A Report on
Compiler Design Lab (CS304): Mini Project Phase 1

by

Akshat Bharara (Roll No: 231CS110)

Rakshith Ashok Kumar (Roll No: 231CS147)

Utsav Singh Bhamra (Roll No: 231CS161)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA
SURATHKAL, MANGALURU-575025
13-August-2025

1 Introduction

1.1 Lexical Analysis

Lexical analysis is the first phase of a compiler that scans the source program as a stream of characters and converts it into meaningful lexemes or tokens. The lexical analyzer plays a crucial role in detecting the structure of source code and providing input to the syntax analyzer.

1.2 Tokens & Lexemes

A token is a category of lexical units such as keywords, identifiers, constants, operators, and punctuation symbols. A lexeme is the actual text from the source code that matches a token pattern. For example, in the statement:

```
int x = 10;
```

the lexeme `int` corresponds to the token **KEYWORD**, `x` corresponds to the token **IDENTIFIER**, and `10` corresponds to the token **INTEGER_CONSTANT**.

2 Overview of Code

The lexical analyzer was implemented using **LEX/Flex**. The scanner supports:

- Recognition of identifiers, keywords, preprocessor directives, constants, operators, and punctuation.
- Handling of white spaces, single-line comments, and nested multi-line comments.
- Generation of a symbol table and a constant table with all relevant details.
- Error handling with line numbers for invalid tokens.

The symbol table tracks the name, type, dimensions, frequency, return type, and parameters of each symbol. The constant table records variable names, line numbers, values, and types for constants.

3 Recognized Tokens

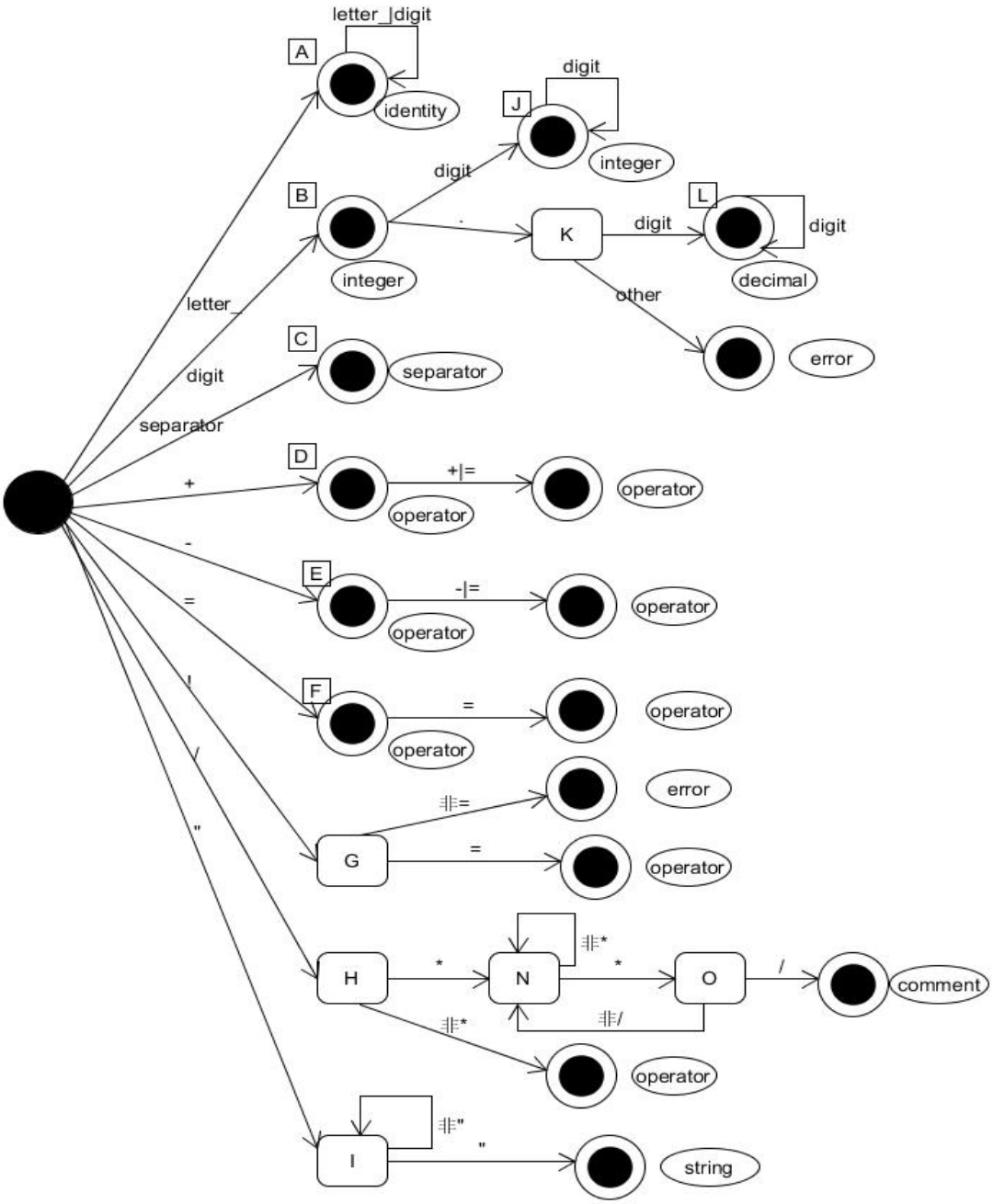
The following tokens are recognized by the scanner:

