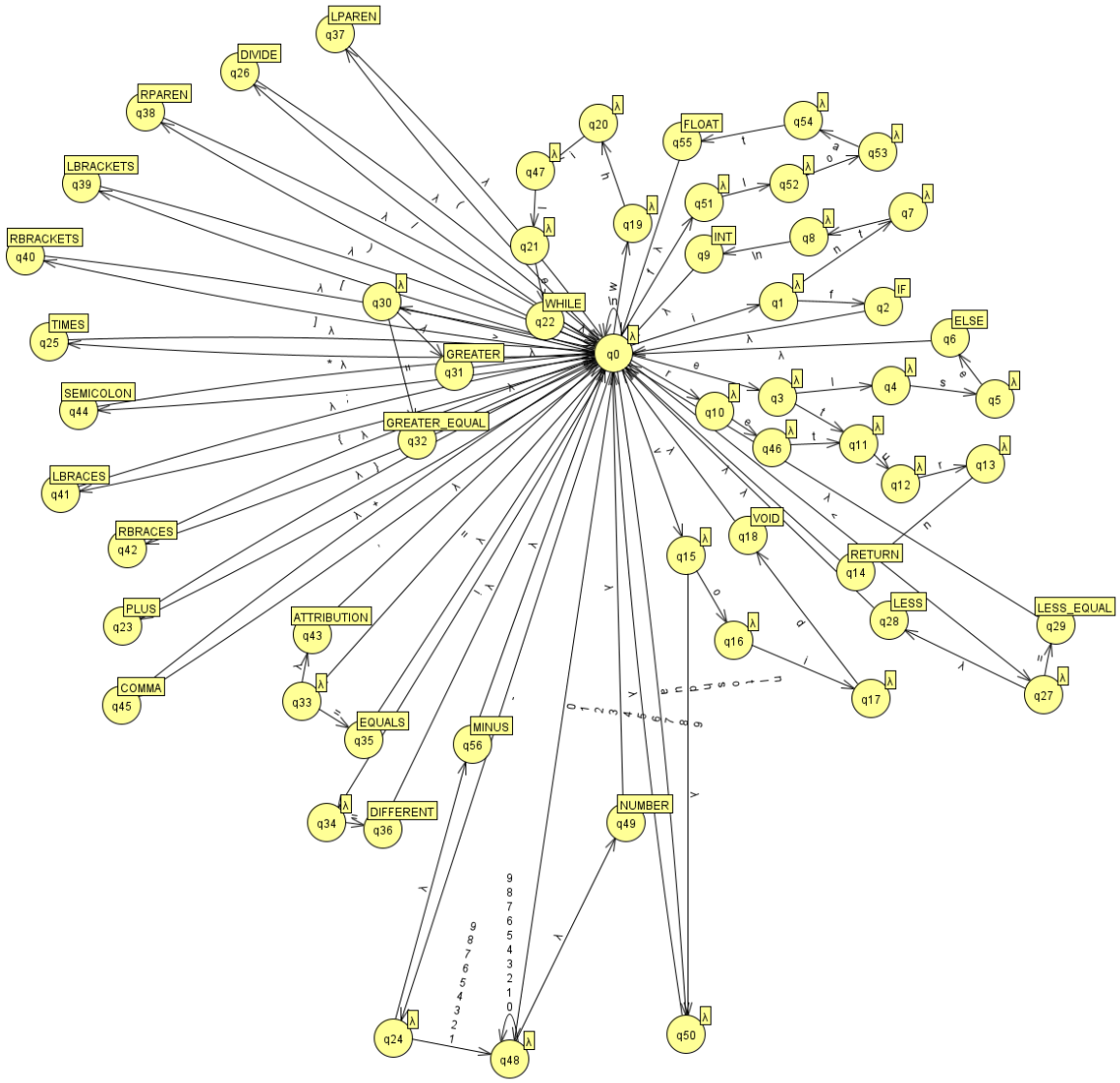

Figura 1: Diagrama feito no JFLAP

Foram criados 56 estados para representar todas as transições envolvendo todos os tokens necessários para a linguagem C-, sendo eles:

IF	ELSE	INT	FLOAT
RETURN	VOID	WHILE	PLUS
MINUS	TIMES	DIVIDE	LESS
LESS_EQUAL	GREATER	GREATER_EQUAL	EQUALS
DIFFERENT	LPAREN	RPAREN	LBRACKETS
RBRACKETS	LBRACES	RBRACES	ATTRIBUTION
SEMICOLON	COMMA	NUMBER	

Tabela 1: Tabela de Tokens

O token *ID* não foi apresentado no diagrama, pois ele é tratado de forma diferente pela implementação do código, que será apresentada posteriormente.

