Lexical Analyzer for the C Language

National Institute of Technology Karnataka, Surathkal

Date: 29th January 2020

Submitted To: Dr. P. Santhi Thilagam

Group members:

1. Aditya Rastogi 171CO105
2. Animesh Kumar 171CO108
3. Manas Gupta 171CO122

Abstract

Compiler: A compiler is a software program that compiles program source code files into an executable program. In simple terms, a compiler is a computer program that changes the language in which programs are written into instructions that a computer can use. It is included as part of the integrated development environment IDE with most programming software packages. The compiler takes source code files that are written in a high-level language, such as C, C++, or Java, and compiles the code into a low-level language, such as machine code or assembly code. This code is created for a specific processor type, such as an Intel Pentium or PowerPC. The program can then be recognized by the processor and run from the operating system.

Phases of a compiler: The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of the source program, and feeds its output to the next phase of the compiler. The phases are:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation

Features implemented in this project:

This project contains the implementation of the lexical analyzer phase of the C compiler. In our lexical analyzer we have implemented the following functionalities:

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

List of Figures and Tables

Introduction

Lexical Analysis

Lexical Analysis is the first process of the compiler. The input to a lexical analyzer is a high-level language program, such as a 'C' program in the form of a sequence of characters. The output is a sequence of tokens, which is passed to the parser for syntax analysis. The blanks, tabs, newlines, and comments from the source program are removed. It keeps track of line numbers and associates error messages from various parts of a compiler with line numbers.

Flex Script

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. Instead of writing a scanner from scratch, we only need to identify the vocabulary of a particular language (e.g., C language), write a specification of patterns using regular expressions (e.g., DIGIT [0-9]), and FLEX constructs a scanner for us. FLEX workflow depicted as:

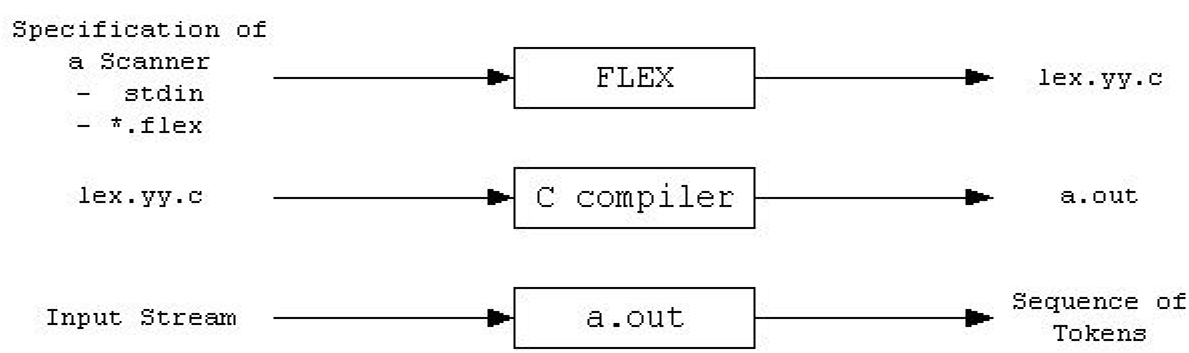

Figure 1

First, FLEX reads a specification of a scanner either from an input file *.lex, or from standard input, and it generates as output a C source file *lex.yy.c*. Then, *lex.yy.c* is compiled and linked with the "-lfl" library to produce an executable *a.out*. Finally, *a.out* analyzes its input stream and transforms it into a sequence of tokens.

The format of the input file contains three sections. These sections are separated by a line with "%%" symbol.

Format of a FLEX Program

Definition section

%%

Rules section

%%

User code section

Definition section: The definition section contains the declaration of variables, regular definitions, manifest constants.

The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets.

This section contains C statements and additional functions. We can also compile these functions separately and load them with the lexical analyzer.

