Lexical Analyzer for the C Language

National Institute of Technology Karnataka

Date: 22nd January 2020

Submitted To: Dr. P. Santhi Thilagam

Group Members: 1. Nihal Haneef 16CO128

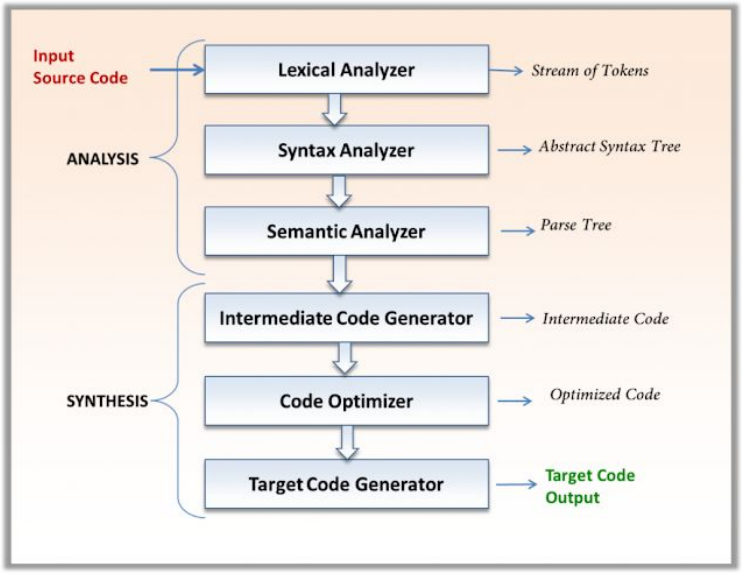
2. Akash Nair 171CO107

3. Meghna Savit 171CO222

ABSTRACT

INTRODUCTION

A compiler is a program that can read a program in one language, the source language - and translate it to an equivalent program in another language, the target language. This project focuses on creating a “Mini C Compiler”, basically a compiler for a subset of the C language. The compiler construction has been divided into four phases called “Phases”.



PHASE I

The first project phase requires us to construct a scanner or in other words a lexical analyzer which is the first part of a compiler. It takes the source program as an input and gives out a *stream of tokens as an output*.

The lexical analyzer maintains a data structure called the symbol table, but for this phase we will just be identifying the token and iThe first project phase requires us to construct a scanner or in other words a lexical analyzer which is the first part of a compiler. It takes the source program as an input and gives out a stream of tokens as an output.

The lexical analyzer maintains a data structure called the symbol table, but for this phase we will just be identifying the token and its type and displaying the necessary information. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme into the symbol table. The scanner will also strip out white spaces and comments from the input and provide necessary error messages as and where it is necessary.ts type and displaying the necessary information. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme into the symbol table.

The scanner will also strip out white spaces and comments from the input and provide necessary error messages as and where it is necessary.

The scanner will take care of the following syntactic structures:

Data Types: int (short and long) and char initialization and declaration.

```
int a = 5, b = 6;
```

Functions: Functions with one parameter of int or char data types. The return type being taken care of will also be int and char.

```
int perimeter (int a) {...}
```

Conditional: If else blocks and nested if else statements.

```
If(a==b) {...} else {...};
```

Array: Single dimensional array initialization.

```
int students [10];
```

Loops: While loop construct will be taken care of.

```
while(a==b) {...}
```

The scanner will analyze the following tokens:

Keywords: The lexer will identify int, long, short, long long, signed, unsigned, for, while,break, continue, if, else, return, for and const.

Identifiers: All identifiers following the C language semantics will be analyzed. The regex for an identifier is `[_a-zA-Z][_a-zA-Z0-9]*`

Strings: The lexer will identify strings in any C program. It can also handle double quotes that are escaped using a \ inside a string.

Constants: The lexer will identify integer, hexadecimal and octal constants.

Operators: The lexer will identify unary, arithmetic, relational, logical, assignment and conditional operators.

Special Symbols: The lexer will identify brackets, braces, parentheses, commas, semicolons, assignment operator and preprocessor directive.

TOOLS & LANGUAGE

1. The language for which the compiler is being made for is a subset of the C language.
2. Lex/Flex will be used.
3. Yacc/Bison will be used.
4. Visual Studio Code will be the primary text editor
5. GitHub will be utilized for collaboration and version control.

