Project Report

Course Code: CS224

Instructor: Muhammad Sajid Ali

Group Members:

- Rafay Akram (2023491)
- Abdullah Waheed (2023048)
- Muhammad Ali (2023326)

Lexical Analyzer Name: `lexer.py`

1. Introduction

The goal of this project is to develop a Lexical Analyzer (LA) for C++ using Python. Our LA reads C++ source code, tokenizes it into keywords, identifiers, literals, operators, separators, and comments, and outputs each token along with its type and line number. This project assignment reinforces the application of regular expressions and finite automata in compiler front-end design.

2. Lexical Analyzer Implementation

Lexer.py

```
import re
from token_definations import COMPILED_PATTERNS, is_keyword, TOKEN_TYPES

class Lexer:
 def _init_(self, input_file, output_file):
     self.input_file = input_file
     self.output_file = output_file
     self.tokens = []

 def tokenize(self):
     """
     Read the input file and tokenize its contents
     """
     with open(self.input_file, 'r') as file:
         for line_number, line in enumerate(file, 1):
```

```
            self._tokenize_line(line, line_number)

    # Write tokens to output file
    self._write_output()

def _tokenize_line(self, line, line_number):
    """
    Tokenize a single line of code
    """
    position = 0
    line = line.rstrip('\n')  # Remove trailing newline

    while position < len(line):
        match = None

        # Special handling for multi-line comments
        if position + 1 < len(line) and line[position:position+2] == '/*':
            # We found the start of a multi-line comment
            # We need to read multiple lines to find the end
            comment_start = position
            comment_text = line[position:]
            end_found = '/' in line and '/' in line[line.find('/*') + 2:]
            current_line = line_number

            if not end_found:
                # Need to read more lines to find the end of comment
                comment_lines = [line[position:]]
                with open(self.input_file, 'r') as file:
                    all_lines = file.readlines()

                for i in range(line_number, len(all_lines)):
                    if '*/' in all_lines[i]:
                        # Found the end
                        end_found = True

 comment_lines.append(all_lines[i].rstrip('\n').split('/')[0] + '/')
                        break
                          elif i > line_number - 1:  # Skip current line as we
 already added it
                        comment_lines.append(all_lines[i].rstrip('\n'))

                # Combine all comment lines
                comment_text = '\n'.join(comment_lines)
```

```
            # Add the multi-line comment as a token
            self.tokens.append({
                'line': line_number,
                'type': TOKEN_TYPES['COMMENT'],
                 'value': comment_text if end_found else comment_text + "*/"  #
Ensure it ends properly
            })

            # Move position to the end of the line
            position = len(line)
            continue

        # Regular token matching
        for pattern, token_type in COMPILED_PATTERNS:
            match = pattern.match(line, position)
            if match:
                lexeme = match.group(0)

                # Skip whitespace tokens
                if token_type == TOKEN_TYPES['WHITESPACE']:
                    position = match.end()
                    continue

                # Check if identifier is actually a keyword
                      if token_type == TOKEN_TYPES['IDENTIFIER'] and
is_keyword(lexeme):
                    token_type = TOKEN_TYPES['KEYWORD']

                # Add token to our list
                self.tokens.append({
                    'line': line_number,
                    'type': token_type,
                    'value': lexeme
                })

                position = match.end()
                break

        # If no match was found, treat the character as unknown and move on
        if not match:
            self.tokens.append({
                'line': line_number,
```

```
                'type': TOKEN_TYPES['UNKNOWN'],
                'value': line[position]
            })
            position += 1

    def _write_output(self):
        """
        Write the tokens to the output file
        """
        with open(self.output_file, 'w') as file:
            for token in self.tokens:
                    file.write(f"Line {token['line']}: Token = {token['value']} ->
    {token['type']}\n")

    def get_tokens(self):
        """
        Return the list of tokens
        """
        return self.tokens
```

main.py

```
import os
import argparse
from lexer import Lexer

def main():
    # Parse command line arguments
    parser = argparse.ArgumentParser(description='C++ Lexical Analyzer')
    parser.add_argument('input', help='Input C++ file')
    parser.add_argument('-o', '--output', help='Output file for tokens',
default='')

    args = parser.parse_args()

    # Determine output file name if not specified
    if not args.output:
        input_base = os.path.splitext(args.input)[0]
        args.output = f"{input_base}_tokens.txt"

    # Check if input file exists
    if not os.path.isfile(args.input):
        print(f"Error: Input file '{args.input}' does not exist.")
```

```
        return

    # Create and run the lexer
    lexer = Lexer(args.input, args.output)
    lexer.tokenize()

    print(f"Lexical analysis complete.")
    print(f"Tokens written to '{args.output}'.")

if __name__ == "__main__":
    main()
```

