A Report on Compiler Design Lab (CS304) Mini Project

M Vineet Nayak (Roll No: 231CS132)

Prahas GR (Roll No: 231CS142)

Nischal Basavaraju (Roll No: 231CS139)

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

Contents

1 Introduction

1.1 Lexical Analysis

Lexical analysis is the first stage of the compiler where the source code is scanned and broken down into smaller meaningful units. This process converts a sequence of characters into a sequence of tokens, which can then be used by the parser in later stages of compilation. The lexical analyzer also removes whitespace and comments, and reports errors for invalid tokens.

1.2 Tokens & Lexemes

A token is a classified unit of code that represents a category of symbols defined in the programming language. Common token types include keywords, identifiers, constants, operators, and punctuation symbols. Each token is recognized by a specific pattern, such as a regular expression, defined in the lexical analyzer.

A lexeme is the actual sequence of characters in the source code that matches the pattern of a token. For example, in the line int x = 5;, the lexemes are int, x, =, and 5, which correspond to the tokens keyword, identifier, operator, and constant, respectively.

2 Implementation

This part of the project has been implemented using Flex, where regular expressions are written to define the patterns of tokens. The Flex code is then compiled into equivalent C code for execution. The program takes a C source file as input and, for each token encountered, prints its details. At the end of execution, it also generates and displays a symbol table and a constant table.

2.1 Scanner Code

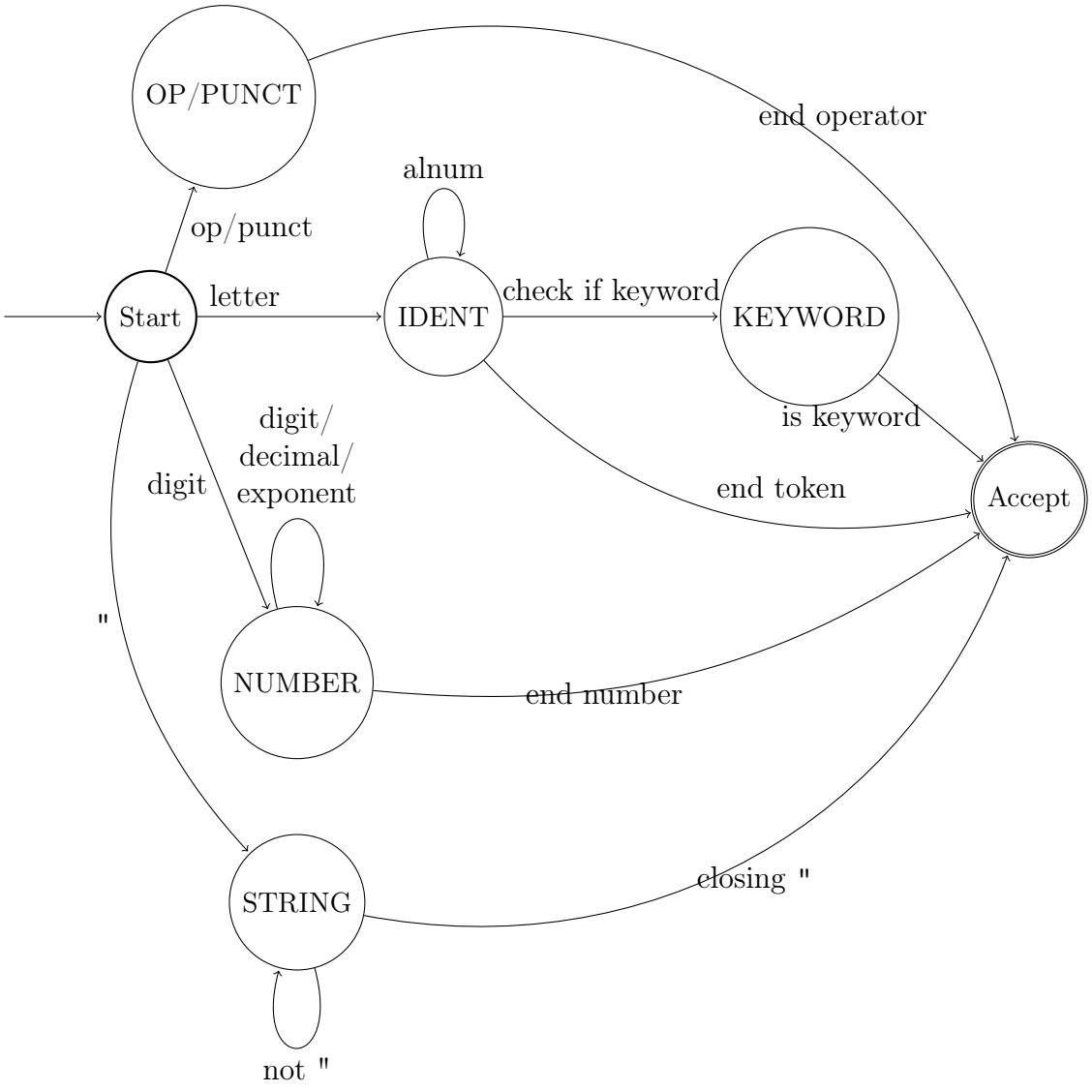
The entire code for the project can be found at www.github.com/vin06eet/Compiler_Design