CONTENTS

TABLE OF FIGURES

INTRODUCTION

LEXICAL ANALYZER

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

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 our 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 in 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.

SYMBOL TABLE & CONSTANT TABLE

This section contains the definition of the symbol table and the constants table and also defines functions for inserting into the table and displaying its contents within the table.h file, & contains enumerated constants for keywords, operators, special symbols, constants and identifiers in tokens.h.

The symbol table contains the list of identifiers captured by the scanner and the constant table contains all the constants captured by the scanner along with the type of the constant. A simple structure array was used to create a linear table so as to store this information.

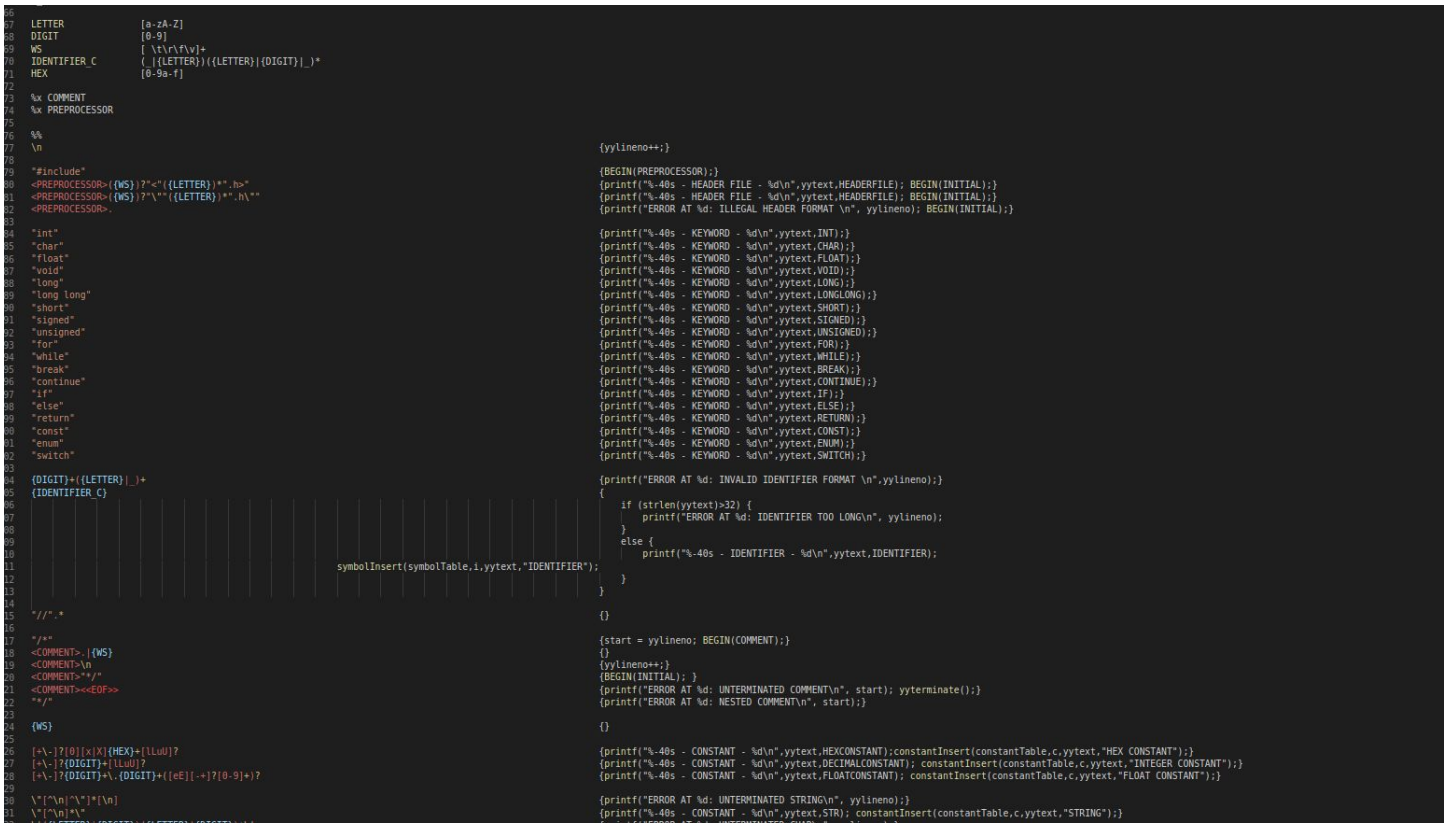
DESIGN OF PROGRAMS

CODE

scanner.l

```
%{

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include "token.h"

int start = 0;
struct symbol
{
    char token[100];
    char type[100];
}symbolTable[100000], constantTable[100000];

int i=0;
int c=0;

int symbolLookup(char* tokenName)
{
    int k=0,flag=0;
    for(k=0;k<=i;k++)
    {
        if(strcmp(symbolTable[k].token,tokenName)==0)
        {
            flag=1;
            break;
        }
    }
    return flag;
}

void symbolInsert(struct symbol table[], int index, char* tokenName, char* tokenType)
{
    int flag;
    flag=symbolLookup(tokenName);
    if(flag==0)
    {
        strcpy(table[index].token, tokenName);
        strcpy(table[index].type, tokenType);
        i++;
    }
}

void constantInsert(struct symbol table[], int index, char* constant, char* constantType)
{
    strcpy(table[index].token, constant);
    strcpy(table[index].type, constantType);
    c++;
}

void display()
{
    int k=0;
    printf("\nSYMBOL TABLE \n\n");
    printf("%-10s \t\t\t %s\n","Symbol","Type");
    for(k=0;k<=i;k++)
        printf("%-10s\t\t\t%-40s\n",symbolTable[k].token,symbolTable[k].type);
    printf("\nCONSTANT TABLE \n\n");
    printf("%-10s \t\t\t %s\n","Constant","Type");
    for(k=0;k<=c;k++)
        printf("%-10s\t\t\t%-40s\n",constantTable[k].token,constantTable[k].type);

}

%}
```

