CSM 3202: Compiler Lab

Lab Report 2: Flex (Lex) program for required lexical analysis.

Name: Md. Tamim Ahmed Fahim

ID: 2209024

Level: 3, Semester: 2

Bioinformatics Engineering

Bangladesh Agricultural University

Report submitted: February 03,2025

This is a documentation to the Lab 2 done in the Compiler Lab Course. The report is submitted to Md. Saif Uddin, Lecturer, Department of Computer Science and Mathematics, Bangladesh Agricultural University.

Contents

ListofFigures

ListofTables

Listings

1 Introduction

1.1 Background of the Experiment

Lexical analysis is the initial phase of a compiler where the source code, a sequence of characters, is read and converted into a sequence of meaningful units called tokens. This process is performed by a program known as a lexical analyzer.
The process is used in compiler design for the following reasons:

- **Simplifies Compiler Design:** Separating character-level processing from grammar-level processing makes each phase more manageable.
- **Improves Efficiency:** Specialized and optimized techniques can be applied to the simple, repetitive task of scanning the input stream.
- **Enhances Portability:** Input-device-specific peculiarities are restricted to the lexer phase.
- **Removes Unnecessary Information:** The lexer handles and discards non-essential elements like whitespace and comments.

The overall purpose of token identification in lexical analysis is to convert raw source code into a structured format that the rest of the compiler can efficiently understand and process. The key purposes are:

- **Simplification of Input:** It transforms a messy stream of individual characters into a clean sequence of meaningful units (tokens).
- **Foundation for Parsing:** Tokens serve as the fundamental building blocks for the next phase, syntax analysis (parsing), which checks the program's structure.
- **Removal of Noise:** Non-essential elements like whitespace and comments are discarded, allowing later compiler stages to focus purely on the code's logic.
- **Efficient Processing:** Working with tokens is significantly faster and easier for a parser than dealing with every single character individually.
- **Early Error Detection:** The process catches basic lexical errors (e.g., illegal characters or invalid number formats) early in the compilation cycle.

1.2 Objectives of the Lab

The objectives for studying lexical analysis and compiler design are to learn how to perform the following tasks effectively:

- **Define and Identify Tokens:** To accurately break down source code into a language's specific terminal symbols, classifying raw characters into categories such as keywords, identifiers, operators, and literals.
- **Understand Regular Expressions:** To master the use of regular expressions as the formal method for defining the patterns of these token types.
- **Implement a Lexer (Scanner):** To write a functional lexical analyzer program, either manually or using tools like Lex or Flex, that reads an input stream and outputs a sequence of identified tokens.
- **Manage Symbol Tables:** To understand how token identification interacts with symbol tables, storing and retrieving information about identifiers and their attributes for use in later compiler phases.

1.3 Tools and Requirements

- 1.Windows11 os ,
- 2.VS code ,
- 3.C Compiler.
- 4.flex.

2 Tasks and Implementations

2.1 Task 1: Character, Word, and Line Counter

2.1.1 Task Description

A Flex program that reads a source file and counts the total number of:

- Counting Characters ,
- Counting Words ,
- Counting Lines

2.1.2 Source Code

```
%{
#include<stdio.h>
int char_count = 0;
int word_count = 0;
int line_count = 0;
%}

%%

\n { line_count++; char_count++; }
[ \t]+ { char_count += yyleng; }
\"[^\"]*\" { word_count++; char_count += yyleng; }
[^ \t\n]+ { word_count++; char_count+= yyleng; }

%%

int yywrap(){
    return 1;
}


int main()
{
    FILE *fp = fopen("input.txt", "r");
    if (!fp)
    {
```

```
        printf("Error: Cannot open input.txt\n");
        return 1;
    }

    yyin = fp;
    yylex();

    printf("Characters: %d\n", char_count);
    printf("Words      : %d\n", word_count);
    printf("Lines      : %d\n", line_count);

    fclose(fp);
    return 0;
}
```

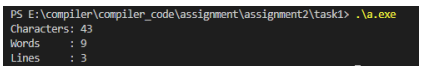
Listing 1: A Flex program that reads a source file and counts the total number of Characters, Words, Lines

2.1.3 Input & Output

Input:

```
int main () {
printf ("Hello World\n") ;
}
```

Output:



2.1.4 Explanation of Code

Use three counters. Match newlines (count line+char), whitespace (count chars using yyleng), and words (count word+chars using yyleng). Read from input.txt and print totals.