2.2 List of Recognized Tokens

The following categories of tokens are recognized by the scanner:

- **Identifiers:** variable names, function names
- **Keywords:** if, else, while, for, int, char, return, etc.
- **Constants:** numeric, string, character literals
- **Preprocessor directives:** #include, #define
- **Operators:** +, -, *, /, =, ==
- **Punctuation symbols:** parentheses, braces, commas, semicolons

2.3 DFA Diagram



2.4 Assumptions

The scanner assumes a C89/C90-like subset of the C language and does not fully implement C99/C11 features. It supports both block comments (/* ... */) and line comments (// ...), with additional support for nested comments (non-standard in C). The preprocessor is only partially supported: #define constants are recognized, while other directives are tokenized without further semantic processing. Identifiers in declarations are treated as types only on their first occurrence, and multi-word types (e.g., `unsigned int`) are concatenated. Function parameters are captured as raw strings, with multiple parameters separated by semicolons. Array dimensions are appended only when specified immediately after an identifier (e.g., `arr[10]`). Numeric constants with suffixes (u, l, f, etc.) are recognized, though suffixes do not affect type classification beyond token recognition. The scanner does not handle trigraphs or digraphs. Errors such as unterminated strings, characters, comments,

or invalid tokens are reported, but scanning continues to allow recovery. Whitespace inside array dimension brackets (e.g., `[ 10 ]`) is permitted.

3 Results

Several files were given as input to the scanner, and the outputs of some of the C codes are given below.

3.1 Test Case 1: Simple Program

Input:

```
int main() {
    int a = 100;
    float b = 2.532;
    char c = 'c';
    return a + b;
}
```

Output:

```
[line 1] TYPE          : int
[line 1] IDENT         : main
[line 1] PUNCT         : (
[line 1] PUNCT         : )
[line 1] PUNCT         : {
[line 2] TYPE          : int
[line 2] IDENT         : a
[line 2] OP            : =
[line 2] NUMBER        : 100
[line 2] PUNCT         : ;
[line 3] TYPE          : float
[line 3] IDENT         : b
[line 3] OP            : =
[line 3] NUMBER        : 2.532
[line 3] PUNCT         : ;
[line 4] TYPE          : char
[line 4] IDENT         : c
[line 4] OP            : =
[line 4] CHAR          : 'c'
[line 4] PUNCT         : ;
[line 5] KEYWORD       : return
[line 5] IDENT         : a
[line 5] OP            : +
[line 5] IDENT         : b
[line 5] PUNCT         : ;
[line 6] PUNCT         : }
```

Symbol Table:

Name	Type	Dimension	Frequency	Return Type	Parameters
a	int	-	2	-	-
b	float	-	2	-	-
c	char	-	2	-	-
main	int	-	1	function	

Constant Table:

Variable Name	Line No.	Value	Type
not_allowed	3	5	int
-	4	0	int

3.2 Test Case 2: With Errors

Input:

```
int main() {
    string s = "Howdy; // wrong syntax for string declaration
    @not_allowed = 5;     // invalid identifier, cannot start with @
    return 0;
}
```

