

Projeto e Implementação de uma Ferramenta de Compilação para a Linguagem TPP

Mateus Santos Fernandes

Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

This paper presents the development and implementation of a Moore-type finite-state automaton designed to function as a lexical analyzer for the C- programming language. The automaton is implemented using Python, ensuring efficient token recognition and classification.

Resumo

Este trabalho apresenta o desenvolvimento e a implementação de um autômato de estados finitos do tipo Máquina de Moore, projetado para atuar como Analisador Léxico da linguagem de programação C. A implementação é realizada em Python, garantindo a identificação e classificação eficiente dos tokens.

1 Introdução

A análise léxica é uma etapa fundamental no processo de compilação, responsável por transformar o código-fonte em uma sequência de tokens com significado específico. Nessa fase inicial, ocorre a identificação e categorização dos lexemas, que correspondem a unidades léxicas como identificadores, palavras-chave, operadores e delimitadores.

Este trabalho explora a análise léxica na linguagem C, uma versão simplificada da linguagem C, amplamente adotada no meio acadêmico para ensino de fundamentos de compiladores e programação. O foco está na concepção e implementação de um analisador léxico para C, que envolve a construção de um autômato capaz de reconhecer os tokens presentes no código-fonte e a criação de um programa em Python que processa esses tokens de forma eficiente.

2 Análise Léxica

A análise léxica é a primeira etapa no processamento de linguagens de programação. Para realizar a conversão de caracteres em tokens, utilizam-se conceitos como expressões regulares, que estabelecem padrões para o reconhecimento de lexemas, e autômatos finitos determinísticos, que são modelos matemáticos capazes de processar a entrada e determinar a classificação de cada lexema. Além disso, a tabela de símbolos armazena informações relevantes sobre identificadores, incluindo tipos e escopos, auxiliando no processo de compilação.

Um aspecto crucial da análise léxica é a detecção e o tratamento de erros, garantindo que caracteres inválidos ou sequências não reconhecidas sejam devidamente identificados e sinalizados. Esse mecanismo é essencial para a robustez do analisador léxico e para a correta interpretação do código-fonte.

A análise léxica é conduzida por um autômato de Moore, um modelo de máquina de estados finitos em que cada estado possui uma saída associada. Esse autômato é estruturado com os seguintes componentes:

- **Estados:** Representam os diferentes estágios do processo de reconhecimento de tokens.
- **Alfabeto de entrada:** Conjunto de caracteres permitidos na linguagem.
- **Tabela de transições:** Define as regras que governam as mudanças de estado do autômato conforme os caracteres são lidos.
- **Tabela de saída:** Associa cada estado a um token ou a uma ação específica.

O analisador léxico desenvolvido é capaz de reconhecer os seguintes tokens:

- **Palavras reservadas:** IF, ELSE, INT, RETURN, VOID, WHILE, FLOAT.
- **Operadores:** PLUS, MINUS, TIMES, DIVIDE, LESS, LESSEQUAL, GREATER, GREATEREQUAL, DIFFERENT, EQUALS, ATTRIBUTION.
- **Delimitadores:** SEMICOLON, COMMA, LPAREN, RPAREN, LBRACKETS, RBRACKETS, LBRACES, RBRACES.
- **Números:** NUMBER.