token_definations.py

```
import re

# Define token types
TOKEN_TYPES = {
    'KEYWORD': 'Keyword',
    'IDENTIFIER': 'Identifier',
    'OPERATOR': 'Operator',
    'SEPARATOR': 'Separator',
    'INTEGER_LITERAL': 'Integer Literal',
    'FLOAT_LITERAL': 'Float Literal',
    'STRING_LITERAL': 'String Literal',
    'CHAR_LITERAL': 'Character Literal',
    'COMMENT': 'Comment',
    'PREPROCESSOR': 'Preprocessor Directive',
    'WHITESPACE': 'Whitespace',
    'UNKNOWN': 'Unknown'
}

# C++ keywords
CPP_KEYWORDS = [
    'alignas', 'alignof', 'and', 'and_eq', 'asm', 'auto', 'bitand', 'bitor',
    'bool', 'break', 'case', 'catch', 'char', 'char8_t', 'char16_t', 'char32_t',
    'class', 'compl', 'concept', 'const', 'consteval', 'constexpr', 'constinit',
    'const_cast', 'continue', 'co_await', 'co_return', 'co_yield', 'decltype',
    'default', 'delete', 'do', 'double', 'dynamic_cast', 'else', 'enum',
'explicit',
    'export', 'extern', 'false', 'float', 'for', 'friend', 'goto', 'if',
'inline',
    'int', 'long', 'mutable', 'namespace', 'new', 'noexcept', 'not', 'not_eq',
```

```
    'nullptr', 'operator', 'or', 'or_eq', 'private', 'protected', 'public',
    'register', 'reinterpret_cast', 'requires', 'return', 'short', 'signed',
    'sizeof', 'static', 'static_assert', 'static_cast', 'struct', 'switch',
    'template', 'this', 'thread_local', 'throw', 'true', 'try', 'typedef',
    'typeid', 'typename', 'union', 'unsigned', 'using', 'virtual', 'void',
    'volatile', 'wchar_t', 'while', 'xor', 'xor_eq'
]

# Token regex patterns
TOKEN_PATTERNS = [
    # Comments
    (r'\/\/.*', TOKEN_TYPES['COMMENT']),
    (r'\/\*(.|\n)*?\*\/', TOKEN_TYPES['COMMENT']),

    # Preprocessor directives
    (r'#\w+(?:\s+<.*?>|\s+".*?")?', TOKEN_TYPES['PREPROCESSOR']),

    # String literals
    (r'"[^"\\]*(\\.[^"\\]*)*"', TOKEN_TYPES['STRING_LITERAL']),

    # Character literals
    (r"'[^'\\]*(\\.[^'\\]*)*'", TOKEN_TYPES['CHAR_LITERAL']),

    # Float literals
    (r'\b\d+\.\d*([eE][+-]?\d+)?\b|\b\.\d+([eE][+-]?\d+)?\b|\b\d+[eE][+-]?\d+\b',
TOKEN_TYPES['FLOAT_LITERAL']),

    # Integer literals (including hex and octal)
    (r'\b0[xX][0-9a-fA-F]+\b|\b0[0-7]+\b|\b\d+\b',
TOKEN_TYPES['INTEGER_LITERAL']),

    # Keywords (added as a separate step)

    # Identifiers
    (r'\b[a-zA-Z_]\w*\b', TOKEN_TYPES['IDENTIFIER']),

    # Operators
    (r'(\+\+|--|\+=|-
=|\*=|/=|%=|&=|\|=|\^=|<<=|>>=|<<|>>|<=|>=|==|!=|&&|\|\||[+\-*/%&|^<>=!~?:])',
TOKEN_TYPES['OPERATOR']),

    # Separators
    (r'[{}()\[\];,.]', TOKEN_TYPES['SEPARATOR']),
```

```
    # Whitespace (usually ignored)
    (r'\s+', TOKEN_TYPES['WHITESPACE'])
]

# Compile regex patterns for efficiency
COMPILED_PATTERNS = [(re.compile(pattern), token_type) for pattern, token_type in
TOKEN_PATTERNS]

# Add keywords to the patterns
def is_keyword(token):
    return token in CPP_KEYWORDS
```

Notes:

- The rules cover C++ keywords, identifiers, operators, separators, numeric literals, and comments.

