RV COLLEGE OF ENGINEERING ®
BENGALURU-560059
(Autonomous Institution Affiliated to VTU, Belagavi)

# "Lexical Analysis"

**Report**

**Compiler Design**

**(18IS54)**

Submitted By

Ankit Kumar Singh (1RV18IS007)

**Under the Guidance of**

B. K. Srinivas

Asst. Professor

**in partial fulfillment for the award of degree of**

**Bachelor of Engineering**

**in**

**INFORMATION SCIENCE AND ENGINEERING**

**2020-21**

## Abstract:

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

### Phases of Compiler

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.

The phases are as below:

### Analysis

1. Lexical Analysis:
2. Parsing:
3. Semantic Analysis:
4. Intermediate Code Generation:

### Synthesis

1. Code Optimization:
2. Code Generation:

### Objectives:

This project aims to undertake a sequence of experiments to design and implement various phases of a compiler for the C programming language. Following constructs will be handled by the mini-compiler :

1. Data Types: int, char data types with all its sub-types. Syntax : `int a=3;`
2. Comments: Single line and multiline comments,
3. Keywords: `char, else, for, if, int, long, return, short, signed, struct, unsigned, void, while, main`
4. Identification of valid identifiers used in the language,
5. Looping Constructs: It will support nested for and while loops. Syntax: `int i; for(i=0;i<n;i++){ }  int x; while(x<10){ … x++}`
6. Conditional Constructs: `if...else-if...else` statements,
7. Operators: ADD(+), MULTIPLY(*), DIVIDE(/), MODULO(%), AND(&), OR(|)
8. Delimiters: SEMICOLON(;), COMMA(,)
9. Structure construct of the language, Syntax: `struct pair{ int a; int b};`
10. Function construct of the language, Syntax: `int func(int x)`
11. Support of nested conditional statement,
12. Support for a 1-Dimensional array. Syntax : `char s[20];`

# Contents:


| | Page No. |
|---|---|
| 1. Introduction | |
| a. Lexical Analyzer | 4 |
| b. Flex Script | 5 |
| c. C Program | 5 |
| 2. Design of Programs | |
| a. Code | 6 |
| b. Explanation | 11 |
| 3. Test Cases | |
| i. Without Errors | 13 |
| ii. With Errors | 22 |
| 4. Implementation | 28 |
| 5. Results / Future Work | 29 |
| 6. References | 29 |

# Introduction

## Lexical Analysis

The Lexical Analyzer is the first phase of the Analysis (front end) stage of a compiler. In layman's terms, the Lexical Analyzer (or Scanner) scans through the input source program character by character, and identifies 'Lexemes' and categorizes them into 'Tokens'. These 'tokens' are represented as a symbol table, and are given as input to the Parser (second phase of the front end of a compiler).

### Tokens

Tokens are essentially just a group of characters which have some meaning or relation when put together.

The Lexical Analyzer detects these tokens with the help of 'Regular Expressions'. While writing the Lexical Analyzer, we have to specify rules for each Token type using Regular Expression. These rules are used to check whether a certain group of characters fall under a given token category or not.

An example, in this case, would be an 'Identifier' token. We specify the rules for an identifier as follows: Any string of characters, that start with an _ or an alphabet, followed by any number of _'s, alphabets or numbers. The regular expression for Identifiers is `{S}({S}|{D})*` where `S` is `[a-zA-z_]` and `D` is `[0-9]`.

### Lexemes

Lexemes are instances of Tokens. An example would be ' long int ', which is a Lexeme of 'Keyword' Token.

### Symbol Table

A symbol table is generated in the Lexical Analyzer stage, which is basically a table with the columns 'Symbol', 'Type' and 'Token ID'. The symbol is the Lexime itself, the 'Type' is the token category and the 'Token ID' is a unique ID given to a token, which is used in the parser stage. There are no duplicate entries in a symbol table. Each symbol is recorded only once, even if there are multiple instances.

A Lexical Analyzer is internally implemented based on the concept of FSM's (Finite State Machines). A DFA (Deterministic Finite State Automata) is internally built for each Token based on the Regular Expression provided. This is used to identify Lexemes and categorize them into Tokens.

## Flex Script

The script written by us is a program that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

The structure of the flex script is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

## C Program

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned into account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input.

The script also has an option to take standard input instead of taking input from a file.