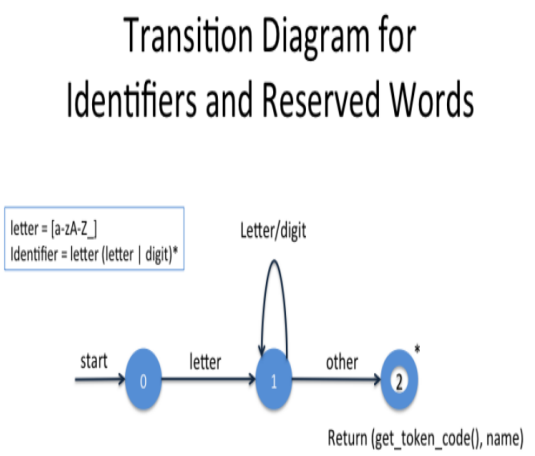
- **Identifiers**: Names of variables and functions.
- **Keywords**: C keywords such as `int`, `float`, `return`, `if`, `while`, etc.
- **Constants**: Integer, float, octal, hexadecimal, character, and string literals.
- **Operators**: Arithmetic, relational, logical, assignment, increment/decrement, bitwise, conditional, and member access operators.

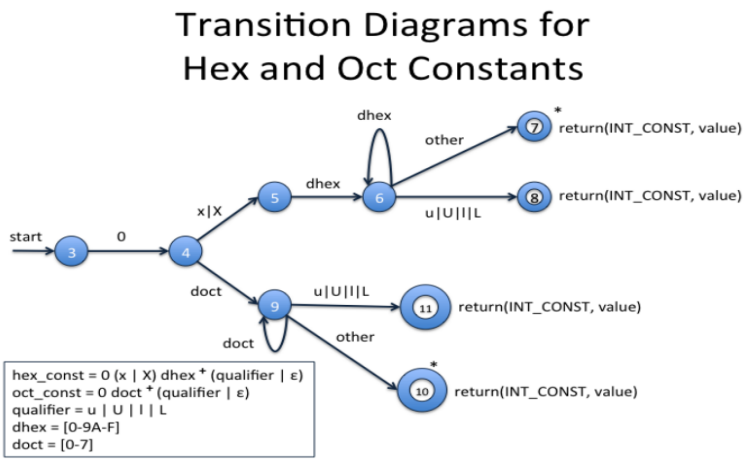
- **Punctuation Symbols**: Parentheses, braces, brackets, commas, and semicolons.
- **Preprocessor Directives**: e.g., `#include`, `#define`.
- **Comments**: Both single-line and multi-line (nested) comments.

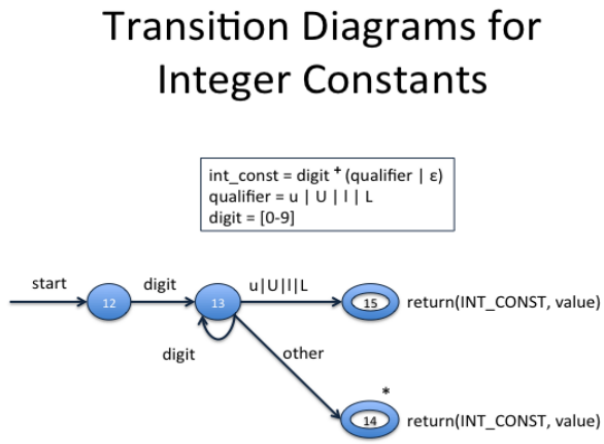
4 DFA Diagram

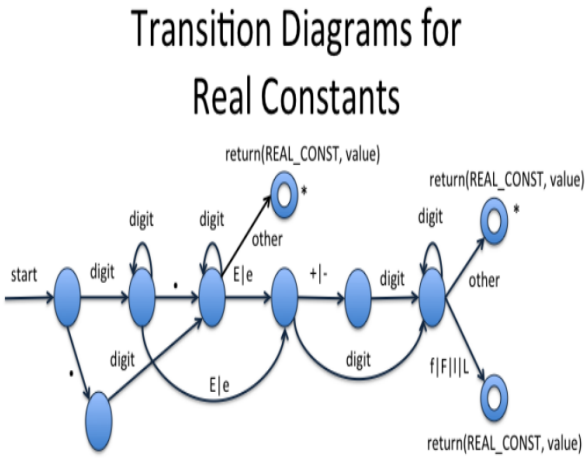
The DFA below illustrates the transition between states for our lexical analyzer-











5 Assumptions

1. Nested multi-line comments are supported but must terminate properly.
2. Invalid tokens generate an error message with the line number.
3. Only standard C keywords are recognized.