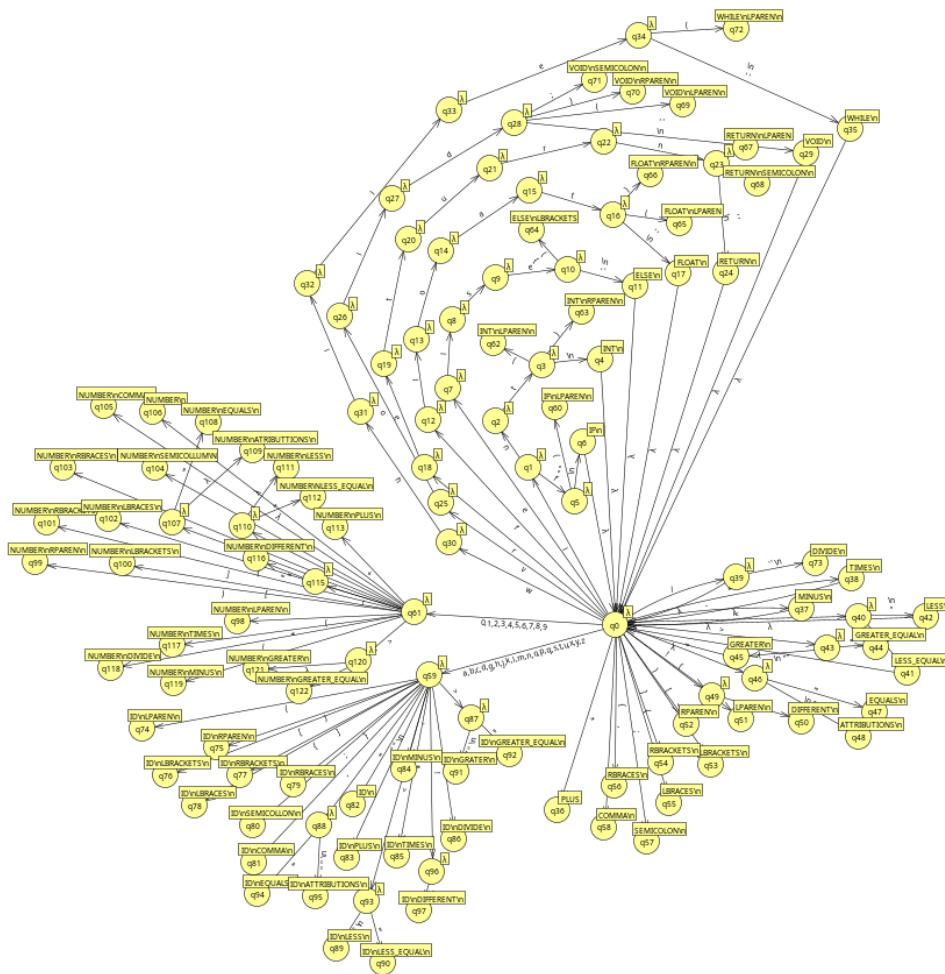
2.1 Autômato para Análise Léxica

O autômato de Moore é um modelo matemático de máquina de estados finitos no qual cada estado possui uma saída associada. Na análise léxica, ele é utilizado para reconhecer padrões no código-fonte e produzir os tokens correspondentes. A implementação do autômato no arquivo `analex.py` faz uso da biblioteca `automata`, que simplifica a definição de estados, transições e saídas.

A estrutura do autômato pode ser representada por um diagrama, onde cada estado simboliza uma condição específica do reconhecimento de lexemas. As transições, indicadas por setas, mostram como o autômato se desloca entre estados ao processar diferentes caracteres da entrada.

O funcionamento do autômato inicia em um estado inicial e, conforme percorre o código-fonte, passa por diferentes estados até alcançar um estado de aceitação, que confirma o reconhecimento de um token válido. No caso da linguagem C-, o autômato foi projetado para identificar corretamente identificadores, números, operadores e símbolos especiais, garantindo a categorização precisa dos tokens.

Para ilustrar a estrutura e o comportamento do autômato, foi utilizado o software JFLAP, que permite a criação e análise de diagramas de autômatos finitos. A Figura 1 apresenta a representação gráfica do autômato implementado neste trabalho.



2.2 Implementação

Código 1: Estados

```

1  [
2      'q0', 'q1_i', 'q2_in', 'q3_int', 'q4_intSpace', 'q5_if', 'q6_ifSpace',
3      'q7_e', 'q8_el', 'q9_els', 'q10_else', 'q11_elseSpace', 'q12_f', 'q13_fl',
4      'q14_flo', 'q15_floa', 'q16_float', 'q17_floatSpace', 'q18_r', 'q19_re',
5      'q20_ret', 'q21_retu', 'q22_retur', 'q23_return', 'q24_returnSpace',
6      'q25_v', 'q26_vo', 'q27_voi', 'q28_void', 'q29_voidSpace', 'q30_w',
7      'q31_wh', 'q32_whi', 'q33_whil', 'q34_while', 'q35_whileSpace',
8      'q36_addition', 'q37_subtraction', 'q38_multiplication', 'q39_division',
9      'q40_divisionEnd', 'q41_d', 'q42_dEqual', 'q43_Less', 'q44_LessEqual',
10     'q45_dEnd', 'q46_LessEnd', 'q47_attr', 'q48_Equal', 'q49_attr_End',
11     'q50_exclamation', 'q51_exclamationEqual', 'q52_Comma', 'q53_DotComma',
12     'q54_Dot', 'q55_leftParent', 'q56_rightParent', 'q57_leftdBrackets',
13     'q58_rightdBrackets', 'q59_leftKeys', 'q60_rightKeys', 'q61_id',
14     'q62_idLeftParent', 'q63_idRightParent', 'q64_idLeftdBrackets',
15     'q65_idRightdBrackets', 'q66_idLeftKeys', 'q67_idRightKeys', 'q68_idDotComma',
        'q69_idComma', 'q70_idSpace', 'q71_idAddition', 'q72_idSubtraction',