Figure 1: Code for Scanner



```
LETTER              [a-zA-Z]
DIGIT               [0-9]
WS                  [ \t\r\f\v]+
IDENTIFIER_C        (_|{LETTER}|{LETTER}|{DIGIT}|_)*
HEX                 [0-9a-f]

%x COMMENT
%x PREPROCESSOR

%%
\n                                                                  {yylineno++;}

"#include"                                                          {BEGIN(PREPROCESSOR);}
<PREPROCESSOR>{WS}?"<"{LETTER}*".h>"                                {printf("%-40s - HEADER FILE - %d\n",yytext,HEADERFILE); BEGIN(INITIAL);}
<PREPROCESSOR>{WS}?"\""{LETTER}*".h\""                              {printf("%-40s - HEADER FILE - %d\n",yytext,HEADERFILE); BEGIN(INITIAL);}
<PREPROCESSOR>.                                                     {printf("ERROR AT %d: ILLEGAL HEADER FORMAT \n", yylineno); BEGIN(INITIAL);}

"int"                                                               {printf("%-40s - KEYWORD - %d\n",yytext,INT);}
"char"                                                              {printf("%-40s - KEYWORD - %d\n",yytext,CHAR);}
"float"                                                             {printf("%-40s - KEYWORD - %d\n",yytext,FLOAT);}
"void"                                                              {printf("%-40s - KEYWORD - %d\n",yytext,VOID);}
"long"                                                              {printf("%-40s - KEYWORD - %d\n",yytext,LONG);}
"long long"                                                         {printf("%-40s - KEYWORD - %d\n",yytext,LONGLONG);}
"short"                                                             {printf("%-40s - KEYWORD - %d\n",yytext,SHORT);}
"signed"                                                            {printf("%-40s - KEYWORD - %d\n",yytext,SIGNED);}
"unsigned"                                                          {printf("%-40s - KEYWORD - %d\n",yytext,UNSIGNED);}
"for"                                                               {printf("%-40s - KEYWORD - %d\n",yytext,FOR);}
"while"                                                             {printf("%-40s - KEYWORD - %d\n",yytext,WHILE);}
"break"                                                             {printf("%-40s - KEYWORD - %d\n",yytext,BREAK);}
"continue"                                                          {printf("%-40s - KEYWORD - %d\n",yytext,CONTINUE);}
"if"                                                                {printf("%-40s - KEYWORD - %d\n",yytext,IF);}
"else"                                                              {printf("%-40s - KEYWORD - %d\n",yytext,ELSE);}
"return"                                                            {printf("%-40s - KEYWORD - %d\n",yytext,RETURN);}
"const"                                                             {printf("%-40s - KEYWORD - %d\n",yytext,CONST);}
"enum"                                                              {printf("%-40s - KEYWORD - %d\n",yytext,ENUM);}
"switch"                                                            {printf("%-40s - KEYWORD - %d\n",yytext,SWITCH);}

{DIGIT}+({LETTER}|_)+                                               {printf("ERROR AT %d: INVALID IDENTIFIER FORMAT \n",yylineno);}
{IDENTIFIER_C}                                                      {
                                                                        if (strlen(yytext)>32) {
                                                                            printf("ERROR AT %d: IDENTIFIER TOO LONG\n", yylineno);
                                                                        }
                                                                        else {
                                                                            printf("%-40s - IDENTIFIER - %d\n",yytext,IDENTIFIER);
                                                symbolInsert(symbolTable,i,yytext,"IDENTIFIER");
                                                                        }
                                                                    }

"//".*                                                              {}

"/*"                                                                {start = yylineno; BEGIN(COMMENT);}
<COMMENT>.|{WS}                                                     {}
<COMMENT>\n                                                         {yylineno++;}
<COMMENT>"*/"                                                       {BEGIN(INITIAL); }
<COMMENT><<EOF>>                                                    {printf("ERROR AT %d: UNTERMINATED COMMENT\n", start); yyterminate();}
