Department of Computer Science and Engineering

Course Code: CSE 420	**Credits: 1.5**
Course Name: Compiler Design	**Semester:** **Spring 2025**

1. Introduction

Lexical analysis is the process of scanning the source program as a sequence of characters and converting them into sequences of tokens. A program that performs this task is called a lexical analyzer or a lexer or a scanner. For example, if a portion of the source program contains int x=5; the scanner would convert in a sequence of tokens like <INT> <ID> <ASSIGNOP> <COST_NUM> <SEMICOLON>. You will also mention the associated symbol where applicable.

In this assignment, we are going to construct a lexical analyzer and a syntax analyzer for a subset of the C language. That means we will perform syntax analysis with a grammar rule containing function implementation in this assignment. To do so, we will build a scanner and a parser with the help of Lex (Flex) and Yacc (Bison).

2. Language

Our chosen subset of the C language has the following characteristics:

- There can be multiple functions. No two functions will have the same name.
- There will be no pre-processing directives like `#include` or `#define.`
- Variables can be declared at suitable places inside a function. Variables can also be declared in the global scope.
- All the operators used in the previous assignment are included. Precedence and associativity rules are as per standard. Although we will ignore consecutive logical operators or consecutive relational operators like, `a && b && c, a < b < c.`
- No `break` statement and `switch-case` statement will be used.

3. Tasks

You have to complete the following tasks in this assignment.

3.1 Identifying Tokens

3.1.1 Keywords

You have to identify the keywords given in Table 1 and print the token in the output file. For example, you will have to print **<IF>** in case you find the keyword **"if"** in the source program.

Keyword	**Token**	**Keyword**	**Token**
if	IF	else	ELSE
for	FOR	while	WHILE
do	DO	break	BREAK
int	INT	char	CHAR
float	FLOAT	double	DOUBLE
void	VOID	return	RETURN
switch	SWITCH	case	CASE
default	DEFAULT	continue	CONTINUE
goto	GOTO	printf	PRINTF

Table 1 : Keyword List

3.1.2 Constants

You have to identify constants using regular expressions.

a. **Integer Literals:** One or more consecutive digits form an integer literal. Type of token will be **CONST_INT**. Note that + or - will not be the part of an integer.
b. **Floating Point Literals:** Numbers like 3.14159, 3.14159E-10, .314159 and 314159E10 will be considered ad floating point constants. In this case, token type will be **CONST_FLOAT**.

3.1.3 Operators and Punctuators

The operator list for the subset of the python language we are dealing with is given in Table 2. A token in the form of **<Type> along with the particular symbol** should be printed in the output log file.

Symbols	Type
+, -	ADDOP
*, /, %	MULOP
++, - -	INCOP
<, >, ==, <=, >=, !=	RELOP
=	ASSIGNOP
&&, ||	LOGICOP
!	NOT
(	LPAREN
)	RPAREN
{	LCURL
}	RCURL
[	LTHIRD
]	RTHIRD
,	COMMA
:	COLON
;	SEMICOLON

Table 2: Operators and Punctuators List

3.1.4 Identifiers

Identifiers are names given to entities, such as variables, functions, structures etc. An identifier can only have alphanumeric characters (a-z, A-Z, 0-9) and underscore (_). The first character of an identifier can only contain the alphabet (a-z, A-Z) or underscore (_). For any identifier encountered in the input file you have to print the token **<ID> along with the symbol**.

3.1.5 White Space and Newlines

You have to capture the white spaces and newlines in the input file, but no actions needed regarding this.

3.1.6 Line count

You have to count the number of lines in the source program. Refer to the sample input output for better understanding.

3.2 Syntax Analysis

For the syntax analysis part you have to do the following tasks:

- Incorporate the grammar given in the file “**Lab1_C_Syntax_Analyzer_Grammar.pdf**” along with this document in your Yacc file.
- You are given a modified lex file names **“lex_analyzer.l”** to use it with your Yacc file. Try to understand the syntax and tokens used.
- Use a `SymbolInfo` pointer to pass information from lexical analyzer to parser when needed. For example, if your lexical analyzer detects an identifier, it will return a token named ID and pass its symbol and type using a SymbolInfo pointer as the attribute of the token. On the other hand in the case of semicolons, it will only return the token as the parser does not need any more information. You can implement this by redefining the type of yylval (YYSTYPE) in parser and associate yylval with new type in the scanner. See the skeleton file for example.
- Handle any ambiguity in the given grammar (For example, if-else). Your Yacc file should compile with 0 conflicts.
- When a grammar matches the input from the C code, it should print the matching rule in the correct order in an output file (log.txt). For each grammar rule matched with some portion of the code, print the rule along with the relevant portion of the code.

4. Input

The input will be a text file containing a c source program. File name will be given from the command line.

5. Output

In this assignment, there will be one output file. The output file should be named as **<Your_student_ID>_log.txt**. This will contain all the tokens as well the line number where it was found along with the **matching grammar rules, and corresponding segments of source code** as instructed in the previous sections. Print the line count at the end of the log file.

For more clarification about input-output check the supplied sample I/O files given in the lab folder. You are highly encouraged to produce output exactly like the sample one.

6. Submission

1. In your local machine create a new folder whose **name is your student id.**
2. Put the lex file named as **<your_student_id>.l**, the Yacc file **<your_student_id>.y** and a script named **script.sh** (modifying with your own filenames), the **symbol_info.h** file, a suitable input file names **input.txt** in a folder **named with your student id**. **DO NOT** put any output file, generated lex.yy.c file or any executable file in this folder.
3. Compress the folder in a **.zip file** which should be **named as your student id**.
4. Submit the .zip file.

Failure to follow these instructions will result in a penalty.