4. String literals do not span multiple lines.
5. Preprocessor directives are matched line by line.

6 Handling of Comments, Strings, and Errors

- **Comments**: Single-line comments (`// ...`) are skipped. Multi-line comments (`/* ... */`) support nesting; errors are reported for unterminated comments.
- **Strings**: Enclosed in double quotes; escape characters are supported.
- **Errors**: Invalid characters are reported with their line numbers.

7 Methodology

The implementation process followed these steps:

1. Designing regular expressions for tokens.
2. Implementing the lexer using `LEX/Flex`.
3. Compiling the `.l` file with `flex` and linking with a C compiler.
4. Running test programs to generate token streams, symbol tables, and constant tables.
5. Recording errors such as invalid tokens and unterminated strings.

8 Results

Test Case 1

Input:

```
int nums[5] = {1, 2, 3, 4, 5};
char *name = "Lexical";
```

Output:

Lexical Analysis Output

Recognized Tokens and Errors

```
[line 1] TYPE          : int
[line 1] IDENT         : nums
[line 1] PUNCT         : [5]
[line 1] OP            : =
[line 1] PUNCT         : {
[line 1] NUMBER        : 1
[line 1] PUNCT         : ,
[line 1] NUMBER        : 2
[line 1] PUNCT         : ,
[line 1] NUMBER        : 3
[line 1] PUNCT         : ,
[line 1] NUMBER        : 4
[line 1] PUNCT         : ,
[line 1] NUMBER        : 5
[line 1] PUNCT         : }
[line 1] PUNCT         : ;
[line 2] TYPE          : char
[line 2] OP            : *
[line 2] IDENT         : name
[line 2] OP            : =
[line 2] STRING        : "Lexical"
[line 2] PUNCT         : ;
```

Symbol Table

Name	Type	Dimensions	Frequency	Return Type	Parameters Lists in Function call
nums	int	[5]	1	-	-
name	char	-	1	-	-

Constant Table

Variable Name	Line Number	Value	Type
-	1	1	int
-	1	2	int
-	1	3	int
-	1	4	int
-	1	5	int
-	2	"Lexical"	string

Figure 1: Output for Test Case 1

Test Case 2

Input:

```
int main() {
    int a = 10;
    float b = 2.5;
    char c = 'x';
    return a + b;
}
```

Output:

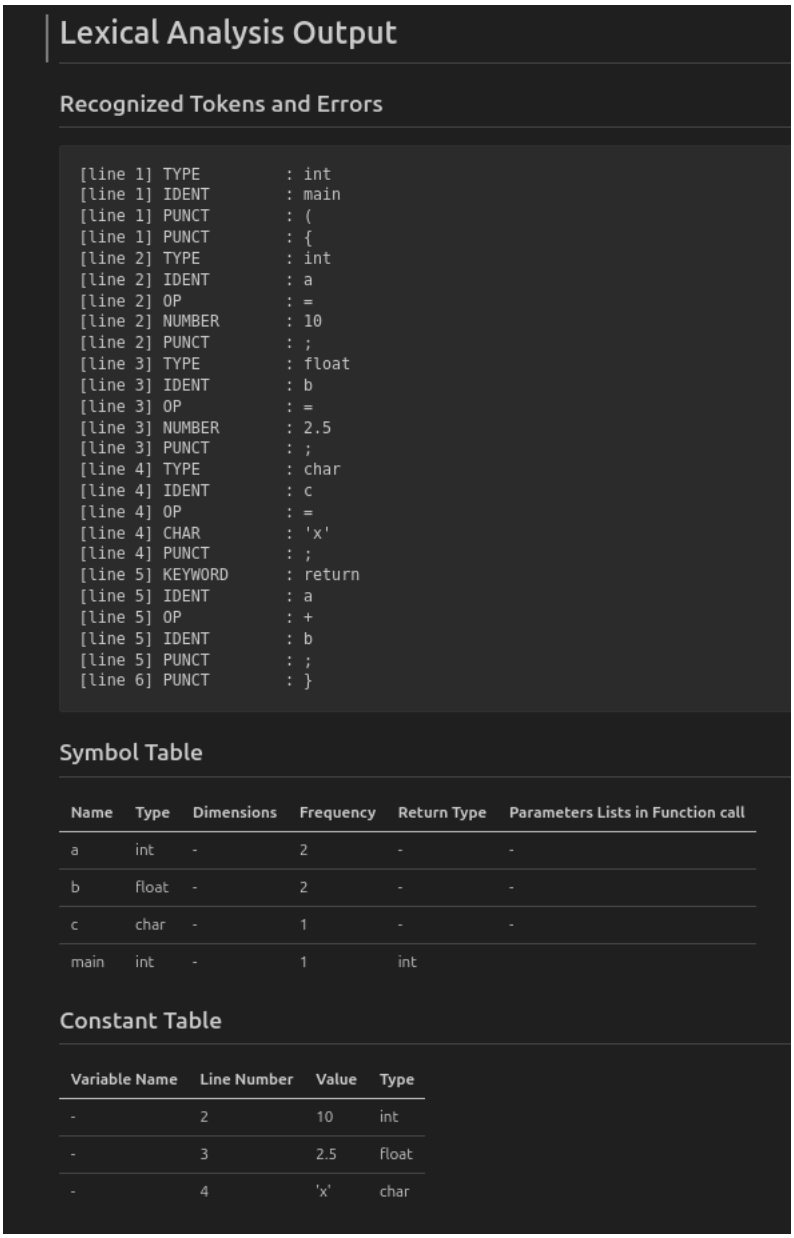


Figure 2: Output for Test Case 2

Test Case 3

Input:

```
// single line comment
int main() {
    /* multi
       line
       comment */
    return 0;
}
```

Output:

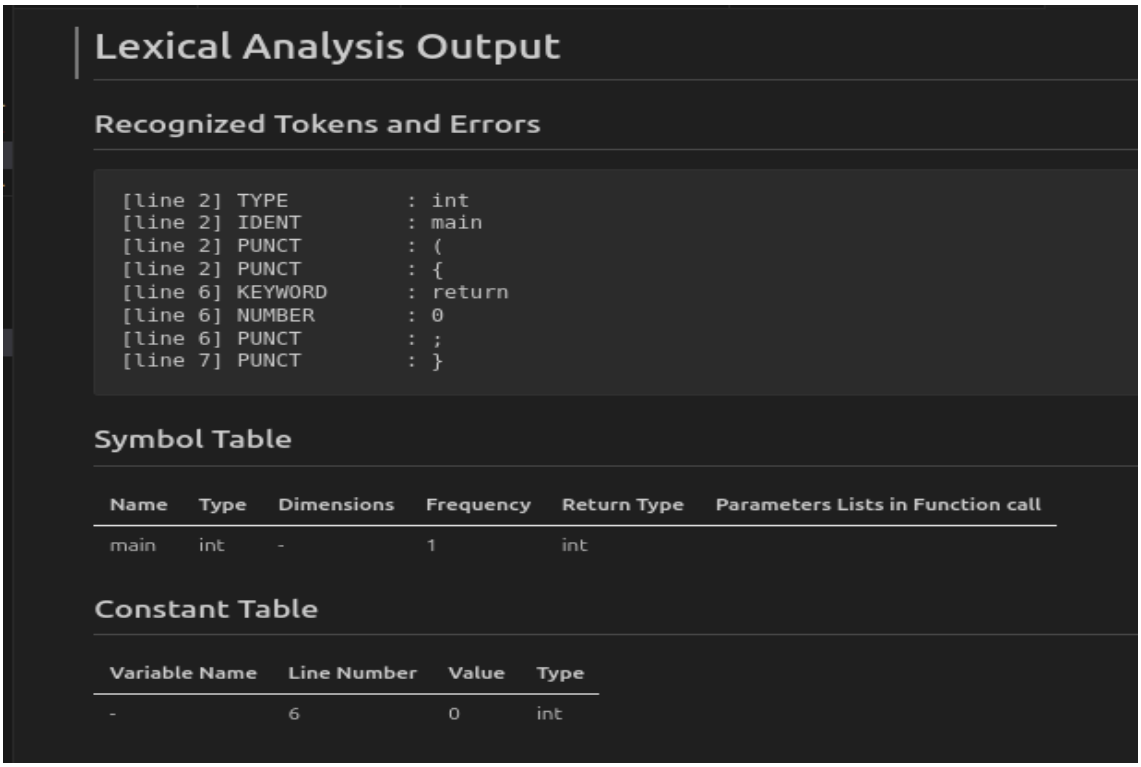


Figure 3: Output for Test Case 3

Test Case 4

Input:

```
int main() {
    string s = "Hello; // unterminated string
    @illegal = 5;     // invalid token
    return 0;
}
```

Output:

