Lexical Analyzer

Group Members

- Abdullah Ejaz Janjua 2023038
- Muhammad Ahmad Amjad 2023361

Lexical Analyzer Code

The lexical analyzer was implemented using Flex to recognize and categorize various tokens in C++ source code. The analyzer identifies tokens such as keywords, identifiers, operators, separators, literals, and comments while tracking line numbers.

```
%{
    #include <stdio.h>
    #include <string.h>
    int line = 1;
    char filename[100];
    char ofilename[100];
     FILE *inputfile, *outputfile;



%}

%%



"//".*      {}

"/*"        {

            int c;
            while((c = input()) != 0) {
              if(c == '\\n') line++;
              if(c == '*' && (c = input()) == '/') break;
              if(c == 0) break;
```

```
            }
          }

^#include[ ]*<[a-zA-Z]+> { fprintf(outputfile, "Line %d: %s \\t ⟶ Heade
r\\n", line, yytext); }

^#define[ ]*<[a-zA-Z]+> { fprintf(outputfile,"Line %d: %s \\t ⟶ Pre-proc
ess Directives\\n", line, yytext); }

^"using namespace"[ a-zA-Z]+  { fprintf(outputfile,"Line %d: %s \\t ⟶ n
amespace\\n", line, yytext); }


int|float|char|double|void|bool { fprintf(outputfile,"Line %d: %s \\t ⟶ Ty
pe\\n", line, yytext); }

if|else|while|for|return|cout { fprintf(outputfile,"Line %d: %s \\t ⟶ Keyw
ord\\n", line, yytext); }

"<<"|">>"|"+"|"-"|"*"|"/"|"="|"<"|">"|"=="|"<="|">=" { fprintf(outputfile,"Li
ne %d: %s \\t ⟶ Operator\\n", line, yytext); }

[{}();,\\[\\]]  { fprintf(outputfile,"Line %d: %s \\t ⟶ Separator\\n", line, yyt
ext); }

[0-9]+\\.[0-9]+ { fprintf(outputfile,"Line %d: %s \\t ⟶ Float\\n", line, yyt
ext); }

[0-9]+      { fprintf(outputfile,"Line %d: %s \\t ⟶ Integer\\n", line, yytext);
}

'(\\\\.|[^\\\\'])' { fprintf(outputfile,"Line %d: %s \\t ⟶ Char Literal\\n", line,
yytext); }

\\".*\\" { fprintf(outputfile,"Line %d: %s \\t ⟶ String Literal\\n", line, yytex
t); }

[a-zA-Z_][a-zA-Z0-9_]* { fprintf(outputfile,"Line %d: %s \\t ⟶ Identifier
```

```
\\n", line, yytext); }

\\n         { line++; }
[ \\t]+     {  }
.          { }

%%

yywrap() {}

int main()
{
    printf("Enter cpp file name: ");
    scanf("%99s", filename);

    for (int i = 0; i < 100; i++)
    {
        if (filename[i] == '.')
        {
            if (filename[i + 1] == 'c' && filename[i + 2] == 'p' && filename[i + 3]
== 'p')
            {

            }
            else
            {
                printf("file extention must be .cpp\\n");
                return 1;
            }
        }
    }

    strcpy(ofilename, filename);
    char *dot = strrchr(ofilename, '.'); // returns a pointer to the end of last oc
curence of '.'
    *dot = '\\0';
    strcat(ofilename, "_tokens.txt");
```

```
    inputfile = fopen(filename, "r");
    outputfile = fopen(ofilename, "w");
    if (!inputfile || !outputfile)
    {
       fprintf(outputfile,"Could not open file\\n");
       return 1;
    }



    yyin = inputfile;
    yylex();
    fclose(inputfile);
    fclose(outputfile);

}
```

Code Explanation

Our lexical analyzer uses Flex pattern matching capabilities to recognize various elements of C++ syntax:

1. **Comments Handling**:
   - Single-line comments (`//`) are recognized and ignored
   - Multi-line comments (`/* */`) are processed character by character to ensure proper line counting
2. **Preprocessor Directives:**
   - Recognizes `#include` statements and `#define` macros
   - Identifies namespace declarations
3. **Keywords and Types:**
   - Basic C++ types: int, float, char, double, void, bool
   - Control flow keywords: if, else, while, for, return
4. **Operators**:
   - Arithmetic: +, -, *, /

   - Assignment: =
   - Comparison: <, >, ==, <=, >=
   - Insertion/Extraction: <<, >>
5. **Separators**:
   - Brackets: {}, []
   - Parentheses: ()
   - Other separators: ;, ,
6. **Literals**:
   - Integer literals: Sequences of digits
   - Float literals: Digits with decimal point
   - Character literals: Single characters in single quotes
   - String literals: Text in double quotes
7. **Identifiers**:
   - Variable names, function names, etc. following C++ naming conventions
8. **Whitespace Handling**:
   - Spaces and tabs are ignored
   - Newlines are counted to track line numbers

The main function prompts the user for a C++ file, verifies its extension, opens it for reading, and passes it to the lexical analyzer.

Sample Examples and Outputs

Example 1: Basic C++ Program

Input File: example1.cpp

```
#include <iostream>
using namespace std;

int main() {
    float x = 3.14;
```

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

Output:

```
Line 1: #include <iostream>  ⟶ Header
Line 2: using namespace std  ⟶ namespace
Line 2: ;  ⟶ Separator
Line 4: int  ⟶ Type
Line 4: main  ⟶ Identifier
Line 4: (  ⟶ Separator
Line 4: )  ⟶ Separator
Line 4: {  ⟶ Separator
Line 5: float  ⟶ Type
Line 5: x  ⟶ Identifier
Line 5: =  ⟶ Operator
Line 5: 3.14  ⟶ Float
Line 5: ;  ⟶ Separator
Line 7: if  ⟶ Keyword
Line 7: (  ⟶ Separator
Line 7: x  ⟶ Identifier
Line 7: >  ⟶ Operator
Line 7: 0  ⟶ Integer
Line 7: )  ⟶ Separator
Line 7: {  ⟶ Separator
Line 8: x  ⟶ Identifier
Line 8: =  ⟶ Operator
Line 8: x  ⟶ Identifier
Line 8: +  ⟶ Operator
Line 8: 1  ⟶ Integer
Line 8: ;  ⟶ Separator
Line 9: }  ⟶ Separator
Line 10: return  ⟶ Keyword
Line 10: 0  ⟶ Integer
```

```
Line 10: ;     ⟶ Separator
Line 11: }   ⟶ Separator
```

Example 2: More Complex C++ Program

Input File: example2.cpp

```
#include <iostream>
#include <string>
#define MAX

using namespace std;

/* This is a
   multi-line comment
   spanning several lines */

double calculateArea(double length, double width) {
    return length * width;
}

int main() {
    char grade = 'A';
    string message = "Hello World";
    int numbers[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        if (numbers[i] <= 3) {
            // Skip small numbers
            continue;
        }
        cout << numbers[i] << endl;
    }

    return 0;
}
```