2.3 Implementação

A implementação do analisador léxico foi feita com a linguagem Python, que fornece a biblioteca *automata-lib*, contendo a função *Moore* para a criação do autômato. Essa função recebe 6 argumentos, sendo eles os estados, alfabeto de entrada e alfabeto de saída, transições, estado inicial e estados de saída, que são representados pelo Código 1.

Código 1: Função de Moore

```
moore = Moore(
  # Moore States
  ['q0', 'q1', 'q2', 'q3', 'q4', 'q5', 'q6', 'q7', 'q8', 'q9', 'q10',
   'q11', 'q12', 'q13', 'q14', 'q15', 'q16', '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'],
  # Moore Input Alphabet
  ['i', 'e', 'r', 'v', 'w', 'f', 'n', 'l', 't', 's', 'o', 'a', 'u', 'd',
     'h',
      '+', '-', '*', '/', '<', '=', '>', '!', ';', ',', '(', ')', '[',
   ']', '{', '}', ' ', '\n', '0', '1', '2', '3', '4', '5', '6', '7', '8',
       '9'],
  # Moore output alphabet
  ['IF', 'ELSE', 'INT', 'RETURN', 'VOID', 'WHILE', 'PLUS', 'MINUS', '
     TIMES',
   'DIVIDE', 'LESS', 'LESS_EQUAL', 'GREATER', 'GREATER_EQUAL', 'DIFFERENT
      ',
   'SEMICOLON', 'COMMA', 'LPAREN', 'RPAREN', 'LBRACKETS', 'RBRACKETS', '
     LBRACES',
   'RBRACES', 'ATTRIBUTION', 'EQUALS', 'NUMBER'],
  # Moore transitions
  {
    'q0': { 'i': 'q1', 'e': 'q3', 'r': 'q10', 'v': 'q15', 'w': 'q19', '+'
       : 'q23',
         'a': 'q50', 'u': 'q50', 'd': 'q50', 'h': 'q50', 's': 'q50', 'o
            ': 'q50',
         't': 'q50', 'l': 'q50', 'n': 'q50', 'f': 'q51', '-': 'q24', '
            *': 'q25',
         '/': 'q26', '<': 'q27', '>': 'q30', '=': 'q33', '!': 'q34', '
            ;': 'q44',
         ',': 'q45', '(': 'q37', ')': 'q38', '[': 'q39', ']': 'q40', '
            {': 'q41',
         '}': 'q42', '0': 'q48', '1': 'q48', '2': 'q48', '3': 'q48', '
            4': 'q48',
         '5': 'q48', '6': 'q48', '7': 'q48', '8': 'q48', '9': 'q48', '
             ': 'q0',
         '\n': 'q0' },
    'q1': { 'f': 'q2', 'n': 'q7' },
    'q2': { '' : 'q0' },
    'q3': { 'l': 'q4', 't': 'q11' },
    'q4': { 's': 'q5' },
    'q5': { 'e': 'q6' },
    'q6': { '' : 'q0' },
    'q7': { 't': 'q8' },
```

```
35      'q8': { '\n': 'q9', ' ': 'q9' },
36      'q9': { '' : 'q0' },
37      'q10': { 'e': 'q46' },
38      'q11': { 'u': 'q12' },
39      'q12': { 'r': 'q13' },
40      'q13': { 'n': 'q14' },
41      'q14': { '' : 'q0' },
42      'q15': { 'o': 'q16', '' : 'q50' },
43      'q16': { 'i': 'q17' },
44      'q17': { 'd': 'q18' },
45      'q18': { '' : 'q0' },
46      'q19': { 'h': 'q20' },
47      'q20': { 'i': 'q47' },
48      'q21': { 'e': 'q22' },
49      'q22': { '' : 'q0' },
50      'q23': { '' : 'q0' },