2.2 Task 2: C Keyword Identifier

2.2.1 Task Description

A Flex program to identify and print all C keywords present in the line. Non-keyword identifiers must not be printed.

2.2.2 Source Code

```
%{
#include <stdio.h>
%}

%%
[a-zA-Z_][a-zA-Z0-9_]*      { printf("%s:valid
   identifier\n", yytext); }

[0-9]+[a-zA-Z_][a-zA-Z0-9_]*  { printf("%s:invalid
   identifier\n", yytext); }

[0-9]+                      { /* Ignore numbers */ }
[+\-*/=<>!&|(){}[\];,.]     { /* Ignore operators and
   punctuation */ }
[ \t\n]                     { /* Ignore whitespace */
   }
.                           { /* Ignore other
   characters */ }

%%
int yywrap() {
    return 1;
}

int main()
{
    FILE *fp = fopen("identifiers.txt","r");
    if(!fp){
        printf("can not open the file\n");
        return 0;
    }
    yyin=fp;
    yylex();
```

```
    fclose(fp);
    return 0;
}
}
```

Listing 2: A Flex program to identify and print all C keywords present in the line.

2.2.3 Explanation of Code

Valid pattern: starts with letter/underscore $[a\text{-}zA\text{-}Z_][a-zA-Z0-9_]*$ *.Invalidpatterns : startswithdigitorcontainsspecialchars.Printclassificationforeachmatch.*

2.2.4 Input & Output

Input:

```c
int main() {
    int x = 10;
    if (x > 0) return x;
}
```

Output:

```
PS E:\compiler\compiler_code\assignment\assignment2\task1> .\a.exe
Characters: 43
Words     : 9
Lines     : 3
```

2.3 Task 3: Identifier

2.3.1 Task Description

A Flex program to extract all identifiers from a given line and classify them as valid or invalid.

2.3.2 Source Code

```
%{
#include <stdio.h>
%}

%%

[a-zA-Z_][a-zA-Z0-9_]*  { printf("%s : valid
   identifier\n", yytext); }
[0-9][a-zA-Z0-9_]*   { printf("%s : invalid identifier
   \n", yytext); }
[ \t\n]+  ;
.         ;

%%

int main() {
    FILE *fp = fopen("input.txt", "r");
    if (!fp) {
        printf("Error: Cannot open input.txt\n");
        return 1;
    }

    yyin = fp;
    yylex();

    fclose(fp);
    return 0;
}

int yywrap() {
    return 1;
}
```

Listing 3: A Flex program to identify identifiers.

2.3.3 Explanation of Code

Valid pattern: An identifier must start with a letter or underscore [a-zA-Z_][a−zA−Z0−9_].Invalidpatterns : Identifiersstartingwithadigitorcontainingspecialcharactersare*

2.3.4 Input & Output

Input:

```
Enter a line :
id1 , _id2 , 3 id , id 5
```

Output:

2.4 Task 4: Identification of Numeric Constants

2.4.1 Identify Integer and Floating-point

2.4.2 Task Description

The task detects and classify the numeric constants

- *Integer,*
- *Floating-point*

2.4.3 Source Code

```
%{
#include <stdio.h>
%}

%%
([0-9]+\.[0-9]*([eE][+-]?[0-9]+)?)|(\.[0-9]+([eE
   ][+-]?[0-9]+)?)|([0-9]+[eE][+-]?[0-9]+)  { printf
   ("%s : float\n", yytext); }

[0-9]+  { printf("%s : integer\n", yytext); }

[ \t\n]+    ;
.           ;
%%

int main() {
    FILE *fp = fopen("input.txt", "r");
    if (!fp) {
        printf("Error: Cannot open input.txt\n");
        return 1;
    }

    yyin = fp;
    yylex();

    fclose(fp);
    return 0;
}

```

```
int yywrap() {
    return 1;
}
```

Listing 4: A flex program to identify Numeric Constants

2.4.4 Explanation of Code

The task scans each token and classifies the numaric constants. Pattern for float: [0-9]+.[0-9]+. Pattern for int: [0-9]+. Float pattern must come first. Print type when matched.