```

```

16      'q73_idMultiplication', 'q74_iddivision', 'q75_idd', 'q76_idLess',
17      'q77_idattr', 'q78_idexclamation', 'q79_number', 'q80_numberLeftParent',
18      'q81_numberRightParent', 'q82_numberLeftdBrackets', '
      q83_numberRightdBrackets',
19      'q84_numberLeftKeys', 'q85_numberRightKeys', 'q86_numberDotComma',
20      'q87_numberComma', 'q88_numberSpace', 'q89_numberaddition',
21      'q90_numberSubtraction', 'q91_numberMultiplication', 'q92_numberdivision',
22      'q93_numberd', 'q94_numberLess', 'q95_numberattr', 'q96_numberexclamation',
23      'q97_intLeftParent', 'q98_intRightParent', 'q99_ifLeftParent',
24      'q100_elseLeftKeys', 'q101_returnLeftParent', 'q102_returnDotComma',
25      'q103_voidRightParent', 'q104_voidDotComma', 'q105_whileLeftParent',
26      'q106_floatLeftParent', 'q107_floatRightParent', '
      q108_divisionMultiplication',
27      'q109_comment', 'q110_commentMultiplication', 'q111_commentEnd',
28      'q112_idEqual'
29 ]

```

Código 2: Alfabeto de entrada

```

1  [
2      ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
      'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '+', '-', '*', '/',
      '<', '>', '=', '!', '(', ')', '[', ']', '{', '}',
3      ';', ',', '.', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ', '\n'
4  ]

```

Código 3: Alfabeto de saída

```

1  [
2      ['ID', 'NUMBER', 'INT', 'IF', 'ELSE', 'RETURN', 'VOID', 'WHILE', 'FLOAT', '
      PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'd', 'd_EQUAL', 'LESS', 'LESS_EQUAL',
      'ATtribution', 'EQUALS', 'DIFFERENT', 'COMMA',
3      'SEMICOLON', 'LPAREN', 'RPAREN', 'LBRACKETS', 'RBRACKETS', 'LBRACES', '
      RBRACES'],
4  ]

```

Código 4: Transições

```

1  [
2      'q0' : { 'a' : 'q61_id', 'b' : 'q61_id', 'c' : 'q61_id', 'd' : 'q61_id', 'e' : '
      q7_e', 'f' : 'q12_f', 'g' : 'q61_id', 'h' : 'q61_id', 'i' : 'q1_i', 'j' : '
      q61_id', 'k' : 'q61_id', 'l' : 'q61_id', 'm' : 'q61_id', 'n' : 'q61_id', 'o' :
      'q61_id', 'p' : 'q61_id', 'q' : 'q61_id', 'r' : 'q18_r', 's' : 'q61_id', 't'
      : 'q61_id', 'u' : 'q61_id', 'v' : 'q25_v', 'w' : 'q30_w', 'x' : 'q61_id', 'y'
      : 'q61_id', 'z' : 'q61_id', '0' : 'q79_number', '1' : 'q79_number', '2' : '
      q79_number', '3' : 'q79_number', '4' : 'q79_number', '5' : 'q79_number', '6'
      : 'q79_number', '7' : 'q79_number', '8' : 'q79_number', '9' : 'q79_number',
      '+' : 'q36_addition', '-' : 'q37_subtraction', '*' : 'q38_multiplication', '/'
      : 'q39_division', '>' : 'q41_d', '<' : 'q43_Less', '=' : 'q47_attr', '!' :
      'q50_exclamation', ',' : 'q52_Comma', ';' : 'q53_DotComma', '(' : '
      q55_leftParent', ')' : 'q56_rightParent', '[' : 'q57_leftdBrackets', ']' : '
      q58_rightdBrackets', '{' : 'q59_leftKeys', '}' : 'q60_rightKeys', ' ' : 'q0'
      , '\n' : 'q0', },
3
4      'q1_i' : ...
5  ]