## Design of Program

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


	struct symboltable
	{
		char name[100];
		char type[100];
		int length;
	}ST[1001];

	struct constanttable
	{
		char name[100];
		char type[100];
		int length;
	}CT[1001];

	int hash(char *str)
	{
		int value = 0;
		for(int i = 0 ; i < strlen(str) ; i++)
		{
			value = 10*value + (str[i] - 'A');
			value = value % 1001;
			while(value < 0)
				value = value + 1001;
		}
		return value;
	}

	int lookupST(char *str)
	{
		int value = hash(str);
		if(ST[value].length == 0)
		{
			return 0;
		}
		else if(strcmp(ST[value].name,str)==0)
```

```
        {
            return 1;
        }
        else
        {
            for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            if(strcmp(ST[i].name,str)==0)
            {
                return 1;
            }
        }
        return 0;
      }
}

int lookupCT(char *str)
{
        int value = hash(str);
        if(CT[value].length == 0)
            return 0;
        else if(strcmp(CT[value].name,str)==0)
            return 1;
        else
        {
            for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
            {
                if(strcmp(CT[i].name,str)==0)
                {
                    return 1;
                }
            }
            return 0;
        }
}

void insertST(char *str1, char *str2)
{
        if(lookupST(str1))
        {
           return;
      }
```

```
    else
    {
        int value = hash(str1);
        if(ST[value].length == 0)
        {
            strcpy(ST[value].name,str1);
            strcpy(ST[value].type,str2);
            ST[value].length = strlen(str1);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            if(ST[i].length == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(ST[pos].name,str1);
        strcpy(ST[pos].type,str2);
        ST[pos].length = strlen(str1);
    }
}

void insertCT(char *str1, char *str2)
{
    if(lookupCT(str1))
        return;
    else
    {
        int value = hash(str1);
        if(CT[value].length == 0)
        {
            strcpy(CT[value].name,str1);
            strcpy(CT[value].type,str2);
            CT[value].length = strlen(str1);
            return;
        }
```

```
            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
            {
                if(CT[i].length == 0)
                {
                    pos = i;
                    break;
                }
            }

            strcpy(CT[pos].name,str1);
            strcpy(CT[pos].type,str2);
            CT[pos].length = strlen(str1);
        }
}

void printST()
{
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(ST[i].length == 0)
            {
                continue;
            }

            printf("%s\t%s\n",ST[i].name, ST[i].type);
        }
}

void printCT()
{
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(CT[i].length == 0)
                continue;

            printf("%s\t%s\n",CT[i].name, CT[i].type);
        }
}
```

```
%}

DE "define"
IN "include"

operator
[[<][=]|[>][=]|[=][=]|[!][=]|[>]|[<]|[\|][\|]|[&][&]|[\!]|[=]|[\^]|[\+][=]|
[\-][=]|[\*][=]|[\/][=]|[\%][=]|[\+][\+]|[\-][\-]|[\+]|[\-]|[\*]|[\/]|[\%]|
[&]|[\|]|[~]|[<][<]|[>][>]]


%%
\n   {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|"
"|"\t"] {printf("%s \t-Pre Processor directive\n",yytext);} //Matches
#include<stdio.h>
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]
{printf("%s \t-Macro\n",yytext);} //Matches macro
\/\/(.*) {printf("%s \t- SINGLE LINE COMMENT\n", yytext);}

\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/  {printf("%s \t- MULTI LINE
COMMENT\n", yytext);}
[ \n\t] ;
; {printf("%s \t- SEMICOLON DELIMITER\n", yytext);}
, {printf("%s \t- COMMA DELIMITER\n", yytext);}
\{ {printf("%s \t- OPENING BRACES\n", yytext);}
\} {printf("%s \t- CLOSING BRACES\n", yytext);}
\( {printf("%s \t- OPENING BRACKETS\n", yytext);}
\) {printf("%s \t- CLOSING BRACKETS\n", yytext);}
\[ {printf("%s \t- SQUARE OPENING BRACKETS\n", yytext);}
\] {printf("%s \t- SQUARE CLOSING BRACKETS\n", yytext);}
\: {printf("%s \t- COLON DELIMITER\n", yytext);}
\\ {printf("%s \t- FSLASH\n", yytext);}
\. {printf("%s \t- DOT DELIMITER\n", yytext);}
auto|break|case|char|const|continue|default|do|double|else|enum|extern|floa
t|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|sw
itch|typedef|union|unsigned|void|volatile|while|main/[\(|"
"|\{|;|:|"\n"|"\t"] {printf("%s \t- KEYWORD\n", yytext); insertST(yytext,
"KEYWORD");}
\"[^\n]*\"/[;|,|\)] {printf("%s \t- STRING CONSTANT\n", yytext);
insertCT(yytext,"STRING CONSTANT");}
\'[A-Z|a-z]\'/[;|,|\)|:] {printf("%s \t- Character CONSTANT\n", yytext);
```