3. Sample Run

3.1 Input File (`test.cpp`)

```
#include <iostream>

using namespace std;

// Simple function to calculate factorial

int factorial(int n) {

 if (n <= 1) {

     return 1;

 }

 return n * factorial(n - 1);

}



int main() {
```

```
 float x = 3.14;

 // This is a comment

 cout << "Hello, World!" << endl;

 int num = 5;

 int result = factorial(num);



 if (result > 0) {

     cout << "Factorial of " << num << " is " << result << endl;

 }

 /* This is a

    multi-line comment */

 for (int i = 0; i < 10; i++) {

     x += 0.5;

 }



 return 0;

}
```

3.2 Input File (demotest.cpp)

```
int main() {
    float x = 3.14;
    // This is a comment
    if (x > 0) {
        x = x + 1;
    }
    return 0;
}
```

3.3 Expected Output(test.cpp)

```
Line 1: Token = #include <iostream> -> Preprocessor Directive
Line 2: Token = using -> Keyword
Line 2: Token = namespace -> Keyword
Line 2: Token = std -> Identifier
Line 2: Token = ; -> Separator
Line 4: Token = // Simple function to calculate factorial -> Comment
Line 5: Token = int -> Keyword
Line 5: Token = factorial -> Identifier
Line 5: Token = ( -> Separator
Line 5: Token = int -> Keyword
Line 5: Token = n -> Identifier
Line 5: Token = ) -> Separator
Line 5: Token = { -> Separator
Line 6: Token = if -> Keyword
Line 6: Token = ( -> Separator
Line 6: Token = n -> Identifier
Line 6: Token = <= -> Operator
Line 6: Token = 1 -> Integer Literal
Line 6: Token = ) -> Separator
Line 6: Token = { -> Separator
Line 7: Token = return -> Keyword
Line 7: Token = 1 -> Integer Literal
Line 7: Token = ; -> Separator
Line 8: Token = } -> Separator
Line 9: Token = return -> Keyword
Line 9: Token = n -> Identifier
Line 9: Token = * -> Operator
Line 9: Token = factorial -> Identifier
Line 9: Token = ( -> Separator
Line 9: Token = n -> Identifier
Line 9: Token = - -> Operator
Line 9: Token = 1 -> Integer Literal
Line 9: Token = ) -> Separator
Line 9: Token = ; -> Separator
Line 10: Token = } -> Separator
Line 12: Token = int -> Keyword
Line 12: Token = main -> Identifier
Line 12: Token = ( -> Separator
Line 12: Token = ) -> Separator
```

```
Line 12: Token = { -> Separator
Line 13: Token = float -> Keyword
Line 13: Token = x -> Identifier
Line 13: Token = = -> Operator
Line 13: Token = 3.14 -> Float Literal
Line 13: Token = ; -> Separator
Line 14: Token = // This is a comment -> Comment
Line 15: Token = cout -> Identifier
Line 15: Token = << -> Operator
Line 15: Token = "Hello, World!" -> String Literal
Line 15: Token = << -> Operator
Line 15: Token = endl -> Identifier
Line 15: Token = ; -> Separator
Line 17: Token = int -> Keyword
Line 17: Token = num -> Identifier
Line 17: Token = = -> Operator
Line 17: Token = 5 -> Integer Literal
Line 17: Token = ; -> Separator
Line 18: Token = int -> Keyword
Line 18: Token = result -> Identifier
Line 18: Token = = -> Operator
Line 18: Token = factorial -> Identifier
Line 18: Token = ( -> Separator
Line 18: Token = num -> Identifier
Line 18: Token = ) -> Separator
Line 18: Token = ; -> Separator
Line 20: Token = if -> Keyword
Line 20: Token = ( -> Separator
Line 20: Token = result -> Identifier
Line 20: Token = > -> Operator
Line 20: Token = 0 -> Integer Literal
Line 20: Token = ) -> Separator
Line 20: Token = { -> Separator
Line 21: Token = cout -> Identifier
Line 21: Token = << -> Operator
Line 21: Token = "Factorial of " -> String Literal
Line 21: Token = << -> Operator
Line 21: Token = num -> Identifier
Line 21: Token = << -> Operator
Line 21: Token = " is " -> String Literal
Line 21: Token = << -> Operator
```

```
Line 21: Token = result -> Identifier
Line 21: Token = << -> Operator
Line 21: Token = endl -> Identifier
Line 21: Token = ; -> Separator
Line 22: Token = } -> Separator
Line 24: Token = /* This is a multi-line comment */ -> Comment
Line 25: Token = multi -> Identifier
Line 25: Token = - -> Operator
Line 25: Token = line -> Identifier
Line 25: Token = comment -> Identifier
Line 25: Token = * -> Operator
Line 25: Token = / -> Operator
Line 27: Token = for -> Keyword
Line 27: Token = ( -> Separator
Line 27: Token = int -> Keyword
Line 27: Token = i -> Identifier
Line 27: Token = = -> Operator
Line 27: Token = 0 -> Integer Literal
Line 27: Token = ; -> Separator
Line 27: Token = i -> Identifier
Line 27: Token = < -> Operator
Line 27: Token = 10 -> Integer Literal
Line 27: Token = ; -> Separator
Line 27: Token = i -> Identifier
Line 27: Token = ++ -> Operator
Line 27: Token = ) -> Separator
Line 27: Token = { -> Separator
Line 28: Token = x -> Identifier
Line 28: Token = += -> Operator
Line 28: Token = 0.5 -> Float Literal
Line 28: Token = ; -> Separator
Line 29: Token = } -> Separator
Line 31: Token = return -> Keyword
Line 31: Token = 0 -> Integer Literal
Line 31: Token = ; -> Separator
Line 32: Token = } -> Separator
```