51      'q24': { '' : 'q56', '1': 'q48', '2': 'q48', '3': 'q48', '4': 'q48',
          '5': 'q48',
52              '6': 'q48', '7': 'q48', '8': 'q48', '9': 'q48' },
53      'q25': { '' : 'q0' },
54      'q26': { '' : 'q0' },
55      'q27': { '': 'q28', '=': 'q29' },
56      'q28': { '' : 'q0' },
57      'q29': { '' : 'q0' },
58      'q30': { '': 'q31', '=': 'q32' },
59      'q31': { '' : 'q0' },
60      'q32': { '' : 'q0' },
61      'q33': { '': 'q43', '=': 'q35' },
62      'q34': { '=': 'q36' },
63      'q35': { '' : 'q0' },
64      'q36': { '' : 'q0' },
65      'q37': { '' : 'q0' },
66      'q38': { '' : 'q0' },
67      'q39': { '' : 'q0' },
68      'q40': { '' : 'q0' },
69      'q41': { '' : 'q0' },
70      'q42': { '' : 'q0' },
71      'q43': { '' : 'q0' },
72      'q44': { '' : 'q0' },
73      'q45': { '' : 'q0' },
74      'q46': { 't': 'q11' },
75      'q47': { 'l': 'q21' },
76      'q48': { '0': 'q48', '1': 'q48', '2': 'q48', '3': 'q48', '4': 'q48',
          '5': 'q48',
77              '6': 'q48', '7': 'q48', '8': 'q48', '9': 'q48', '': 'q49'},
78      'q49': { '' : 'q0' },
79      'q50': { '' : 'q0' },
80      'q51': { 'l': 'q52' },
81      'q52': { 'o': 'q53' },
82      'q53': { 'a': 'q54' },
83      'q54': { 't': 'q55' },
84      'q55': { '' : 'q0' },
85      'q56': { '' : 'q0' },
```

```
86    },
87    # Initial state
88    'q0',
89    # Output state
90    {
91      'q0': '',
92      'q1': '',
93      'q2': 'IF',
94      'q3': '',
95      'q4': '',
96      'q5': '',
97      'q6': 'ELSE',
98      'q7': '',
99      'q8': '',
100     'q9': 'INT',
101     'q10': '',
102     'q11': '',
103     'q12': '',
104     'q13': '',
105     'q14': 'RETURN',
106     'q15': '',
107     'q16': '',
108     'q17': '',
109     'q18': 'VOID',
110     'q19': '',
111     'q20': '',
112     'q21': '',
113     'q22': 'WHILE',
114     'q23': 'PLUS',
115     'q24': '',
116     'q25': 'TIMES',
117     'q26': 'DIVIDE',
118     'q27': '',
119     'q28': 'LESS',
120     'q29': 'LESS_EQUAL',
121     'q30': '',
122     'q31': 'GREATER',
123     'q32': 'GREATER_EQUAL',
124     'q33': '',
125     'q34': '',
126     'q35': 'EQUALS',
127     'q36': 'DIFFERENT',
128     'q37': 'LPAREN',
129     'q38': 'RPAREN',
130     'q39': 'LBRACKETS',
131     'q40': 'RBRACKETS',
132     'q41': 'LBRACES',
133     'q42': 'RBRACES',
134     'q43': 'ATTRIBUTION',
135     'q44': 'SEMICOLON',
136     'q45': 'COMMA',
137     'q46': '',
138     'q47': '',
```

```
    'q48': '',
    'q49': 'NUMBER',
    'q50': '',
    'q51': '',
    'q52': '',
    'q53': '',
    'q54': '',
    'q55': 'FLOAT',
    'q56': 'MINUS',
  },
)
```