```
insertCT(yytext,"Character CONSTANT");}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[ {printf("%s \t- ARRAY IDENTIFIER\n",
yytext); insertST(yytext, "IDENTIFIER");}

{operator}/[a-z]|[0-9]|;|" "|[A-Z]|\(|\"|\'|\)|\n|\t {printf("%s \t-
OPERATOR\n", yytext);}

[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
{printf("%s \t- NUMBER CONSTANT\n", yytext); insertCT(yytext, "NUMBER
CONSTANT");}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
{printf("%s \t- Floating CONSTANT\n", yytext); insertCT(yytext, "Floating
CONSTANT");}
[A-Za-z_][A-Za-z_0-9]*/["
"|;|,|\(|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\.|\{|\^|\t] {printf("%s \t-
IDENTIFIER\n", yytext); insertST(yytext, "IDENTIFIER");}


(.?) {
           if(yytext[0]=='#')
           {
                printf("Error in Pre-Processor directive at line no.
%d\n",yylineno);
           }
         else if(yytext[0]=='/')
         {
                printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
           }
           else if(yytext[0]=='"')
           {
                printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
           }
           else
           {
                printf("ERROR at line no. %d\n",yylineno);
           }
           printf("%s\n", yytext);
           return 0;
}

%%
```

```
int main(int argc , char **argv){


printf("===================================================================
=\n");

      int i;
      for (i=0;i<1001;i++){
            ST[i].length=0;
            CT[i].length=0;
      }

      yyin = fopen(argv[1],"r");
      yylex();

      printf("\n\nSYMBOL TABLE\n\n");
      printST();
      printf("\n\nCONSTANT TABLE\n\n");
      printCT();
}

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

# Explanation:

**Definition Section:**

In the definition section of the program, all necessary header files were included. Apart from that structure declaration for both the symbol table and constant table were made. In order to convert a string of the source program into a particular integer value a hash function was written that takes a string as input and converts it into a particular integer value. Standard table operations like look-up and insert were also written. Linear Probing hashing technique was used to implement the symbol table i.e. if there is a collision, then after the point of collision, the table is searched linearly in order to find an empty slot. Functions to print the symbol table and constant table was also written.

**Rules section:**

In this section rules related to the specification of C language were written in the form of valid regular expressions. E.g. for a valid C identifier the regex written was `[A-Za-z_][A-Za-z_0-9]*` which means that a valid identifier needs to start with an alphabet or underscore followed by 0 or more occurrences of alphabets, numbers or underscore. In order to resolve conflicts we used a lookahead method of scanner by which a scanner decides whether an expression is a valid token or not by looking at its adjacent character. E.g. in order to differentiate between comments and division operator lookahead characters of a valid operator were also given in the regular expression to resolve a conflict. If none of the patterns matched with the input, we said it is a lexical error as it does not match with any valid pattern of the source language. Each character/pattern along with its token class was also printed.

**C code section:**

In this section both the tables (symbol and constant) were initialised to `0` and `yylex()` function was called to run the program on the given input file. After that, both the symbol table and constant table were printed in order to show the result.