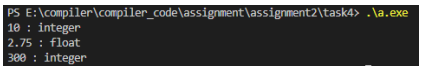
2.4.5 Input & Output

Input:

```
Enter a line :
a = 10; b = 2.75; c = 300;

```

Output:



2.5 Task 5: Identification of Operators

2.5.1 Assignment, Arithmetic, Relational, and Logical operators.

2.5.2 Task Description

The task detects the following operators :

- *Assignment*
- *Arithmetic*
- *Relational*
- *Logical*

2.5.3 Source Code

```
%{
#include <stdio.h>
%}

%%
"&&"        { printf("%s : logical operator\n", yytext); }
"||"        { printf("%s : logical operator\n", yytext); }
"!"         { printf("%s : logical operator\n", yytext); }

"=="        { printf("%s : relational operator\n", yytext); }
"!="        { printf("%s : relational operator\n", yytext); }
">="        { printf("%s : relational operator\n", yytext); }
"<="        { printf("%s : relational operator\n", yytext); }
">"         { printf("%s : relational operator\n", yytext); }
"<"         { printf("%s : relational operator\n", yytext); }

```

```
"="             { printf("%s : assignment operator\n",
   yytext); }

"+"             { printf("%s : arithmetic operator\n",
   yytext); }
"-"             { printf("%s : arithmetic operator\n",
   yytext); }
"*"             { printf("%s : arithmetic operator\n",
   yytext); }
"/"             { printf("%s : arithmetic operator\n",
   yytext); }
"%"             { printf("%s : arithmetic operator\n",
   yytext); }

[ \t\n]+        ;
.               ;

%%

int main() {
    FILE *fp = fopen("input.txt", "r");
    if (!fp) {
        printf("Error: Cannot open input.txt\n");
        return 1;
    }

    yyin = fp;
    yylex();
    fclose(fp);
    return 0;
}

int yywrap() {
    return 1;
}
```

Listing 5: A Flex program to identify and print operators

2.5.4 Explanation of Code

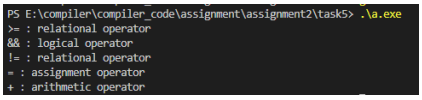
Scans the file and identifies assignment, arithmetic, relational, and logical operators, printing each when detected.

2.5.5 Input & Output

Input:

```
Enter a line :
if ( a >= 10 && b != 5) a = a + 1;
```

Output:



2.6 Task 6: Identification of Punctuation and Special Symbols

2.6.1 Punctuation & Special symbols

2.6.2 Task Description

The task detects the following operators :

- *Assignment*
- *Arithmetic*
- *Relational*
- *Logical*

2.6.3 Source Code

```
%{
#include <stdio.h>
%}

%%

"{"       { printf("%s : special symbol\n", yytext); }
"}"       { printf("%s : special symbol\n", yytext); }
"("       { printf("%s : special symbol\n", yytext); }
")"       { printf("%s : special symbol\n", yytext); }
";"       { printf("%s : special symbol\n", yytext); }
","       { printf("%s : special symbol\n", yytext); }
"["       { printf("%s : special symbol\n", yytext); }
"]"       { printf("%s : special symbol\n", yytext); }
"."       { printf("%s : special symbol\n", yytext); }

[ \t\n]+    ;
.           ;

%%

int main() {
    FILE *fp = fopen("input.txt", "r");
    if (!fp) {
```

```
        printf("Error: Cannot open input.txt\n");
        return 1;
    }

    yyin = fp;
    yylex();
    fclose(fp);
    return 0;
}

int yywrap() {
    return 1;
}
```

Listing 6: A flex program to identify punctuation and special symbols

2.6.4 Explanation of Code

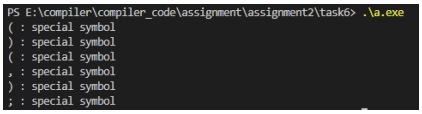
The task helped to create literal patterns for each symbol. Print "Special symbol:" with the matched symbol. Ignore other characters.