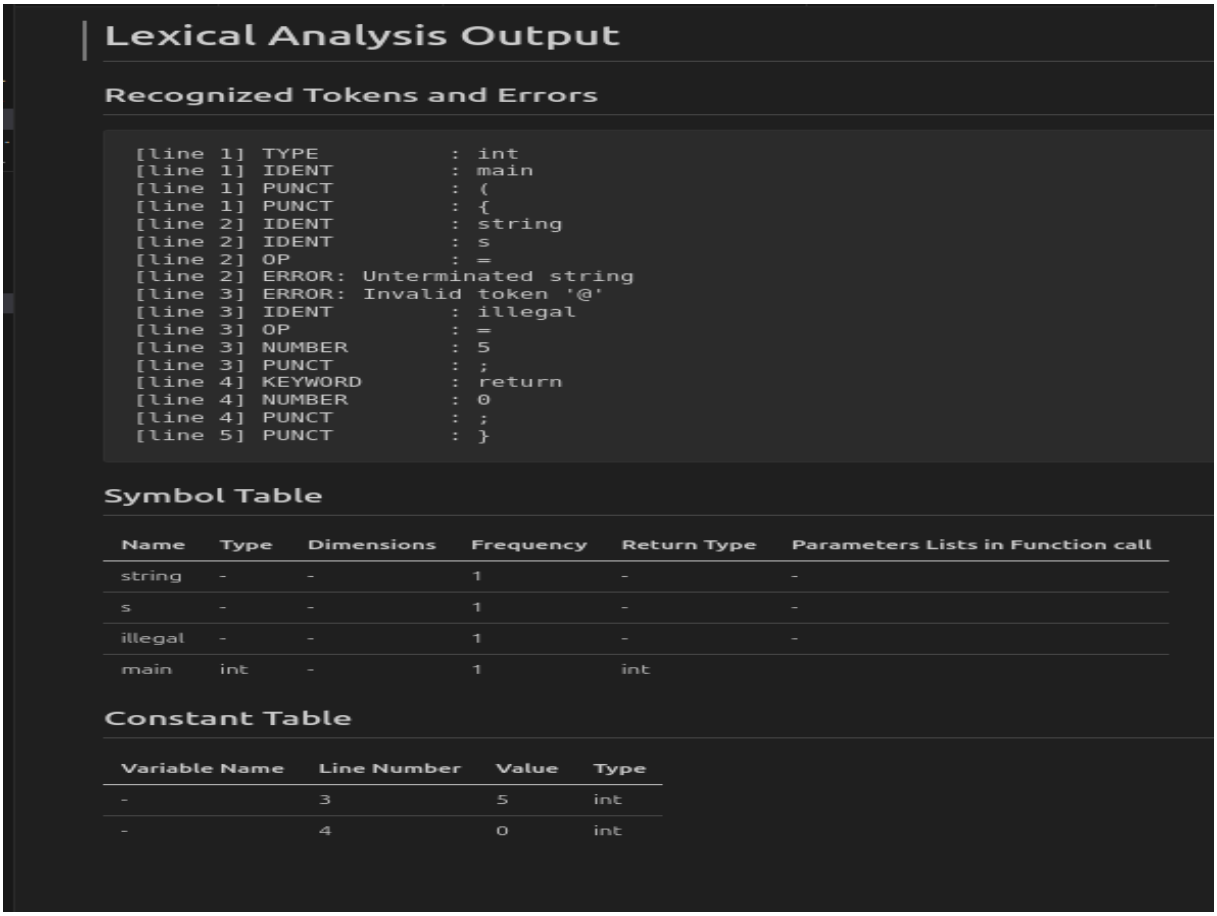


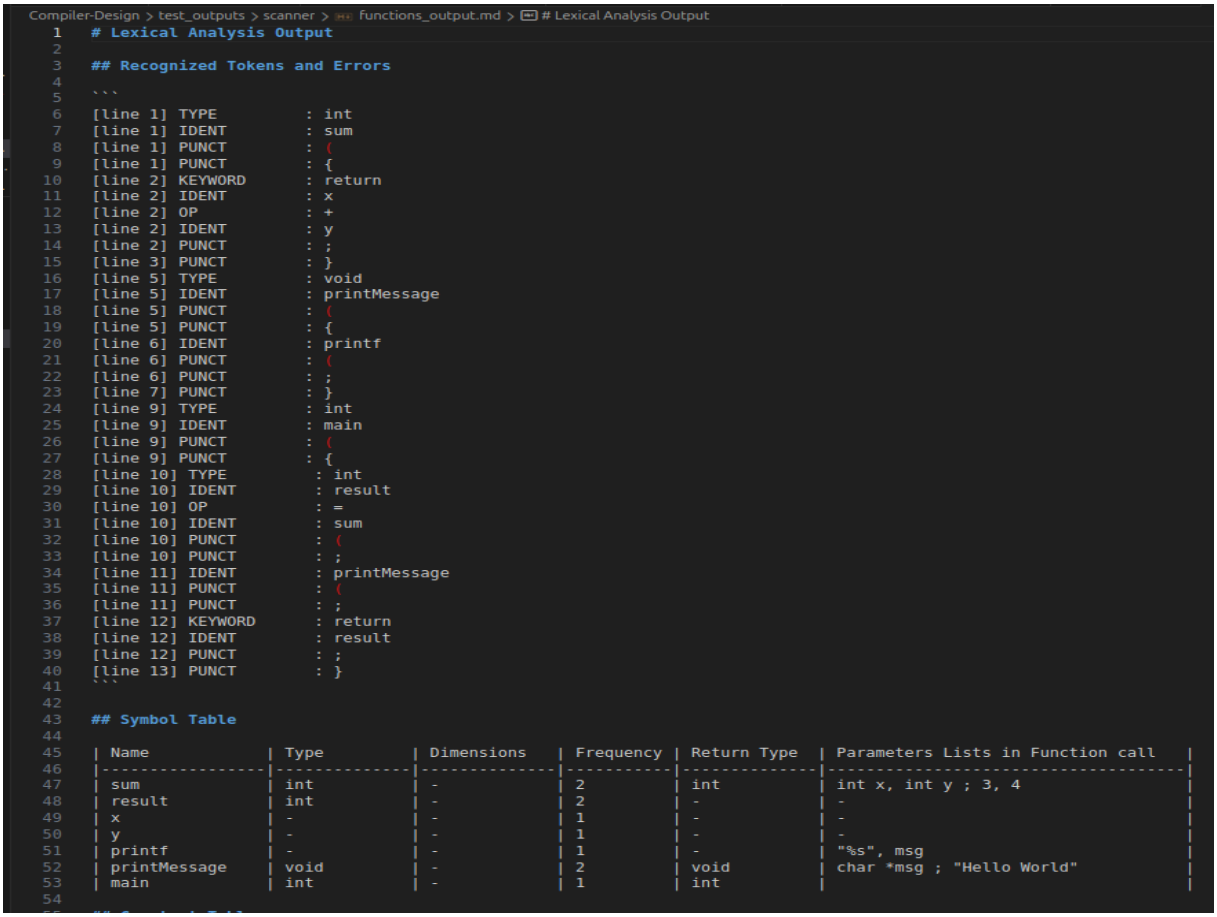
Figure 4: Output for Test Case 4

Test Case 5

Input:

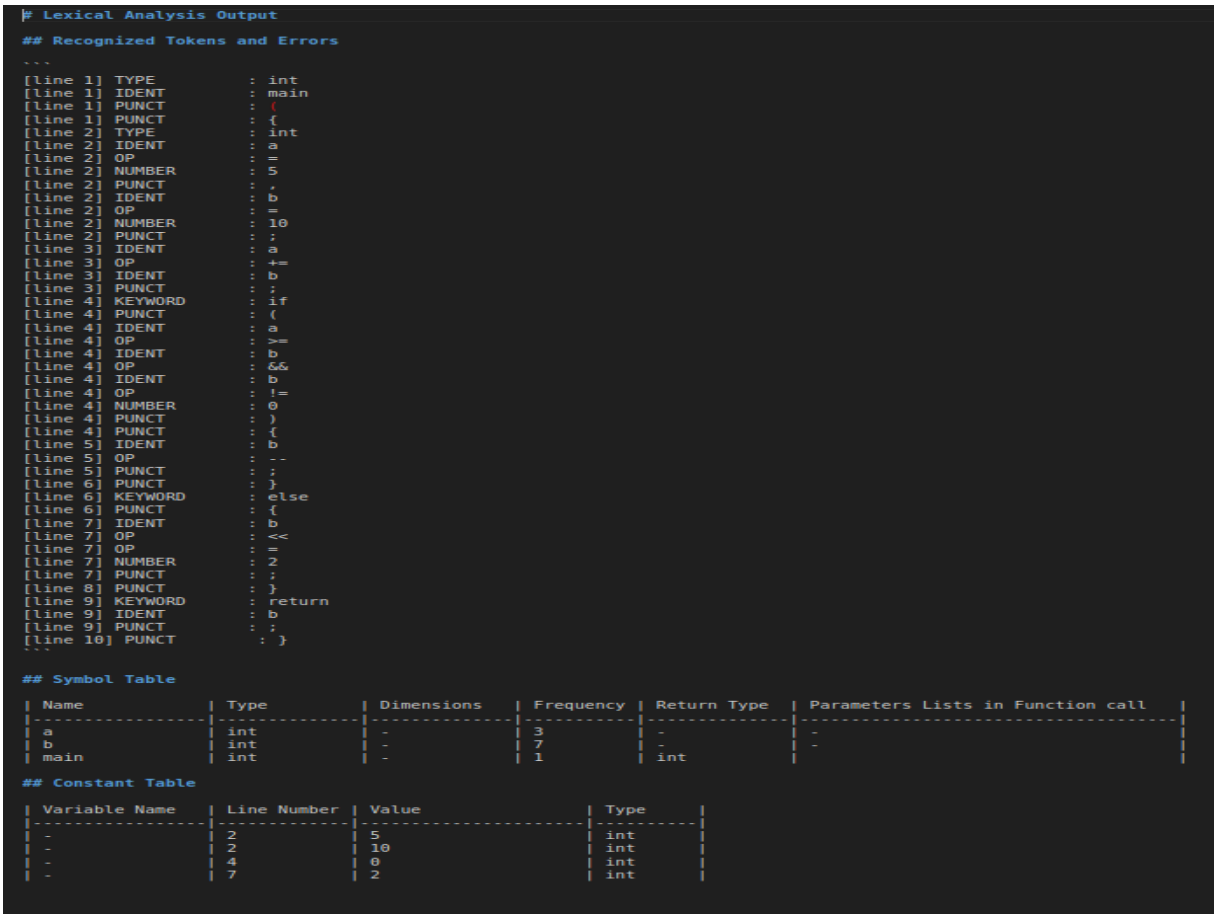
```
int sum(int x, int y) {
    return x + y;
}

void printMessage(char *msg) {
    printf("%s", msg);
}

int main() {
    int result = sum(3, 4);
    printMessage("Hello World");
    return result;
}
```