The flex script recognises the following classes of tokens from the input:

- Pre-processor instructions
    - Statements processed : `#include<stdio.h>, #define var1 var2`
    - Token generated : Preprocessor Directive

- Errors in pre-processor instructions
    - Statements processed : `#include<stdio.h>, #include<stdio.?`
    - Token generated : Error with line number

- Single-line comments
    - Statements processed : `//...........`
    - Token generated : Single Line Comment

- Multi-line comments
    - Statements processed : `/*...........*/, /*.../*    */`
    - Token generated : Multi Line Comment

- Errors for unmatched comments
  - Statements processed : `/*..........`
  - Token generated : Error with line number

- Errors for nested comments
  - Statements processed : `/*....../*....*/     */`
  - Token generated : Error with line number

- Parentheses (all types)
  - Statements processed : `(..), {..}, [..]`
  - Token generated : Parenthesis

- Operators
- Literals (integer, float, string)
  - Statements processed : `int, float, char`
  - Tokens generated : Keywords

- Errors for unclean integers and floating point numbers
  - Statements processed : `123rf`
  - Tokens generated : Error

- Errors for incomplete strings
  - Statements processed : `char a[]= ”abcd`
  - Tokens generated : Error Incomplete string and line number

- Keywords
  - Statements processed : `if, else, void, while, do, int, float, break` and so on.
  - Tokens generated : Keyword

- Identifiers
  - Statements processed : a, abc, a_b, a12b4
  - Tokens generated : Identifier

- Errors for any invalid character used that is not in the C character set.
  - Keywords accounted for:

    ```
    auto, break, case, char, const, continue, default, do, double,
    else, enum, extern, float, for, goto, if, int, long, register,
    return, short, signed, sizeof, static, struct, switch,
    typedef, union, unsigned, void, volatile, while, main.
    ```