"*/"                                                                {printf("ERROR AT %d: NESTED COMMENT\n", start);}

{WS}                                                                {}

[+\-]?[0][x|X]{HEX}+[lLuU]?                                         {printf("%-40s - CONSTANT - %d\n",yytext,HEXCONSTANT);constantInsert(constantTable,c,yytext,"HEX CONSTANT");}
[+\-]?{DIGIT}+[lLuU]?                                               {printf("%-40s - CONSTANT - %d\n",yytext,DECIMALCONSTANT); constantInsert(constantTable,c,yytext,"INTEGER CONSTANT");}
[+\-]?{DIGIT}+\.{DIGIT}+([eE][-+]?[0-9]+)?                          {printf("%-40s - CONSTANT - %d\n",yytext,FLOATCONSTANT); constantInsert(constantTable,c,yytext,"FLOAT CONSTANT");}

\"[^\n|^\"]*[\n]                                                    {printf("ERROR AT %d: UNTERMINATED STRING\n", yylineno);}
\"[^\n|*\"                                                          {printf("%-40s - CONSTANT - %d\n",yytext,STR); constantInsert(constantTable,c,yytext,"STRING");}
\'{LETTER}|{DIGIT}|{LETTER}|{DIGIT}+\'                              {printf("ERROR AT %d: UNTERMINATED CHAR\n", yylineno);}
\'.\'                                                               {printf("%-40s - CONSTANT - %d\n",yytext,CHARCONSTANT); constantInsert(constantTable,c,yytext,"CHAR");}

"["                                                                 {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,LEFTBRACKET);}
"]"                                                                 {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,RIGHTBRACKET);}
"{"                                                                 {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,LEFTBRACE);}
"}"                                                                 {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,RIGHTBRACE);}
"("                                                                 {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,LEFTPARANTHESES);}
")"                                                                 {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,RIGHTPARANTHESES);}
";"                                                                 {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,SEMICOLON);}
","                                                                 {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,COMMA);}

"++"                                                                {printf("%-40s - OPERATOR - %d\n",yytext,INCREMENT);}
"--"                                                                {printf("%-40s - OPERATOR - %d\n",yytext,DECREMENT);}
"=="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,EQUAL);}
"!="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,NOTEQUAL);}
">="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,GREATEREQUAL);}
"<="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,LESSEREQUAL);}
"&&"                                                                {printf("%-40s - OPERATOR - %d\n",yytext,LOGICALAND);}
"||"                                                                {printf("%-40s - OPERATOR - %d\n",yytext,LOGICALOR);}
"+="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,ADDASSIGN);}
"-="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,SUBTRACTASSIGN);}
"*="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,MULTASSIGN);}
"/="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,DIVIDEASSIGN);}
"%="                                                                {printf("%-40s - OPERATOR - %d\n",yytext,MODASSIGN);}

"-"                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,MINUS);}
"!"                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,NOT);}
"+"                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,PLUS);}
"*"                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,MULT);}
"/"                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,DIV);}
"%"                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,MOD);}
">"                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,GREATER);}
"<"                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,LESSER);}
"="                                                                 {printf("%-40s - OPERATOR - %d\n",yytext,ASSIGNMENT);}

.                                                                   {printf("ERROR AT %d: ILLEGAL CHARACTER\n", yylineno);}

%%



int yywrap(){}
int main(){
    int newtoken;
    yyin = fopen("test6.c","r");
    yylex();
    display();
    return 0;
}
```