```

Código 5: Estado inicial e tabela de saída

```

1  'q0'
2

```

```

3 {
4     'q0' : '',
5     'q1_i' : '', 'q2_in' : '', 'q3_int' : '', 'q4_intSpace' : 'INT\n', 'q5_if' : '',
6     'q6_ifSpace' : 'IF\n', 'q7_e' : '', 'q8_el' : '', 'q9_els' : '',
7     'q10_else' : '', 'q11_elseSpace' : 'ELSE\n', 'q12_f' : '', 'q13_fl' : '',
8     'q14_flo' : '', 'q15_floa' : '', 'q16_float' : '', 'q18_r' : '',
9     'q17_floatSpace' : 'FLOAT\n', 'q19_re' : '', 'q20_ret' : '', 'q21_retu' : '',
10    'q22_retur' : '', 'q23_return' : '', 'q24_returnSpace' : 'RETURN\n', 'q25_v' : '',
11    'q26_vo' : '', 'q27_voi' : '', 'q28_void' : '', 'q29_voidSpace' : 'VOID\n',
12    'q30_w' : '', 'q31_wh' : '', 'q32_whi' : '', 'q33_whil' : '', 'q34_while' : ''
13    ,
14    'q35_whileSpace' : 'WHILE\n', 'q36_addition' : 'PLUS\n', 'q37_subtraction' : '
15    MINUS\n', 'q38_multiplication' : 'TIMES\n', 'q39_division' : '',
16    'q40_divisionEnd' : 'DIVIDE\n', 'q41_d' : '', 'q42_dEqual' : 'd_EQUAL\n',
17    'q43_dEnd' : 'd\n', 'q44_Less' : '',
18    'q44_LessEqual' : 'LESS_EQUAL\n', 'q46_LessEnd' : 'LESS\n', 'q47_attr' : '',
19    'q48_Equal' : 'EQUALS\n', 'q49_attr_End' : 'ATtribution\n',
20    'q50_exclamation' : '', 'q51_exclamationEqual' : 'DIFFERENT\n', 'q52_Comma' : '
21    COMMA\n', 'q53_DotComma' : 'SEMICOLON\n', 'q54_Dot' : '',
22    'q55_leftParent' : 'LPAREN\n', 'q56_rightParent' : 'RPAREN\n',
23    'q57_leftdBrackets' : 'LBRACKETS\n', 'q58_rightdBrackets' : 'RBRACKETS\n',
24    'q59_leftKeys' : 'LBRACES\n', 'q60_rightKeys' : 'RBRACES\n', 'q61_id' : '',
25    'q62_idLeftParent' : 'ID\nLPAREN\n', 'q63_idRightParent' : 'ID\nRPAREN\n',
26    'q64_idLeftdBrackets' : 'ID\nLBRACKETS\n', 'q65_idRightdBrackets' : 'ID\nRB
27    RACKETS\n', 'q66_idLeftKeys' : 'ID\nLBRACES\n', 'q67_idRightKeys' : 'ID\
28    nRBRACES\n',
29    'q68_idDotComma' : 'ID\nSEMICOLON\n', 'q69_idComma' : 'ID\nCOMMA\n',
30    'q70_idSpace' : 'ID\n', 'q71_idAddition' : 'ID\nPLUS\n', 'q72_idSubtraction' : 'ID\n
31    MINUS\n',
32    'q73_idMultiplication' : 'ID\nTIMES\n', 'q74_iddivision' : 'ID\nDIVIDE\n',
33    'q75_idd' : 'ID\nd\n', 'q76_idLess' : 'ID\nLESS\n', 'q77_idattr' : 'ID\
34    nATTRIBUTION\n',
35    'q78_idexclamation' : 'ID\nDIFFERENT\n', 'q79_number' : '',
36    'q80_numberLeftParent' : 'NUMBER\nLPAREN\n', 'q81_numberRightParent' : '
37    NUMBER\nRPAREN\n',
38    'q82_numberLeftdBrackets' : 'NUMBER\nLBRACKETS\n', 'q83_numberRightdBrackets' : '
39    NUMBER\nRBRACKETS\n', 'q84_numberLeftKeys' : 'NUMBER\nLBRACES\n',
40    'q85_numberRightKeys' : 'NUMBER\nRBRACES\n', 'q86_numberDotComma' : 'NUMBER\
41    nSEMICOLON\n', 'q87_numberComma' : 'NUMBER\nCOMMA\n',
42    'q88_numberSpace' : 'NUMBER\n', 'q89_numberaddition' : 'NUMBER\nPLUS\n',
43    'q90_numberSubtraction' : 'NUMBER\nMINUS\n', 'q91_numberMultiplication' : '
44    NUMBER\nTIMES\n',