3.4 Expected Output(demotest.cpp)

```
Line 1: Token = int       → Keyword
Line 1: Token = main      → Identifier
Line 1: Token = (         → Separator
Line 1: Token = )         → Separator
Line 1: Token = {         → Separator
Line 2: Token = float     → Keyword
Line 2: Token = x         → Identifier
Line 2: Token = =         → Operator
Line 2: Token = 3.14      → Float Literal
Line 3: Token = // This is a comment → Comment
Line 4: Token = if        → Keyword
Line 4: Token = (         → Separator
Line 4: Token = x         → Identifier
Line 4: Token = >         → Operator
Line 4: Token = 0         → Integer Literal
Line 4: Token = )         → Separator
Line 4: Token = {         → Separator
Line 5: Token = x         → Identifier
Line 5: Token = =         → Operator
Line 5: Token = x         → Identifier
Line 5: Token = +         → Operator
Line 5: Token = 1         → Integer Literal
Line 5: Token = ;         → Separator
Line 6: Token = }         → Separator
Line 7: Token = return    → Keyword
Line 7: Token = 0         → Integer Literal
Line 7: Token = ;         → Separator
Line 8: Token = }         → Separator
```

3.5 Screenshot

Generated Output

```
abdullah@DESKTOP-15Q6HDL:~/cpp_lexer_project$ nano auto_assignment.l
abdullah@DESKTOP-15Q6HDL:~/cpp_lexer_project$ nano auto_assignment.l
abdullah@DESKTOP-15Q6HDL:~/cpp_lexer_project$ flex auto_assignment.l
gcc lex.yy.c -o auto_assignment -lfl
./auto_assignment < test.cpp
Line 1: Token = int → Keyword
Line 1: Token = main → Identifier
Line 1: Token = ( → Separator
Line 1: Token = ) → Separator
Line 1: Token = { → Separator
Line 2: Token = float → Keyword
Line 2: Token = x → Identifier
Line 2: Token = = → Operator
Line 2: Token = 3.14 → Float Literal
Line 2: Token = ; → Separator
Line 4: Token = if → Keyword
Line 4: Token = ( → Separator
Line 4: Token = x → Identifier
Line 4: Token = > → Operator
Line 4: Token = 0 → Integer Literal
Line 4: Token = ) → Separator
Line 4: Token = { → Separator
Line 5: Token = x → Identifier
Line 5: Token = = → Operator
Line 5: Token = x → Identifier
Line 5: Token = + → Operator
Line 5: Token = 1 → Integer Literal
Line 5: Token = ; → Separator
Line 6: Token = } → Separator
Line 7: Token = return → Keyword
Line 7: Token = 0 → Integer Literal
Line 7: Token = ; → Separator
Line 8: Token = } → Separator
abdullah@DESKTOP-15Q6HDL:~/cpp_lexer_project$
```

4. Conclusion

The implemented Lexical Analyzer correctly recognizes and categorizes fundamental C++ tokens, illustrating the power of pattern matching with Python. **Future work** may include:

- Handling C++ preprocessor directives and macros.
- Extending support for string and character literals.
- Integrating with a parser (e.g., using Bison) to form a complete compiler front end.

5. Submission Details

- **Folder (zipped):**

  Project_Assignment_2023491_2023048_2023326

- **Contents:**
  - `lexer.py, main.py, token_definations.py` (Flex source)
  - `test.cpp, demo_test.cpp` (sample input)
  - `demo_test_tokens.txt, test_tokens` (token list)
  - Screenshots (`.png/.jpg`)