Output:

```
Line 1: #include <iostream>  ⟶ Header
Line 2: #include <string>  ⟶ Header
Line 3: define  ⟶ Identifier
Line 3: MAX  ⟶ Identifier
Line 5: using namespace std  ⟶ namespace
Line 5: ;  ⟶ Separator
Line 11: double  ⟶ Type
Line 11: calculateArea  ⟶ Identifier
Line 11: (  ⟶ Separator
Line 11: double  ⟶ Type
Line 11: length  ⟶ Identifier
Line 11: ,  ⟶ Separator
Line 11: double  ⟶ Type
Line 11: width  ⟶ Identifier
Line 11: )  ⟶ Separator
Line 11: {  ⟶ Separator
Line 12: return  ⟶ Keyword
Line 12: length  ⟶ Identifier
Line 12: *  ⟶ Operator
Line 12: width  ⟶ Identifier
Line 12: ;  ⟶ Separator
Line 13: }  ⟶ Separator
Line 15: int  ⟶ Type
Line 15: main  ⟶ Identifier
Line 15: (  ⟶ Separator
Line 15: )  ⟶ Separator
Line 15: {  ⟶ Separator
Line 16: char  ⟶ Type
Line 16: grade  ⟶ Identifier
Line 16: =  ⟶ Operator
Line 16: 'A'  ⟶ Char Literal
Line 16: ;  ⟶ Separator
Line 17: string  ⟶ Identifier
Line 17: message  ⟶ Identifier
Line 17: =  ⟶ Operator
Line 17: "Hello World"  ⟶ String Literal
Line 17: ;  ⟶ Separator
Line 18: int  ⟶ Type
```

```
Line 18: numbers  ⟶ Identifier
Line 18: [  ⟶ Separator
Line 18: 5  ⟶ Integer
Line 18: ]  ⟶ Separator
Line 18: =  ⟶ Operator
Line 18: {  ⟶ Separator
Line 18: 1  ⟶ Integer
Line 18: ,  ⟶ Separator
Line 18: 2  ⟶ Integer
Line 18: ,  ⟶ Separator
Line 18: 3  ⟶ Integer
Line 18: ,  ⟶ Separator
Line 18: 4  ⟶ Integer
Line 18: ,  ⟶ Separator
Line 18: 5  ⟶ Integer
Line 18: }  ⟶ Separator
Line 18: ;  ⟶ Separator
Line 20: for  ⟶ Keyword
Line 20: (  ⟶ Separator
Line 20: int  ⟶ Type
Line 20: i  ⟶ Identifier
Line 20: =  ⟶ Operator
Line 20: 0  ⟶ Integer
Line 20: ;  ⟶ Separator
Line 20: i  ⟶ Identifier
Line 20: <  ⟶ Operator
Line 20: 5  ⟶ Integer
Line 20: ;  ⟶ Separator
Line 20: i  ⟶ Identifier
Line 20: +  ⟶ Operator
Line 20: +  ⟶ Operator
Line 20: )  ⟶ Separator
Line 20: {  ⟶ Separator
Line 21: if  ⟶ Keyword
Line 21: (  ⟶ Separator
Line 21: numbers  ⟶ Identifier
Line 21: [  ⟶ Separator
Line 21: i  ⟶ Identifier
```

```
Line 21: ]      ⟶ Separator
Line 21: <=     ⟶ Operator
Line 21: 3      ⟶ Integer
Line 21: )      ⟶ Separator
Line 21: {      ⟶ Separator
Line 23: continue    ⟶ Identifier
Line 23: ;      ⟶ Separator
Line 24: }      ⟶ Separator
Line 25: cout    ⟶ Keyword
Line 25: <<      ⟶ Operator
Line 25: numbers    ⟶ Identifier
Line 25: [      ⟶ Separator
Line 25: i      ⟶ Identifier
Line 25: ]      ⟶ Separator
Line 25: <<      ⟶ Operator
Line 25: endl    ⟶ Identifier
Line 25: ;      ⟶ Separator
Line 26: }      ⟶ Separator
Line 28: return     ⟶ Keyword
Line 28: 0      ⟶ Integer
Line 28: ;      ⟶ Separator
Line 29: }      ⟶ Separator
```

Design Considerations and Implementation Details

Pattern Matching Strategy

Our lexical analyzer follows a "longest match" strategy, where it tries to match the longest possible pattern at each step. The patterns are arranged in a specific order to ensure correct token identification. For example, keywords and identifiers have patterns that could potentially overlap, so keywords are checked first.

Comment Handling

For handling multi-line comments, we implemented a character-by-character scanning approach to ensure accurate line counting. This approach also helps in properly identifying the end of a comment with the "*/" sequence.

Conclusion

This project successfully demonstrates the application of formal language theory concepts, specifically regular expressions, in creating a lexical analyzer for C++ programs. The analyzer can identify and categorize various tokens as required, providing a solid foundation for the front-end of a compiler.

The implementation effectively uses Flex to define patterns for different token types, manages state through line counting, and handles complex structures like comments. The output format provides clear information about each token's type, value, and line number, which can be useful for subsequent phases of compilation.

Through this project, we gained practical experience in formal language applications and a better understanding of the lexical analysis phase of compiler design.