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 in the 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 the Program

Code of scanner.l file

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



    struct symboltable
    {
        char name[100];
        char type[100];
        int length;
    }ST[1007];

    struct constanttable
    {
        char name[100];
        char type[100];
        int length;
    }CT[1007];

```

```
int hash(char *str)
{
    int value = 0;
    for(int i = 0 ; i < strlen(str) ; i++)
    {
        value = 10*value + (str[i] - 'A');
        value = value % 1007;
        while(value < 0)
            value = value + 1007;
    }
    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)%1007)
            {
                if(strcmp(ST[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)%1007)

        {

            if(ST[i].length == 0)

            {

                pos = i;

                break;

            }

        }


        strcpy(ST[pos].name,str1);

        strcpy(ST[pos].type,str2);

        ST[pos].length = strlen(str1);

    }

}

void printST()

{

    for(int i = 0 ; i < 1007 ; i++)
```

```
        {
            if(ST[i].length == 0)
            {
                continue;
            }

            printf("%s\t%s\n",ST[i].name, ST[i].type);
        }
    }
    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)%1007)
            {
                if(strcmp(CT[i].name,str)==0)
                {
                    return 1;
                }
```

```
            }
            return 0;
        }
    }
    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)%1007)
            {
                if(CT[i].length == 0)
```

```
                {
                    pos = i;
                    break;
                }
            }

            strcpy(CT[pos].name,str1);
            strcpy(CT[pos].type,str2);
            CT[pos].length = strlen(str1);
        }
    }
    void printCT()
    {
        for(int i = 0 ; i < 1007 ; i++)
        {
            if(CT[i].length == 0)
                continue;

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

%}
```

```
DEF "define"

INC "include"

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




%%

\n   {yylineno++;}

\/\/(.*) {printf("%s \t- same line comment\n", yytext);}

([#][" "]*({INC})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|"
"|"\t"] {printf("%s \t-Preprocessor statement\n",yytext);}  //Matches
preprocessor directives

[ \n\t] ;

([#][" "]*({DEF})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]
{printf("%s \t-Definition\n",yytext);} //Matches definition

\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/  {printf("%s \t- multiple line
comment\n", yytext);}

, {printf("%s \t- comma separator\n", yytext);}

; {printf("%s \t- semicolon\n", yytext);}

\} {printf("%s \t- closing curly brackets\n", yytext);}

\] {printf("%s \t- closing square brackets\n", yytext);}

\( {printf("%s \t- opening brackets\n", yytext);}

\) {printf("%s \t- closing brackets\n", yytext);}

\. {printf("%s \t- dot\n", yytext);}
```

```
\[ {printf("%s \t- opening square brackets\n", yytext);}

\: {printf("%s \t- colon\n", yytext);}

\\ {printf("%s \t- forward slash\n", yytext);}

\{ {printf("%s \t- opening curly brackets\n", yytext);}

auto|break|default|printf|case|void|scanf|const|do|double|long|enum|float|
sizeof|for|goto|char|if|int|register|continue|return|short|else|typedef|st
atic|unsigned|struct|switch|signed|union|extern|while|volatile|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");}

~ {return 0;}

{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- Integer Constant\n", yytext); insertCT(yytext, "Integer
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: incomplete string at line no. %d\n",yylineno);
        }
        else if(yytext[0]=='#')
        {
            printf("ERROR: Pre-Processor directive at line no.
%d\n",yylineno);
        }
        else if(yytext[0]=='/')
        {
            printf("ERROR: unmatched comment 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<1007;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();

printf("*************************************************************\n");