2.6.5 Input & Output

Input:

```
Enter a line :
if( a %2==0) printf ("%d is even ", a
) ;
```

Output:



2.7 Task 7: Identification of Single-line and Multi-line Comments

2.7.1 Comments identification

2.7.2 Task Description

The task detects-

- *Single-line comments*
- *Multi-line comments*

2.7.3 Source Code

```
%{
#include <stdio.h>
%}

%%


"//".*      { printf("Single-line comment\n"); }
"/*"([^*]|\*+[^*/])*(\*+"/")     { printf("Multi-line
   comment\n"); }
"/*"([^*]|\*+[^*/])* { printf("Lexical Error:
   unterminated comment\n"); }

[ \t\n]+          ;
.                 ;

%%

int main() {
    FILE *fp = fopen("input.txt", "r");
    if (!fp) {
        printf("Error: Cannot open input.txt\n");
        return 1;
    }

    yyin = fp;
    yylex();
```

```
    fclose(fp);
    return 0;
}

int yywrap() {
    return 1;
}
```

Listing 7: A flex program to identify single-line and multi-line comments

2.7.4 Explanation of Code

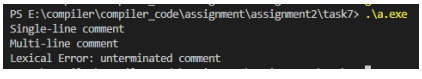
This Flex program scans a source file to detect comments. It identifies single-line comments starting with //, multi-line comments enclosed in / ... */, and reports unterminated multi-line comments as lexical errors. Whitespace and other characters are ignored. The program reads from input.txt and prints the detected comment type or error.*

2.7.5 Input & Output

Input:

```
// Single-line comment
int x = 5;
/* Multi-line
   comment */
/* Unterminated comment
x = 10;
```

Output:



2.8 Task 8: Identification of String Literals and Character Constants

2.8.1 Task Description

Validate strings and char constants. Detect errors: unterminated strings, empty chars (''), multiple chars ('ab').

2.8.2 Source Code

```
%{
#include <stdio.h>
%}

%%


\"([^"\n]|\\.)*\"              { printf("%s : Valid
   String Literal\n", yytext); }

\"([^"\n]|\\.)*                { printf("%s :
   Unterminated string Literal  (Invalid)\n", yytext);
    }

\'([^\'\n]|\\.)\'               { printf("%s : Valid
   Character constant\n", yytext); }

\'\'                            { printf("'' : Empty
   char constant (Invalid)\n"); }

\'([^\'\n]|\\.)[^\'\n]+\'       { printf("%s : Multiple
    characters in constant (Invalid)\n", yytext); }

\'([^\'\n]|\\.)*               { printf("' :
   Unterminated char constant (Invalid)\n"); }

[ \t\n]+          ;
.                 ;

%%

```

```
int main() {
    FILE *fp = fopen("input.txt", "r");
    if (!fp) {
        printf("Error: Cannot open input.txt\n");
        return 1;
    }

    yyin = fp;
    yylex();
    fclose(fp);
    return 0;
}

int yywrap() {
    return 1;
}
```

Listing 8: A flex program to identify string literals and character constants

2.8.3 Explanation of Code

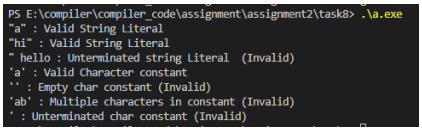
The code detects string and character literals using Valid string: "[^"] ". Unterminated : quotewithoutclosingbeforenewline. Validchar : singlequote, onechar, singlequ charpatterns.*

2.8.4 Input & Output

Input:

```
"a"
"hi"
" hello
'a'
''
'ab'
'a
```

Output:


PS E:\compiler\compiler_code\assignment\assignment2\task8> .\a.exe
"a" : Valid String Literal
"hi" : Valid String Literal
" hello : Unterminated string Literal (Invalid)
'a' : Valid Character constant
'' : Empty char constant (Invalid)
'ab' : Multiple characters in constant (Invalid)
' : Unterminated char constant (Invalid)

2.9 Task 9: Full Lexical Token Classification and Token Stream Generation

2.9.1 Complete Lexical Analyzer

2.9.2 Lexical Classification

Classify all lexemes into token types (keyword, identifier, operator, constant, special symbol, string) and generate them in the `<token_type, lexeme>` *format while counting each type.*