```

```

39      'q92_numberdivision' : 'NUMBER\nDIVIDE\n', 'q93_numberd' : 'NUMBER\nd\n', '
      q94_numberLess' : 'NUMBER\nLESS\n', 'q95_numberattr' : 'NUMBER\
      nATtribution',
40
41      'q96_numberexclamation' : 'NUMBER\nDIFFERENT\n', 'q97_intLeftParent' : 'INT\
      nLPAREN\n', 'q98_intRightParent' : 'INT\nRPAREN\n',
42
43      'q99_ifLeftParent' : 'IF\nLPAREN\n', 'q100_elseLeftKeys' : 'ELSE\nLBRACES\n'
      , 'q101_returnLeftParent' : 'RETURN\nLPAREN\n',
44
45      'q102_returnDotComma' : 'RETURN\nSEMICOLON\n', 'q103_voidRightParent' : 'VOID
      \nRPAREN\n', 'q104_voidDotComma' : 'VOID\nSEMICOLON\n',
46
47      'q105_whileLeftParent' : 'WHILE\nLPAREN\n', 'q106_floatLeftParent' : 'FLOAT\
      nLPAREN\n', 'q107_floatRightParent' : 'FLOAT\nRPAREN\n',
48
49      'q108_divisionMultiplication' : '', 'q109_comment' : '', '
      q110_commentMultiplication' : '', 'q111_commentEnd' : '',
50 }

```

Código 6: Função main e código restante

```

1  def main():
2      # Inicializa flags e variáveis
3      check_cm = False
4      check_key = False
5      idx_cm = -1
6
7      # Verifica os argumentos passados
8      for idx, arg in enumerate(sys.argv):
9          if arg.endswith('.cm'):
10             check_cm = True
11             idx_cm = idx
12             if arg == "-k":
13                 check_key = True
14
15     # Valida o número mínimo de argumentos
16     if len(sys.argv) < 3:
17         raise TypeError(error_handler.newError(check_key, 'ERR-LEX-USE'))
18
19     # Verifica se o arquivo .cm foi fornecido e se existe
20     if not check_cm:
21         raise IOError(error_handler.newError(check_key, 'ERR-LEX-NOT-CM'))
22     elif not os.path.exists(sys.argv[idx_cm]):
23         raise IOError(error_handler.newError(check_key, 'ERR-LEX-FILE-NOT-EXISTS
24         '))
25
26     # Lê o conteúdo do arquivo .cm
27     with open(sys.argv[idx_cm], 'r') as file:
28         source_file = file.read()
29
30     # Processa o conteúdo do arquivo
31     if not check_cm:
32         print("Def")
33         print(moore)
34         print("Entrada:")
35         print(source_file)
36         print("Tokens:")
37
38     # Obtém e imprime a saída do processamento
39     print(moore.get_output_from_string(source_file))

```

```

39
40
41 if __name__ == "__main__":
42
43     try:
44         main()
45     except Exception as e:
46         print(e)
47     except (ValueError, TypeError):
48         print(ValueError)

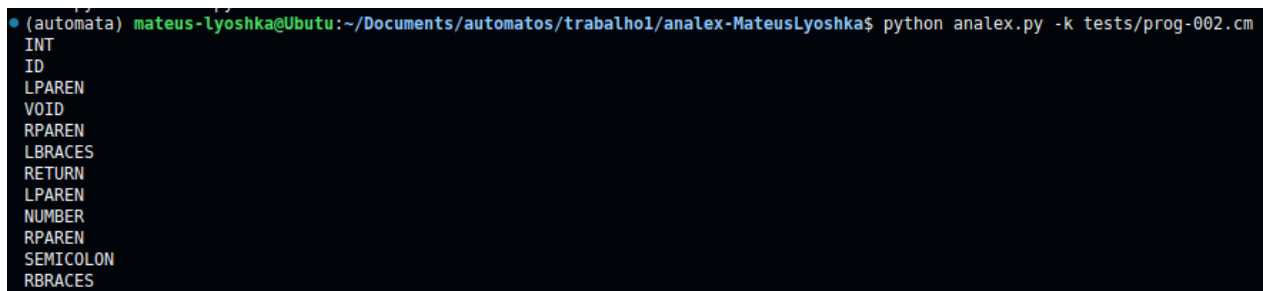
```

3 Geração de Código

Para assegurar o correto funcionamento do Analisador Léxico, foram desenvolvidos testes automatizados utilizando a biblioteca `pytest`. Esses testes, implementados no arquivo `analextest.py`, verificam se a saída gerada pelo analisador corresponde aos resultados esperados, armazenados em arquivos com extensão `.lex.out`.