Output:



Compiler-Design > test_outputs > scanner > functions_output.md > # Lexical Analysis Output

```
# Lexical Analysis Output

## Recognized Tokens and Errors

```
[line 1] TYPE         : int
[line 1] IDENT        : sum
[line 1] PUNCT        : (
[line 1] PUNCT        : {
[line 2] KEYWORD      : return
[line 2] IDENT        : x
[line 2] OP           : +
[line 2] IDENT        : y
[line 2] PUNCT        : ;
[line 3] PUNCT        : }
[line 5] TYPE         : void
[line 5] IDENT        : printMessage
[line 5] PUNCT        : (
[line 5] PUNCT        : {
[line 6] IDENT        : printf
[line 6] PUNCT        : (
[line 6] PUNCT        : ;
[line 7] PUNCT        : }
[line 9] TYPE         : int
[line 9] IDENT        : main
[line 9] PUNCT        : (
[line 9] PUNCT        : {
[line 10] TYPE         : int
[line 10] IDENT        : result
[line 10] OP           : =
[line 10] IDENT        : sum
[line 10] PUNCT        : (
[line 10] PUNCT        : ;
[line 11] IDENT        : printMessage
[line 11] PUNCT        : (
[line 11] PUNCT        : ;
[line 12] KEYWORD      : return
[line 12] IDENT        : result
[line 12] PUNCT        : ;
[line 13] PUNCT        : }
```

Symbol Table

Name	Type	Dimensions	Frequency	Return Type	Parameters Lists in Function call
sum	int	-	2	int	int x, int y ; 3, 4
result	int	-	2	-	-
x	-	-	1	-	-
y	-	-	1	-	-
printf	-	-	1	-	"%s", msg
printMessage	void	-	2	void	char *msg ; "Hello World"
main	int	-	1	int	

Figure 5: Output for Test Case 5

Test Case 6

Input:

```
int main() {
    int a = 5, b = 10;
    a += b;
    if (a >= b && b != 0) {
        b--;
    } else {
        b <<= 2;
    }
    return b;
}
```

Output:
```



Lexical Analysis Output

Recognized Tokens and Errors

```
[line 1] TYPE          : int
[line 1] IDENT         : main
[line 1] PUNCT         : (
[line 1] PUNCT         : {
[line 2] TYPE          : int
[line 2] IDENT         : a
[line 2] OP            : =
[line 2] NUMBER        : 5
[line 2] PUNCT         : ,
[line 2] IDENT         : b
[line 2] OP            : =
[line 2] NUMBER        : 10
[line 2] PUNCT         : ;
[line 3] IDENT         : a
[line 3] OP            : +=
[line 3] IDENT         : b
[line 3] PUNCT         : ;
[line 4] KEYWORD       : if
[line 4] PUNCT         : (
[line 4] IDENT         : a
[line 4] OP            : >=
[line 4] IDENT         : b
[line 4] OP            : &&
[line 4] IDENT         : b
[line 4] OP            : !=
[line 4] NUMBER        : 0
[line 4] PUNCT         : )
[line 4] PUNCT         : {
[line 5] IDENT         : b
[line 5] OP            : --
[line 5] PUNCT         : ;
[line 6] PUNCT         : }
[line 6] KEYWORD       : else
[line 6] PUNCT         : {
[line 7] IDENT         : b
[line 7] OP            : <<
[line 7] OP            : =
[line 7] NUMBER        : 2
[line 7] PUNCT         : ;
[line 8] PUNCT         : }
[line 9] KEYWORD       : return
[line 9] IDENT         : b
[line 9] PUNCT         : ;
[line 10] PUNCT         : }
```

Symbol Table

Name	Type	Dimensions	Frequency	Return Type	Parameters Lists in Function call
a	int	-	3	-	-
b	int	-	7	-	-
main	int	-	1	int	