# Test Cases

## Valid Test Cases:

```
(base) ankitz@Ankitz:~/Downloads/Intermediate Code Generation/Lexical Analysis$ bash run.sh
Running: 8


Running TestCase 1
====================================================================
#include<stdio.h>       -Pre Processor directive
int     - KEYWORD
main    - KEYWORD
(       - OPENING BRACKETS
)       - CLOSING BRACKETS
{       - OPENING BRACES
int     - KEYWORD
n       - IDENTIFIER
,       - COMMA DELIMITER
i       - IDENTIFIER
;       - SEMICOLON DELIMITER
char    - KEYWORD
ch      - IDENTIFIER
;       - SEMICOLON DELIMITER
//Character Datatype    - SINGLE LINE COMMENT
for     - KEYWORD
(       - OPENING BRACKETS
i       - IDENTIFIER
=       - OPERATOR
0       - NUMBER CONSTANT
;       - SEMICOLON DELIMITER
i       - IDENTIFIER
<       - OPERATOR
n       - IDENTIFIER
;       - SEMICOLON DELIMITER
i       - IDENTIFIER
++      - OPERATOR
)       - CLOSING BRACKETS
{       - OPENING BRACES
if      - KEYWORD
(       - OPENING BRACKETS
i       - IDENTIFIER
<       - OPERATOR
10      - NUMBER CONSTANT
)       - CLOSING BRACKETS
{       - OPENING BRACES
int     - KEYWORD
x       - IDENTIFIER
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
}       - CLOSING BRACES
/*
    This File Contains Test cases about Datatypes,Keyword,Identifier,Nested For and while loop,
    Conditional Statement,Single line Comment,MultiLine Comment etc.*/  - MULTI LINE COMMENT
}       - CLOSING BRACES


SYMBOL TABLE

i       IDENTIFIER
n       IDENTIFIER
x       IDENTIFIER
for     KEYWORD
char    KEYWORD
ch      IDENTIFIER
if      KEYWORD
int     KEYWORD
main    KEYWORD


CONSTANT TABLE

10      NUMBER CONSTANT
0       NUMBER CONSTANT
```

```
Running TestCase 2
====================================================================
#include<stdio.h>       -Pre Processor directive
int     - KEYWORD
main    - KEYWORD
(       - OPENING BRACKETS
)       - CLOSING BRACKETS
{       - OPENING BRACES
//Program to add 2 numbers and increment by 1   - SINGLE LINE COMMENT
int     - KEYWORD
a       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
3       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
=       - OPERATOR
{       - OPENING BRACES
1       - NUMBER CONSTANT
,       - COMMA DELIMITER
2       - NUMBER CONSTANT
}       - CLOSING BRACES
;       - SEMICOLON DELIMITER
a       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
2       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
=       - OPERATOR
a       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
1       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
+       - OPERATOR
a       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
2       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
;       - SEMICOLON DELIMITER
a       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
2       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
++      - OPERATOR
;       - SEMICOLON DELIMITER
printf  - IDENTIFIER
(       - OPENING BRACKETS
"%d"    - STRING CONSTANT
,       - COMMA DELIMITER
a       - ARRAY IDENTIFIER
[       - SQUARE OPENING BRACKETS
2       - NUMBER CONSTANT
]       - SQUARE CLOSING BRACKETS
)       - CLOSING BRACKETS
;       - SEMICOLON DELIMITER
return  - KEYWORD
0       - NUMBER CONSTANT
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES


SYMBOL TABLE

a       IDENTIFIER
return  KEYWORD
int     KEYWORD
main    KEYWORD
printf  IDENTIFIER


CONSTANT TABLE

"%d"    STRING CONSTANT
0       NUMBER CONSTANT
1       NUMBER CONSTANT
2       NUMBER CONSTANT
3       NUMBER CONSTANT
```

```
Running TestCase 3
====================================================================
#include<stdio.h>       -Pre Processor directive
int      - KEYWORD
main     - KEYWORD
(        - OPENING BRACKETS
)        - CLOSING BRACKETS
{        - OPENING BRACES
int      - KEYWORD
a        - IDENTIFIER
=        - OPERATOR
5        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
while    - KEYWORD
(        - OPENING BRACKETS
a        - IDENTIFIER
>        - OPERATOR
0        - NUMBER CONSTANT
)        - CLOSING BRACKETS
{        - OPENING BRACES
printf   - IDENTIFIER
(        - OPENING BRACKETS
"Hello world"    - STRING CONSTANT
)        - CLOSING BRACKETS
;        - SEMICOLON DELIMITER
a        - IDENTIFIER
--       - OPERATOR
;        - SEMICOLON DELIMITER
}        - CLOSING BRACES
}        - CLOSING BRACES


SYMBOL TABLE

a        IDENTIFIER
int      KEYWORD
main     KEYWORD
printf   IDENTIFIER
while    KEYWORD


CONSTANT TABLE

"Hello world"    STRING CONSTANT
0        NUMBER CONSTANT
5        NUMBER CONSTANT
```

```
Running TestCase 4
====================================================================
#include<stdio.h>       -Pre Processor directive
int      - KEYWORD
main     - KEYWORD
(        - OPENING BRACKETS
)        - CLOSING BRACKETS
{        - OPENING BRACES
int      - KEYWORD
a        - IDENTIFIER
=        - OPERATOR
2        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
printf   - IDENTIFIER
(        - OPENING BRACKETS
"%d"     - STRING CONSTANT
,        - COMMA DELIMITER
a        - IDENTIFIER
)        - CLOSING BRACKETS
;        - SEMICOLON DELIMITER
a        - IDENTIFIER
++       - OPERATOR
;        - SEMICOLON DELIMITER
int      - KEYWORD
b        - IDENTIFIER
=        - OPERATOR
4        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
int      - KEYWORD
c        - IDENTIFIER
=        - OPERATOR
3        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
}        - CLOSING BRACES


SYMBOL TABLE

a        IDENTIFIER
b        IDENTIFIER
c        IDENTIFIER
int      KEYWORD
main     KEYWORD
printf   IDENTIFIER


CONSTANT TABLE

"%d"     STRING CONSTANT
2        NUMBER CONSTANT
3        NUMBER CONSTANT
4        NUMBER CONSTANT
```