2.9.3 Source Code

```
        %{
#include <stdio.h>
int keyword_count = 0;
int identifiers_count = 0;
int constant_count = 0;
int operators_count = 0;
int sp_symbol_count = 0;
int string_count = 0;
int invalid_count = 0;
%}

%%
int([^a-zA-Z0-9_]|$)        { printf("<int : keyword>\n
   "); keyword_count++;}
return([^a-zA-Z0-9_]|$)     { printf("<return : keyword
   >\n"); keyword_count++; }
if([^a-zA-Z0-9_]|$)         { printf("<if : keyword>\n
   "); keyword_count++;}
for([^a-zA-Z0-9_]|$)        { printf("<for : keyword>\n
   "); keyword_count++;}
while([^a-zA-Z0-9_]|$)      { printf("<while : keyword
   >\n"); keyword_count++;}
float([^a-zA-Z0-9_]|$)      { printf("<float : keyword
   >\n"); keyword_count++;}
char([^a-zA-Z0-9_]|$)       { printf("<char : keyword>\
   n"); keyword_count++;}
double([^a-zA-Z0-9_]|$)     { printf("<double : keyword
   >\n"); keyword_count++;}
```

```
21 void([^a-zA-Z0-9_]|$)       { printf("<void : keyword>\
      n"); keyword_count++;}
22 long([^a-zA-Z0-9_]|$)       { printf("<long : keyword>\
      n"); keyword_count++;}
23 short([^a-zA-Z0-9_]|$)      { printf("<short : keyword
      >\n"); keyword_count++;}
24 unsigned([^a-zA-Z0-9_]|$)   { printf("<unsigned :
      keyword>\n"); keyword_count++;}
25 signed([^a-zA-Z0-9_]|$)     { printf("<signed : keyword
      >\n"); keyword_count++;}
26
27 \"([^"\n]|\\.)*\"             { printf("<%s : String
      Literal>\n", yytext); string_count++;}
28
29 \"([^"\n]|\\.)*               { invalid_count++; }
30
31 \'([^\'\n]|\\.)\'              { printf("<%s :
      Character constant>\n", yytext); string_count++;}
32
33 \'\'                           { invalid_count++;}
34
35 \'([^\'\n]|\\.)[^\'\n]+\'      { invalid_count++; }
36
37 \'([^\'\n]|\\.)*               { invalid_count++; }
38
39 ([0-9]+\.[0-9]*([eE][+-]?[0-9]+)?)|(\.[0-9]+([eE
      ][+-]?[0-9]+)?)|([0-9]+[eE][+-]?[0-9]+)  { printf
      ("<%s : constant>\n", yytext); constant_count++; }
40 [0-9]+  { printf("<%s : constant>\n", yytext);
      constant_count++; }
41
42 [a-zA-Z_][a-zA-Z0-9_]*  { printf("<%s : identifier>\n
      ", yytext);identifiers_count++; }
43 [0-9][a-zA-Z0-9_]*   { invalid_count++; }
44
45
46 "&&" { printf("<%s : logical operator>\n", yytext);
      operators_count++; }
47 "||" { printf("<%s : logical operator>\n", yytext);
      operators_count++; }
```

```
48 "!"  { printf("<%s : logical operator>\n", yytext);
     operators_count++; }
49
50 "==" { printf("<%s : relational operator>\n", yytext);
     operators_count++; }
51 "!=" { printf("<%s : relational operator>\n", yytext);
     operators_count++; }
52 ">=" { printf("<%s : relational operator>\n", yytext);
     operators_count++; }
53 "<=" { printf("<%s : relational operator>\n", yytext);
     operators_count++; }
54 ">"  { printf("<%s : relational operator>\n", yytext);
     operators_count++; }
55 "<"  { printf("<%s : relational operator>\n", yytext);
     operators_count++; }
56
57 "="  { printf("<%s : assignment operator>\n", yytext);
     operators_count++; }
58
59 "+"  { printf("<%s : arithmetic operator>\n", yytext);
     operators_count++; }
60 "-"  { printf("<%s : arithmetic operator>\n", yytext);
     operators_count++; }
61 "*"  { printf("<%s : arithmetic operator>\n", yytext);
     operators_count++; }
62 "/"  { printf("<%s : arithmetic operator>\n", yytext);
     operators_count++; }
63 "%"  { printf("<%s : arithmetic operator>\n", yytext);
     operators_count++; }
64
65 "{" { printf("<%s : special symbol>\n", yytext);
     sp_symbol_count++; }
66 "}" { printf("<%s : special symbol>\n", yytext);
     sp_symbol_count++; }
67 "(" { printf("<%s : special symbol>\n", yytext);
     sp_symbol_count++; }
68 ")" { printf("<%s : special symbol>\n", yytext);
     sp_symbol_count++; }
69 ";" { printf("<%s : special symbol>\n", yytext);
     sp_symbol_count++; }
```

```
70 "," { printf("<%s : special symbol>\n", yytext);
      sp_symbol_count++; }
71 "[" { printf("<%s : special symbol>\n", yytext);
      sp_symbol_count++; }
72 "]" { printf("<%s : special symbol>\n", yytext);
      sp_symbol_count++; }
73 "." { printf("<%s : special symbol>\n", yytext);
      sp_symbol_count++; }
74
75
76
77
78
79 [ \t\n]+    ;
80 .           ;
81 %%
82 int main() {
83     FILE *fp = fopen("input.txt", "r");
84     if (!fp) {
85         printf("Error: Cannot open input.txt\n");
86         return 1;
87     }
88
89     yyin = fp;
90     yylex();
91
92     printf("\n\nToken Counts :- \n");
93     printf("keyword: %d\n", keyword_count);
94     printf("identifiers: %d\n", identifiers_count);
95     printf("constants: %d\n", constant_count);
96     printf("operators: %d\n", operators_count);
97     printf("special symbols: %d\n", sp_symbol_count);
98     printf("string literals: %d\n", string_count);
99     printf("Invalid tokens: %d\n", invalid_count);
100
101    fclose(fp);
102    return 0;
103 }
104
105 int yywrap() {
106     return 1;
```

```
107 }
```