Constant Table

Variable Name	Line Number	Value	Type
-	2	5	int
-	2	10	int
-	4	0	int
-	7	2	int

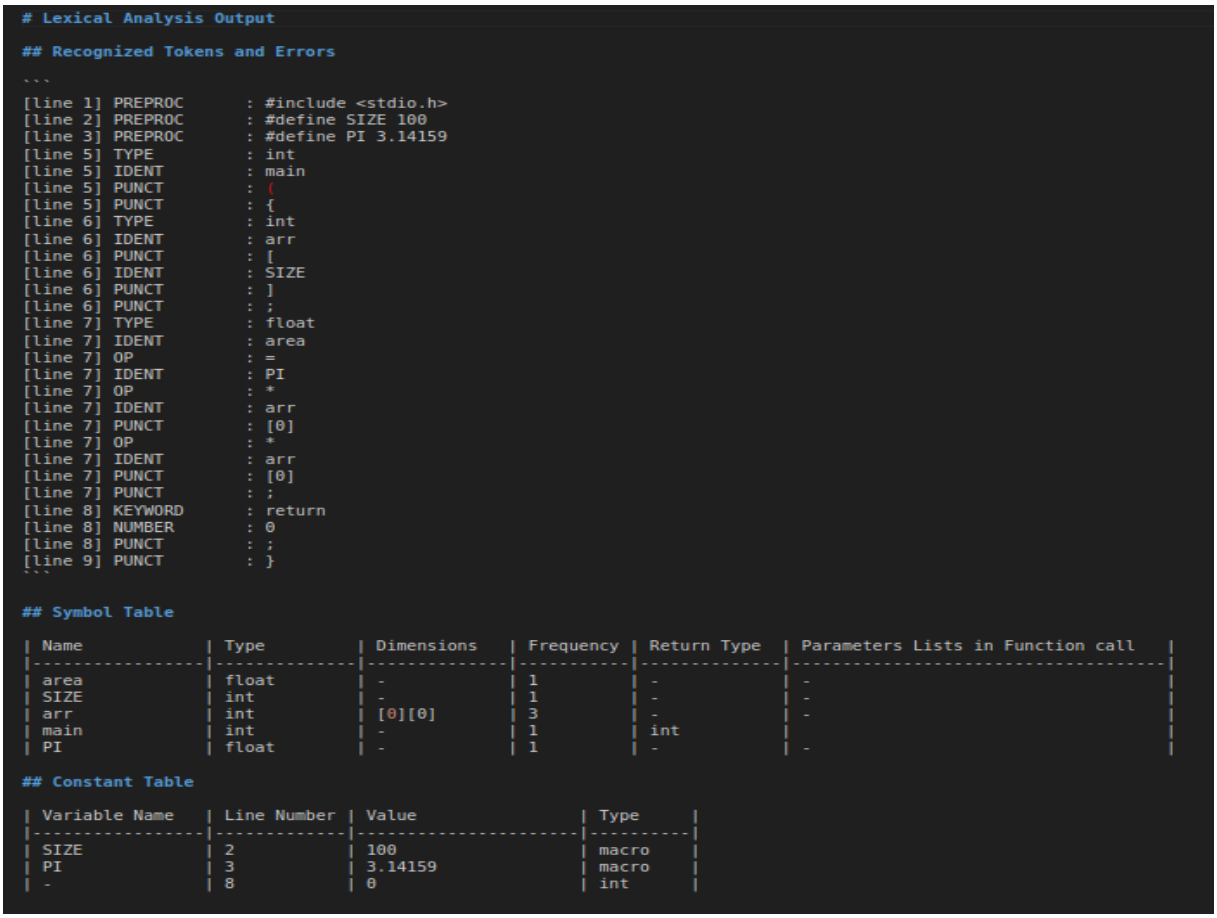
Figure 6: Output for Test Case 6

Test Case 7

Input:

```
#include <stdio.h>
#define SIZE 100
#define PI 3.14159

int main() {
    int arr[SIZE];
    float area = PI * arr[0] * arr[0];
    return 0;
}
```

Output:



Lexical Analysis Output

Recognized Tokens and Errors

```
[line 1] PREPROC      : #include <stdio.h>
[line 2] PREPROC      : #define SIZE 100
[line 3] PREPROC      : #define PI 3.14159
[line 5] TYPE         : int
[line 5] IDENT        : main
[line 5] PUNCT        : (
[line 5] PUNCT        : {
[line 6] TYPE         : int
[line 6] IDENT        : arr
[line 6] PUNCT        : [
[line 6] IDENT        : SIZE
[line 6] PUNCT        : ]
[line 6] PUNCT        : ;
[line 7] TYPE         : float
[line 7] IDENT        : area
[line 7] OP           : =
[line 7] IDENT        : PI
[line 7] OP           : *
[line 7] IDENT        : arr
[line 7] PUNCT        : [0]
[line 7] OP           : *
[line 7] IDENT        : arr
[line 7] PUNCT        : [0]
[line 7] PUNCT        : ;
[line 8] KEYWORD      : return
[line 8] NUMBER       : 0
[line 8] PUNCT        : ;
[line 9] PUNCT        : }
```

Symbol Table

Name	Type	Dimensions	Frequency	Return Type	Parameters Lists in Function call
area	float	-	1	-	-
SIZE	int	-	1	-	-
arr	int	[0][0]	3	-	-
main	int	-	1	int	
PI	float	-	1	-	-

Constant Table

Variable Name	Line Number	Value	Type
SIZE	2	100	macro
PI	3	3.14159	macro
-	8	0	int

Figure 7: Output for Test Case 7

9 Conclusion

In this phase of the mini project, we successfully implemented a lexical analyzer using `LEX/Flex`. The analyzer correctly identifies tokens, generates symbol and constant tables, and reports lexical errors with line numbers. The test cases demonstrate the correctness of token recognition and the robustness of error handling.

Future work will focus on integrating this lexical analyzer with parsing and semantic analysis modules, eventually leading to a complete compiler front-end implementation.