Seguindo a implementação do código para análise léxica, foram criadas diversas constantes representadas pelo Código 2, utilizadas para ajudar a tratar casos de erro e validações de casos específicos dentro da lógica principal do código que será apresentada posteriormente, como palavras reservadas, caracteres destas mesmas palavras reservadas, caracteres utilizados após um "ID", dígitos e caracteres inválidos.

Código 2: Constantes

```

# Constants
global check_cm
global check_key


# Reserved words
reservedWords = ['if', 'else', 'int', 'float', 'return', 'void', 'while']

# Valid letters for reserved words
validLetters = ['i', 'e', 'r', 'v', 'w', 'f', 'n', 'l', 't', 's', 'o', 'a', 'u', 'd', 'h']

# Start reserved letters
startReservedLetters = ['i', 'e', 'f', 'v', 'w', 'r']

# Characters that can be right after an identifier
validPosIdCharacters = ['(', ')', '[', ']', '{', '}', ' ', '\n', ';', ',', '+', '-', '*', '/', '<', '=', '>', '!']

# Digits
digits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

# Invalid characters
invalidCharacters = ['@', '#', '$', '%', '&', '?', '´', '`', '¨', '^', '~',
                     'º', 'ª', '§', '¬', '¢', '£', '³', '²', '¹', 'µ', '§',
                     'ª', 'º', '°', 'ç', 'Ç', 'ã', 'Ã', 'õ', 'Õ', 'á', 'Á',
                     'é', 'É', 'í', 'Í', 'ó', 'Ó', 'ú', 'Ú', 'à', 'À', 'è',
                     'È', 'ì', 'Ì', 'ò', 'Ò', 'ù', 'Ù', 'â', 'Â', 'ê', 'Ê',
                     'î', 'Î', 'ô', 'Ô', 'û', 'Û', 'ä', 'Ä', 'ë', 'Ë', 'ï
```

```
                    ',
29                'Ï', 'ö', 'Ö', 'ü', 'Ü', 'ÿ', 'Ÿ', 'ç', 'Ç', ... ,'¬
                    ']
```

O Código 3 apresenta a lógica principal utilizada na implementação do analisador léxico, possuindo uma lista de variáveis de apoio, e então um *loop* pelos símbolos lidos do arquivo de entrada escrito na linguagem C-, neste *loop* são apresentados diversos *handlers* de situações específicas que ocorrem dentro do código, como números de diferentes tamanhos, comentários dentro do código, caracteres inválidos e caracteres válidos. Dentro da lógica de caracteres válidos, os tokens são tratados como sendo *ID* ou não, caso não sejam, diversas validações de situações específicas são feitas e a atualização do estado do autômato pela sua transição é feita. Caso seja um *ID*, também são feitas validações com as constantes criadas anteriormente e os tokens são armazenados para a apresentação futura.

Código 3: Lógica principal

```
1
2  # Function to analyze the source code
3  def analysis(sourceFileString):
4    # Variables
5    state = moore.initial_state
6    outputTable = moore.output_table
7    inputAlphabet = moore.input_alphabet
8    tokens = []
9    var = ''
10   output = 'q0'
11   startReserved = False
12   startOfComment = False
13   isComment = False
14   endOfComment = False
15   startNumber = False
16   endNumber = False
17   previousSymbol = ''
18   check_key = False
19
20   # loop through the source code
21   for symbol in sourceFileString:
22
23     # Number lenght handling
24     if symbol in digits and not startNumber:
25       startNumber = True
26     if startNumber and symbol not in digits:
27       startNumber = False
28       endNumber = True
29
30     # Comment handling
31     if startOfComment and symbol != '*':
32       startOfComment = False
33     if isComment and symbol == '*' and not endOfComment:
34       endOfComment = True
35     if isComment and symbol == '/' and endOfComment:
36       isComment = False
37       endOfComment = False
38       continue
```

```
39      if symbol == '*' and startOfComment:
40        tokens.pop()
41        startOfComment = False
42        isComment = True
43      if symbol == '/' and not startOfComment:
44        startOfComment = True
45      if isComment:
46        continue
47
48      # Invalid character handling
49      if symbol in invalidCharacters:
50        # raise IOError(error_handler.newError(check_key, 'ERR-LEX-INV-CHAR
            '))
51        tokens.append(('ERROR', 'ERR-LEX-INV-CHAR'))
52
53      # Valid character handling
54      if (((symbol in inputAlphabet) and var == '' )):
55
56          # Minus handling when it is not a number
57          if (previousSymbol == '-' and symbol != ' ' and symbol != '\n'
               and symbol not in digits):
58              tokens.append(('MINUS', symbol))
59              startReserved = False
60
61          # Transition to the next state when the state has a empty
               transition
62          if ('' in moore.transitions[state] and (symbol == ' ' or (
               previousSymbol == '-' and symbol not in digits))) or endNumber
               :
63              output = moore.transitions[state]['']
64              state = output
65              endNumber = False
66              if outputTable[output] != '':
67                  tokens.append((outputTable[output], symbol))
68
69          # ID handling when symbols in the reserved words are found
70          if (startReserved and (symbol not in validLetters) and symbol !=
               ' ' and symbol != '\n'):
71              tokens.append(('ID', symbol))
72              startReserved = False
73          if (symbol in startReservedLetters and not startReserved):
74              startReserved = True
75
76          # Transition to the next state when the state has a transition
               for the symbol
77          if symbol in moore.transitions[state]:
78              output = moore.transitions[state][symbol]
79          # Transition to the next state when the state has a empty
               transition
80          else:
81              output = moore.transitions[moore.initial_state][symbol]
82
83          # Number handling
```

```
84          if output == 'q50':
85            var += symbol
86
87          # ! handling
88          if output == 'q34':
89              startReserved = False
90          elif outputTable[output] != '':
91              tokens.append((outputTable[output], symbol))
92              startReserved = False
93
94          # Update the state
95          state = output
96      # ID handling
97      else:
98          var += symbol
99          if var != '' and (symbol in inputAlphabet and (symbol not in
                validLetters or len(var) == 1)):
100             if (symbol in validPosIdCharacters):
101                 output = moore.transitions[moore.initial_state][symbol]
102                 tokens.append(('ID', var))
103                 tokens.append((outputTable[output], symbol))
104                 var = ''
105                 startReserved = False
106
107     # Update the previous symbol
108     previousSymbol = symbol
109
110   # filter '' tokens
111   tokens = list(filter(lambda x: x[0] != '', tokens))
112
113   return tokens
```