3.1 Testes

```
1 python analex.py tests/prog -002.cm -k
```



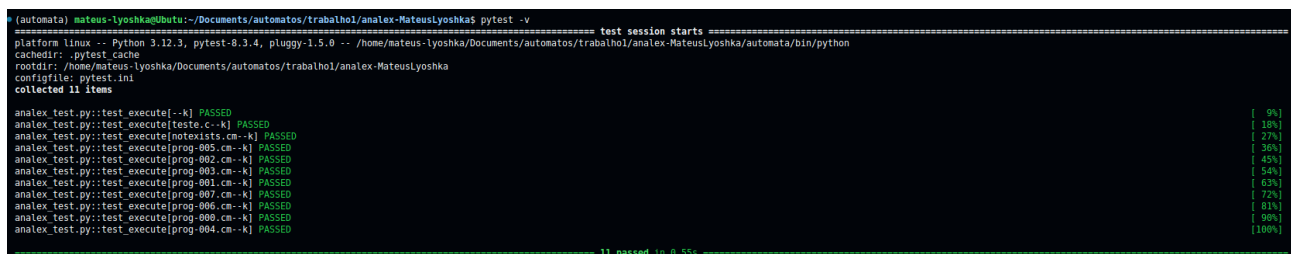
```

(mateus-lyoshka@Ubuntu:~/Documents/automatos/trabalho1/analex-MateusLyoshka$ python analex.py -k tests/prog-002.cm
INT
ID
LPAREN
VOID
RPAREN
LBRACES
RETURN
LPAREN
NUMBER
RPAREN
SEMICOLON
RBRACES

```

Figura 2: Execução do comando `python analex.py tests/prog -002.cm -k`

```
1 pytest -v
```



```

(mateus-lyoshka@Ubuntu:~/Documents/automatos/trabalho1/analex-MateusLyoshka$ pytest -v
===== test session starts =====
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0 -- /home/mateus-lyoshka/Documents/automatos/trabalho1/analex-MateusLyoshka/automata/bin/python
cachedir: .pytest_cache
rootdir: /home/mateus-lyoshka/Documents/automatos/trabalho1/analex-MateusLyoshka
configfile: pytest.ini
collected 11 items

analex_test.py::test_execute[-k] PASSED [ 9%]
analex_test.py::test_execute[test.c--k] PASSED [ 18%]
analex_test.py::test_execute[notexists.cm--k] PASSED [ 27%]
analex_test.py::test_execute[prog-005.cm--k] PASSED [ 36%]
analex_test.py::test_execute[prog-002.cm--k] PASSED [ 45%]
analex_test.py::test_execute[prog-003.cm--k] PASSED [ 54%]
analex_test.py::test_execute[prog-001.cm--k] PASSED [ 63%]
analex_test.py::test_execute[prog-007.cm--k] PASSED [ 72%]
analex_test.py::test_execute[prog-006.cm--k] PASSED [ 81%]
analex_test.py::test_execute[prog-008.cm--k] PASSED [ 90%]
analex_test.py::test_execute[prog-004.cm--k] PASSED [100%]

===== 11 passed in 0.55s =====

```

Figura 3: Execução do comando `pytest`

4 Conclusão

A implementação da análise léxica baseada em uma Máquina de Moore para a linguagem C demonstrou ser uma abordagem eficiente e precisa na identificação e categorização dos elementos léxicos dessa linguagem simplificada.

Como um modelo de autômato finito determinístico, a Máquina de Moore se mostrou uma ferramenta robusta para estruturar a lógica do analisador léxico. Através de um conjunto de

estados e transições bem definidas, o autômato é capaz de reconhecer padrões específicos no código-fonte e associá-los aos tokens correspondentes.

Entre as principais vantagens dessa abordagem, destacam-se a eficiência no processamento, o baixo consumo de recursos computacionais e a modularidade do sistema, que facilita a manutenção e a ampliação do analisador. Esses fatores tornam a Máquina de Moore uma solução prática e escalável para a análise léxica de linguagens formais.

No entanto, é fundamental ressaltar que a análise léxica representa apenas a primeira etapa do processo de compilação. Os tokens gerados precisam ser posteriormente analisados por fases subsequentes, como a análise sintática e a análise semântica, para garantir a correta interpretação e execução do código.