Output:

```
[line 1] TYPE          : int
[line 1] IDENT         : main
[line 1] PUNCT         : (
[line 1] PUNCT         : {
[line 2] IDENT         : string
[line 2] IDENT         : s
[line 2] OP            : =
[line 2] ERROR: Unterminated string literal
[line 3] ERROR: Invalid token '@'
[line 3] IDENT         : not_allowed
[line 3] OP            : =
[line 3] NUMBER        : 5
[line 3] PUNCT         : ;
[line 4] KEYWORD       : return
[line 4] NUMBER        : 0
[line 4] PUNCT         : ;

[line 5] PUNCT         : }
```

Symbol Table:

Name	Type	Dimension	Frequency	Return Type	Parameters
string	-	-	1	-	-
s	-	-	1	-	-
not_allowed	-	-	1	-	-
main	int	-	1	function	

Constant Table (excerpt):

Variable Name	Line No.	Value	Type
a	2	100	int
b	3	2.532	float
c	4	'c'	char

Explanation for unusual behaviour: Since the scanner is designed to continue scanning even after detecting an error, some invalid tokens may still be recognized and added to the symbol or constant table. As a result, these tables may contain incorrect entries. However, this limitation can be addressed in later phases of compilation by discarding or ignoring the affected symbol and constant tables whenever an error is invoked.

3.3 Test Case 3: With multiline comments, preprocessors and array usage

Input:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int globalVar = 10;

int main(){
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int b = 1000;
/*
This is to
show how
multiline comments are
handled
*/
return b*b*b;
}
```

Output:

```
[line 1] PREPROC      : #include <stdio.h>
[line 2] PREPROC      : #include <stdlib.h>
[line 3] PREPROC      : #include <stdbool.h>
[line 5] TYPE         : int
[line 5] IDENT        : globalVar
[line 5] OP           : =
[line 5] NUMBER       : 10
[line 5] PUNCT        : ;
[line 7] TYPE         : int
[line 7] IDENT        : main
```

```
[line 7] PUNCT         : (
[line 7] PUNCT         : {
[line 8] TYPE          : int
[line 8] IDENT         : arr
[line 8] PUNCT         : [10]
[line 8] OP            : =
[line 8] PUNCT         : {
[line 8] NUMBER        : 1
[line 8] PUNCT         : ,
[line 8] NUMBER        : 2
[line 8] PUNCT         : ,
[line 8] NUMBER        : 3
[line 8] PUNCT         : ,
[line 8] NUMBER        : 4
[line 8] PUNCT         : ,
[line 8] NUMBER        : 5
[line 8] PUNCT         : ,
[line 8] NUMBER        : 6
[line 8] PUNCT         : ,
[line 8] NUMBER        : 7
[line 8] PUNCT         : ,
[line 8] NUMBER        : 8
[line 8] PUNCT         : ,
[line 8] NUMBER        : 9
[line 8] PUNCT         : ,
[line 8] NUMBER        : 10
[line 8] PUNCT         : }
[line 8] PUNCT         : ;
[line 9] TYPE          : int
[line 9] IDENT         : b
[line 9] OP            : =
[line 9] NUMBER        : 1000
[line 9] PUNCT         : ;
[line 16] KEYWORD       : return
[line 16] IDENT         : b
[line 16] OP            : *
[line 16] IDENT         : b
[line 16] OP            : *
[line 16] IDENT         : b
[line 16] PUNCT         : ;
[line 17] PUNCT         : }
```

Symbol Table:

Name	Type	Dimension	Frequency	Return Type	Parameters
b	int	-	4	-	-
globalVar	int	-	1	-	-
arr	int	[10]	1	-	-
main	int	-	1	int	

Constant Table (excerpt):

Variable Name	Line No.	Value	Type
globalVar	5	10	int
-	8	1	int
-	8	2	int
-	8	3	int
-	8	4	int
-	8	5	int
-	8	6	int
-	8	7	int
-	8	8	int
-	8	0	int
b	9	1000	int