}


int yywrap(){

    return 1;
```

```
}
```

--------------------------------------END OF CODE--

Explanation

The following is the explanation for the lexical analyzer, the explanation is divided into three sections based on the same format as **FLEX**(Fast Lexical analyzer generator).

Definition Section:

In the definition section of the program, all necessary header files were included. Apart from that, the structure declaration for both the symbol table and a constant table was 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. Functions to print the symbol table and constant table were also written.

Rules section:

In this section, rules for the lexical analyzer were specified. **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 occurrence of alphabets, numbers or underscore. For resolving conflicts, we used the lookahead method of the scanner by which a scanner decides whether an expression is a valid token or not by looking at its adjacent character. **E.g.**, to differentiate between modulus operator and format specifier for string, we used lookahead characters for valid operators which were also implemented in the regular expressions to resolve conflict.

NOTE: 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 initialized to 0 and yylex() function was called to run the program on the given input file. After that, both the symbol table and the constant table were printed in order to show the result.

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

- **Same-line comment**
  - Statements processed : //...........
  - Token generated: Same Line Comment
- **Pre-processor statement**
  - Statements processed : #include<stdio.h>, #define var1 var2
  - Token generated: Preprocessor statement
- **Errors in pre-processor instructions**
  - Statements processed : #include<stdio.h>, #include<stdio.?
  - Token generated: Error with line number
- **Multiple line comment**
  - Statements processed : /*...........*/, /*.../*...*/
  - Token generated: Multiple line comment
- **Parentheses (all types)**
  - Statements processed : (..), {..}, [..]
  - Token generated : Parenthesis
- **Comma Seperator**
  - Statements processed : ‘,’
  - Token generated : comma seperator

- **Semicolon, dot, colon and forward slash**
  - Statements processed : ‘;’ , ’.’ , ‘:’ and ‘\’
  - Token generated : semicolon, dot, colon and forward slash

- **Brackets**
  - Statements processed : ..),,, ..(..., …[..., …]..., …{..., …}...
  - Token generated : opening brackets and closing brackets (respective types of brackets)
- **Literals (integer, float, string)**
  - Statements processed : int, float, char
  - Tokens generated : Keyword
- **Errors for incomplete strings**
  - Statements processed : char a[]= ”abcd
  - Tokens generated : ERROR: incomplete string at line no.
- **Errors for unmatched comments**
  - Statements processed : /*..........
  - Token generated : ERROR: unmatched comment at line no.
- **Errors for nested comments**
  - Statements processed : /*....../*....*/....*/
  - Token generated : Error with line number
- **Errors for unclean integers and floating point numbers**
  - Statements processed : 123rf
  - Tokens generated : Error with line number
- **Keywords**

  Statements processed : if, else, void, while, do, int, float, break and so on.

   ○ Tokens generated : Keyword

- **Identifiers**

   ○ Statements processed : a3, abc, t_b, a12b4

   ○ Tokens generated: Identifier

- **Errors for any invalid character used that is not in C character set.**

   ○ Keywords accounted for:

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

DFA for Regular Expressions

1. **Multi-line comments**

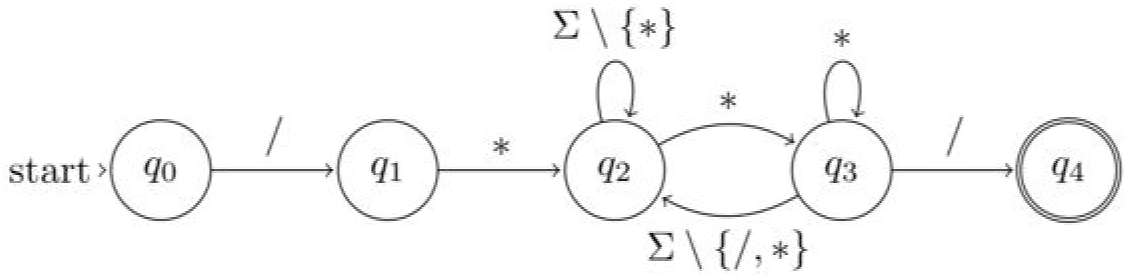

Figure 2.1

2. **Single line comments**

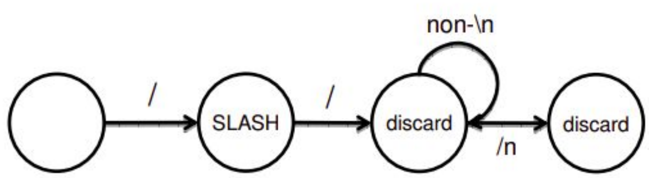

Figure 2.2

3. **String Literals**

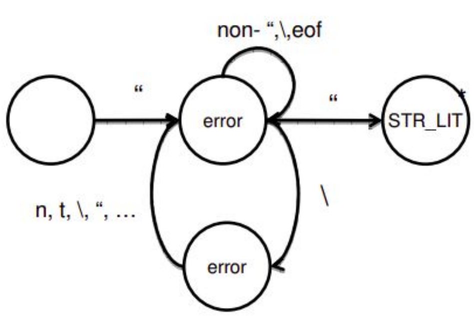

Figure 2.3

4. Integer and Float Constants

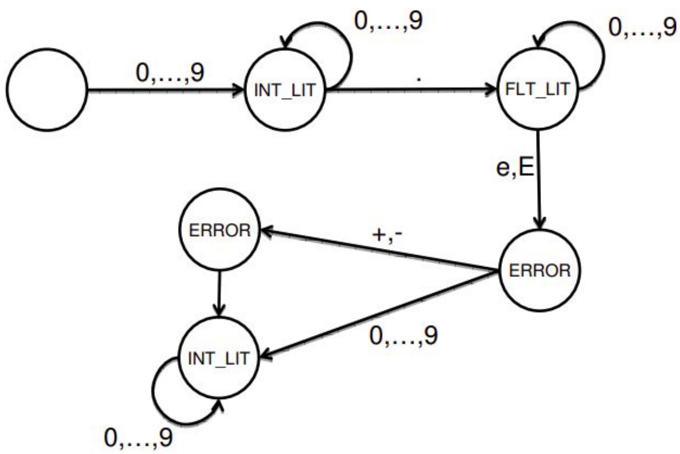

Figure 2.4

Test Cases

The lexical analyzer was tested against the test cases of C programs. The test C programs with their outputs are depicted below:

Test Cases Without Errors

TEST CASE 1: Without Error Code:

```
#include<stdio.h>


int main(){

   char ch;

   int n,i;

   for (i=0;i<n;i++){

       if(i<10)

       {

           int ite;

           while(ite<10){

               ite++;

               printf("%d",ite);

           }

       }

   }

}
```

Output: figure 3.1

```
Running TestCase 1
*************************************************************
#include<stdio.h>       -Preprocessor statement
int      - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
char     - Keyword
ch       - Identifier
;        - semicolon
int      - Keyword
n        - Identifier
,        - comma separator
i        - Identifier
;        - semicolon
for      - Keyword
(        - opening brackets
i        - Identifier
=        - Operator
0        - Integer Constant
;        - semicolon
i        - Identifier
<        - Operator
n        - Identifier
;        - semicolon
i        - Identifier
++       - Operator
)        - closing brackets
{        - opening curly brackets
if       - Keyword
(        - opening brackets
i        - Identifier
<        - Operator
10       - Integer Constant
)        - closing brackets
{        - opening curly brackets
int      - Keyword
ite      - Identifier
```

```
;        - semicolon
while    - Keyword
(        - opening brackets
ite      - Identifier
<        - Operator
10       - Integer Constant
)        - closing brackets
{        - opening curly brackets
ite      - Identifier
++       - Operator
;        - semicolon
printf   - Keyword
(        - opening brackets
"%d"     - String Constant
,        - comma separator
ite      - Identifier
)        - closing brackets
;        - semicolon
}        - closing curly brackets
}        - closing curly brackets
}        - closing curly brackets
}        - closing curly brackets


Symbol Table

char     Keyword
i        Identifier
n        Identifier
for      Keyword
main     Keyword
ch       Identifier
while    Keyword
if       Keyword
int      Keyword
ite      Identifier
printf   Keyword


Constant Table

"%d"     String Constant
10       Integer Constant
```

TEST CASE 2: Without Error Code:

```c
#include<stdio.h>


int main()

{

   int arr[3] = { 1, 2 };

   arr[2] = arr[1] + arr[2];

   arr[2]++;/*checking for multi

   line

   comment*/


   printf("%d", arr[2]);


   return 0;

}
```

OUTPUT: figure 3.2

```
Running TestCase 4
************************************************************
#include<stdio.h>       -Preprocessor statement
int      - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
int      - Keyword
arr      - Array Identifier
[        - opening square brackets
3        - Integer Constant
]        - closing square brackets
=        - Operator
{        - opening curly brackets
1        - Integer Constant
,        - comma separator
2        - Integer Constant
}        - closing curly brackets
;        - semicolon
arr      - Array Identifier
[        - opening square brackets
2        - Integer Constant
]        - closing square brackets
=        - Operator
arr      - Array Identifier
[        - opening square brackets
1        - Integer Constant
]        - closing square brackets
+        - Operator
arr      - Array Identifier
[        - opening square brackets
2        - Integer Constant
]        - closing square brackets
;        - semicolon
arr      - Array Identifier
[        - opening square brackets
2        - Integer Constant
]        - closing square brackets
++       - Operator
;        - semicolon
```

```
2        - Integer Constant
]        - closing square brackets
++       - Operator
;        - semicolon
/*checking for multi
         line
         comment*/       - multiple line comment
printf   - Keyword
(        - opening brackets
"%d"     - String Constant
,        - comma separator
arr      - Array Identifier
[        - opening square brackets
2        - Integer Constant
]        - closing square brackets
)        - closing brackets
;        - semicolon
return   - Keyword
0        - Integer Constant
;        - semicolon
}        - closing curly brackets


Symbol Table

main     Keyword
int      Keyword
printf   Keyword
arr      Identifier
return   Keyword