O Código 4 apresenta o restante da lógica utilizada na implementação do analisador léxico, sendo apresentado a forma de leitura do arquivo da linguagem C- e mensagens de erro.

Código 4: Restante do código

```
1
2   def main():
3       check_cm = False
4       check_key = False
5
6       for idx, arg in enumerate(sys.argv):
7           # print("Argument #{} is {}".format(idx, arg))
8           aux = arg.split('.')
9           if aux[-1] == 'cm':
10              check_cm = True
11              idx_cm = idx
12
13          if(arg == "-k"):
14              check_key = True
15
16      # print ("No. of arguments passed is ", len(sys.argv))
17
18      # Caso esteja comentado, nao passa no 1 teste do array
19      # if(len(sys.argv) < 3):
20      #     raise TypeError(error_handler.newError(check_key, 'ERR-LEX-USE'))
```

```
21
22      if not check_cm:
23        raise IOError(error_handler.newError(check_key, 'ERR-LEX-NOT-CM'))
24      elif not os.path.exists(sys.argv[idx_cm]):
25          raise IOError(error_handler.newError(check_key, 'ERR-LEX-FILE-NOT-EXISTS
                '))
26      else:
27          data = open(sys.argv[idx_cm])
28          source_file = data.read()
29          tokens = analysis(source_file)
30          for token in tokens:
31              if (token[0] == 'ERROR'):
32                  raise IOError(error_handler.newError(check_key, token[1]))
33              else:
34                  print(token[0])
35
36
37  if __name__ == "__main__":
38      try:
39          main()
40      except Exception as e:
41          print(e)
42      except (ValueError, TypeError):
43          print(e)
```

2.4 Exemplo de entrada e saída

Um exemplo de entrada e saída é apresentado respectivamente pelos Códigos 5 e 6

Código 5: Código de entrada

```
1
2  int main(void){
3  int x;
4  float y;
5
6  float u = x * y / 2;
7
8  return(0);
9  }
```

Código 6: Saída resultante

```
1
2  INT
3  ID
4  LPAREN
5  VOID
6  RPAREN
7  LBRACES
8  INT
9  ID
10 SEMICOLON
11 FLOAT
12 ID
13 SEMICOLON
14 FLOAT
15 ID
16 ATTRIBUTION
```

```
17 ID
18 TIMES
19 ID
20 DIVIDE
21 NUMBER
22 SEMICOLON
23 RETURN
24 LPAREN
25 NUMBER
26 RPAREN
27 SEMICOLON
28 RBRACES
```

3 Conclusão

Concluindo, a análise léxica baseada na Máquina de Moore para a linguagem C- demonstra a eficiência e a precisão dessa abordagem na identificação e classificação dos componentes léxicos dessa linguagem simplificada.

A Máquina de Moore, um tipo de autômato finito determinístico, é uma ferramenta poderosa para modelar a lógica da análise léxica. Utilizando estados e transições, ela reconhece padrões específicos e atribui tokens aos lexemas identificados no código-fonte.

Essa abordagem apresenta vantagens notáveis, como alta eficiência no processamento e baixo consumo de recursos computacionais. Além disso, a modularidade e flexibilidade da Máquina de Moore facilitam a manutenção e a expansão do analisador léxico.

Entretanto, é importante destacar que a análise léxica representa apenas a etapa inicial do processo de compilação. Os tokens gerados precisam ser integrados às fases seguintes, como a análise sintática e semântica, para compor um compilador completo.