## Invalid Test Cases:

```
Running TestCase 5
=====================================================================
#include<stdio.h>       -Pre Processor directive
struct  - KEYWORD
student         - IDENTIFIER
{       - OPENING BRACES
int     - KEYWORD
rollNum         - IDENTIFIER
;       - SEMICOLON DELIMITER
int     - KEYWORD
marks   - IDENTIFIER
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
student1        - IDENTIFIER
;       - SEMICOLON DELIMITER
int     - KEYWORD
main    - KEYWORD
(       - OPENING BRACKETS
)       - CLOSING BRACKETS
{       - OPENING BRACES
int     - KEYWORD
a       - IDENTIFIER
=       - OPERATOR
1       - NUMBER CONSTANT
,       - COMMA DELIMITER
b       - IDENTIFIER
=       - OPERATOR
0       - NUMBER CONSTANT
;       - SEMICOLON DELIMITER
student1        - IDENTIFIER
.       - DOT DELIMITER
rollNum         - IDENTIFIER
=       - OPERATOR
1       - NUMBER CONSTANT
;       - SEMICOLON DELIMITER
student1        - IDENTIFIER
.       - DOT DELIMITER
marks   - IDENTIFIER
=       - OPERATOR
90      - NUMBER CONSTANT
;       - SEMICOLON DELIMITER
if      - KEYWORD
(       - OPENING BRACKETS
a       - IDENTIFIER
>=      - OPERATOR
1       - NUMBER CONSTANT
&&      - OPERATOR
a       - IDENTIFIER
<=      - OPERATOR
10      - NUMBER CONSTANT
)       - CLOSING BRACKETS
b       - IDENTIFIER
++      - OPERATOR
;       - SEMICOLON DELIMITER
else    - KEYWORD
{       - OPENING BRACES
b       - IDENTIFIER
--      - OPERATOR
;       - SEMICOLON DELIMITER
ERR_UNMATCHED_COMMENT at line no. 21
/


SYMBOL TABLE

struct  KEYWORD
a       IDENTIFIER
b       IDENTIFIER
if      KEYWORD
int     KEYWORD
student IDENTIFIER
main    KEYWORD
else    KEYWORD
rollNum IDENTIFIER
marks   IDENTIFIER
student1        IDENTIFIER


CONSTANT TABLE

10      NUMBER CONSTANT
90      NUMBER CONSTANT
0       NUMBER CONSTANT
1       NUMBER CONSTANT
```

```
Running TestCase 6
===================================================================
Error in Pre-Processor directive at line no. 1
#


SYMBOL TABLE



CONSTANT TABLE



Running TestCase 7
===================================================================
// Implicit Error that our Language doesn't support      - SINGLE LINE COMMENT
#include<stdio.h>        -Pre Processor directive
int      - KEYWORD
main     - KEYWORD
(        - OPENING BRACKETS
)        - CLOSING BRACKETS
{        - OPENING BRACES
char     - KEYWORD
ERROR at line no. 6
@


SYMBOL TABLE

char     KEYWORD
int      KEYWORD
main     KEYWORD


CONSTANT TABLE



Running TestCase 8
===================================================================
#define A 10     -Macro
int      - KEYWORD
a        - IDENTIFIER
=        - OPERATOR
5        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER


SYMBOL TABLE

a        IDENTIFIER
int      KEYWORD


CONSTANT TABLE

5        NUMBER CONSTANT
```

## Implementation

The Regular Expressions for most of the features of C are fairly straightforward. However, a few features require a significant amount of thought, such as:

- **The Regex for Identifiers:** The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.
- **Multiline comments should be supported:** To implement it a proper regular expression was written along with that lookahead character set for operators were thought so to resolve conflict with the division operator.
- **Literals:** Different regular expressions have been implemented in the code to support all kinds of literals, i.e integers, floats, strings, etc.
- **Error Handling for Incomplete String:** Open and close quote missing, both kind  of errors have been handled in the rules written in the script.
- **Error Handling for Unmatched Comments:** This has been handled by adding lookahead characters to operator regular expression. If there is an unmatched comment then it does not match with any of the patterns in the rule. Hence it goes to default state which in turn throws an error.
- **Error Handling for unclean integer constant:** This has been handled by adding appropriate lookahead characters for integer constant. E.g. int a = 786rt, is rejected as the integer constant should never follow an alphabet.

At the end of the token recognition, the lexer prints a list of all the identifiers and constants present in the program. We use the following technique to implement this:

- We maintain two structures one for symbol table and other for constant tableone corresponding to identifiers and other to constants.
- Four functions have been implemented `lookupST( )`, `lookupCT( )`, these functions return true if the identifier and constant respectively are already present in the table. `InsertST( )`, `InsertCT( )` help to insert identifier/constant in the appropriate table.
- Whenever we encounter an identifier/constant, we call the `insertST()` or `insertCT()` function which in turns calls `lookupST( )` or `lookupCT( )` and adds it to the corresponding structure.
- In the end, in the main`( )` function, after yylex returns, we call `printST( )` and `printCT( )`, which in turn prints the list of identifiers and constants in a proper format.

## Results:

1. Token --- Token Class
2. Symbol Table:

   Token --- Attribute
3. Constant Table

   Token --- Attribute

## References:


- Compilers Principles, Techniques and Tool by Alfred V.Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- http://dinosaur.compilertools.net/lex/index.html
- http://www.csd.uwo.ca/~moreno/CS447/Lectures/Lexical.html/node11.html