Constant Table

"%d"     String Constant
0        Integer Constant
1        Integer Constant
2        Integer Constant
3        Integer Constant
************************************************************
```

TEST CASE 3: Without Error Code:

```c
#include <stdio.h>

int main()

{

   i = 10;

   //simple program

}
```

OUTPUT: figure 3.3

```
Running TestCase 15
**************************************************************
#include <stdio.h>      -Preprocessor statement
int     - Keyword
main    - Keyword
(       - opening brackets
)       - closing brackets
{       - opening curly brackets
i       - Identifier
=       - Operator
10      - Integer Constant
;       - semicolon
//simple program        - same line comment
}       - closing curly brackets


Symbol Table

i       Identifier
main    Keyword
int     Keyword


Constant Table

10      Integer Constant
**************************************************************
```

Test Cases With Errors

TEST CASE 1: With Error Code:

```
#include<stdio.h>


int main() {

   char @hello;

   @hello = 'c';

}
```

OUTPUT: figure 3.4

```
Running TestCase 14
*************************************************************
#include<stdio.h>       -Preprocessor statement
int      - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
char     - Keyword
ERROR: at line no. 5
@


Symbol Table

char     Keyword
main     Keyword
int      Keyword


Constant Table
```

TEST CASE 2: With Error Code:

```
#include<stdio.h>


int main()

{

   int a = 2;

   printf("%d",a);

   a++;

   int b = 4;

   int c = 3;

   /*nested

   comment/*error*/*/

   int b = 8;

   int c = 3;

   int d = c*(a+b);

   a--;

}
```

OUTPUT: figure 3.5

```
Running TestCase 6
*****************************************************************
#include<stdio.h>       -Preprocessor statement
int      - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
int      - Keyword
a        - Identifier
=        - Operator
2        - Integer Constant
;        - semicolon
printf   - Keyword
(        - opening brackets
"%d"     - String Constant
,        - comma separator
a        - Identifier
)        - closing brackets
;        - semicolon
a        - Identifier
++       - Operator
;        - semicolon
int      - Keyword
b        - Identifier
=        - Operator
4        - Integer Constant
;        - semicolon
int      - Keyword
c        - Identifier
=        - Operator
3        - Integer Constant
;        - semicolon
/*nested
        comment/*error*/        - multiple line comment
ERROR: at line no. 10
*


Symbol Table

a       Identifier
```

```
b       Identifier
c       Identifier
main    Keyword
int     Keyword
printf  Keyword


Constant Table

"%d"    String Constant
2       Integer Constant
3       Integer Constant
4       Integer 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.

  {alpha}({alpha}|{digit}|{und})*

  Where, alpha [A-Za-z]
  digit [0-9]
  und [_]
  space [ ]

- **Multiline comments should be supported**: This has been supported by checking the occurence of '/*' and '*/' in the code. The statements between them has been excluded. Errors for unmatched and nested comments have also been displayed.
- **Literals:** Different regular expressions have been implemented in the code to support all kinds of literals, i.e integers, floats, strings, etc.

  Float : ({digit}+)\.({digit}+)

- **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.

- **User Defined Functions :** User-defined functions are also supported. Parsing is done for return type, function name, parameters as well as opening and closing braces.

  {alpha}({alpha}|{digit}|{und})*\(({alpha}|{digit}|{und}|{space})*\)

At the end of the token recognition, a list of all the identifiers and constants present in the program is printed by the lexer. The following technique is used to implement this:

● Two structures are maintained: 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 call lookupST( ) or lookupCT( ) and adds it to the corresponding structure.

● In the end, in main( ) function, after yylex returns, we call printST( ) and printCT( ), which in turn prints the list of identifier and constants in a proper format.

Results and Future Work

Result

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

   Token --- Attribute

3. Constant Table:

   Token --- Attribute

Future Work

The following work contains a flex script used to generate a non-exhaustive set of tokens. The flex script takes care of a number of syntaxes followed by the C language, however a good portion is still not supported by the script. The future work will be focused on creating a much more dense and complete set of token table by extending the existing flex script. The final aim is to provide an exhaustive token support of all the syntaxes supported by the C language.

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
2. The Lex & Yacc Page: http://dinosaur.compilertools.net/
3. John R. Levine, Tony Mason, and Doug Brown. 1992. *Lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., USA.
4. Lex & Yacc: https://www.epaperpress.com/lexandyacc/
5. Lex & yacc tutorial - Aquamentus: http://aquamentus.com/tut_lexyacc.html