Listing 9: A flex program of classification and token stream generatiion

2.9.4 Explanation of Code

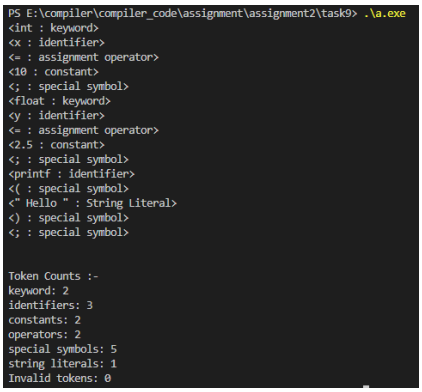
Combine all previous patterns. Maintain counters for each token type. Print in <lexeme, type> format when matched. Print summary counts at end.

2.9.5 Input & Output

Input:

```
1 int x = 10;
2 float y = 2.5;
3 printf (" Hello ") ;
```

Output:



2.10 Task 10: Detection of Lexical Errors

2.10.1 Lexical Error Detector

2.10.2 Task Description

Detect and report errors: identifiers starting with digits, invalid chars in identifiers (@,$, #) unterminated strings, unclosed comments.

2.10.3 Source Code

```
%{
#include <stdio.h>
%}

%%

\"([^"\n]|\\.)*          { printf("Lexical Error: Unterminated string literal: %s\n", yytext); }
"/*"(.|\n)*$             { printf("Lexical Error: Unclosed comment\n"); }
[0-9]+[a-zA-Z_]+         { printf("Lexical Error: Invalid identifier: %s\n", yytext); }
[a-zA-Z_]+-[a-zA-Z_]+    { printf("Lexical Error: Invalid identifier: %s\n", yytext); }
"@"[a-zA-Z_]+            { printf("Lexical Error: Invalid identifier: %s\n", yytext); }
"@"|"$"|"#"              { printf("Lexical Error: Invalid character: %s\n", yytext); }

[ \t\n]+                 ;
.                        ;

%%
int main() {
    FILE *fp = fopen("input.txt", "r");
    if (!fp) {
        printf("Error: Cannot open input.txt\n");
        return 1;
    }

    yyin = fp;
```

```
    yylex();

    fclose(fp);
    return 0;
}

int yywrap() {
    return 1;
}
```

Listing 10: A flex program to detect lexical errors

2.10.4 Explanation of Code

This Flex program detects lexical errors in a source file. It reports unterminated string literals, unclosed comments, invalid identifiers (like starting with digits, containing hyphens, or starting with @), and invalid characters (@,). Whitespace and other characters are ignored. The program reads from input.txt and prints the corresponding error messages.

2.10.5 Input & Output

Input:

```
int 2sum = 10;
2 char * s = " hello ;
3 float x@ = 2.5;

```

Output:

```
PS E:\compiler\compiler_code\assignment\assignment2\task10> .\a.exe
Lexical Error: Invalid identifier: 2sum
Lexical Error: Unterminated string literal: " hello ;
Lexical Error: Invalid character: @
```

3 Conclusion

In this lab, we implemented lexical analysis using flex , to scan a source file and classify its elements into keywords, identifiers, constants, operators, and special symbols while detecting errors like invalid tokens or unterminated strings and comments. This exercise shows the importance of lexical analysis in ensuring correct syntax and preparing code for further compilation steps.