Figure 2: Code for Scanner

Figure 3: Code for Scanner

token.h

```
#ifndef TOKEN_H
#define TOKEN_H

enum keywords
{
  INT=100,
  CHAR,
  FLOAT,
  SIGNED,
  UNSIGNED,
  SHORT,
  LONG,
  LONGLONG,
  CONST,
  RETURN,
  IF,
  ELSE,
  FOR,
  WHILE,
  BREAK,
  CONTINUE,
  VOID,
  ENUM,
  SWITCH
};

enum IDENTIFIER
{
  IDENTIFIER=200
};

enum constants
{
  HEXCONSTANT=300,
  FLOATCONSTANT,
  DECIMALCONSTANT,
  CHARCONSTANT,
  STR,
  HEADERFILE,
  MACRO
};

enum operators
{
  MINUS=400,
  INCREMENT,
  DECREMENT,
  NOT,
  PLUS,
  MULT,
  DIV,
  MOD,
  EQUAL,
  NOTEQUAL,
  GREATER,
  LESSER,
  GREATEREQUAL,
  LESSEREQUAL,
  ASSIGNMENT,
  LOGICALAND,
  LOGICALOR,
  ADDASSIGN,
  SUBTRACTASSIGN,
  MULTASSIGN,
  DIVIDEASSIGN,
  MODASSIGN
};

enum special_symbols
{
  DELIMITER=500,
  LEFTBRACKET,
  RIGHTBRACKET,
  LEFTBRACE,
  RIGHTBRACE,
  LEFTPARANTHESES,
  RIGHTPARANTHESES,
  COMMA,
  SEMICOLON
};

#endif
```

Figure 4: Code for token.h

EXPLANATION

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

- Pre-processor instructions
- Single-line comments
- Multi-line comments
- Errors for unmatched comments
- Errors for nested comments
- Parentheses (all types)
- Operators
- Literals (integer, float, string)
- Errors for incomplete strings
- Keywords
- Identifiers

Keywords accounted for: int, char, float, signed, unsigned, short, long, long long, const, break, continue, for, while, if, else, void, enum and return.

TEST CASES

WITHOUT ERRORS

```c
// Normal Program to show case program with no errors
/* This will work */

#include<stdio.h>
#include<string.h>

int main() {
    int a = 5;
    float b = 5.0;
    char aString[10] = "Hello World";
    int a = 0x321;
    char zy = 'a';
    a+=5;
    return 0;
}
```

Figure 5: Code for No Errors

Overview:

- The program is fed with the above C program which is error free.
- It contains both single line and multi-line comments.
- It contains int, float and char initializations.
- It contains the use of operators.
- It contains the use of various constants as well as special symbols.
- The output for the following is shown in the following page.

```
<stdio.h>                       - HEADER FILE - 305
<string.h>                      - HEADER FILE - 305
int                             - KEYWORD - 100
main                            - IDENTIFIER - 200
(                               - SPECIAL SYMBOL - 505
)                               - SPECIAL SYMBOL - 506
{                               - SPECIAL SYMBOL - 503
int                             - KEYWORD - 100
a                               - IDENTIFIER - 200
=                               - OPERATOR - 414
5                               - CONSTANT - 302
;                               - SPECIAL SYMBOL - 508
float                           - KEYWORD - 102
b                               - IDENTIFIER - 200
=                               - OPERATOR - 414
5.0                             - CONSTANT - 301
;                               - SPECIAL SYMBOL - 508
char                            - KEYWORD - 101
aString                         - IDENTIFIER - 200
[                               - SPECIAL SYMBOL - 501
10                              - CONSTANT - 302
]                               - SPECIAL SYMBOL - 502
=                               - OPERATOR - 414
"Hello World"                   - CONSTANT - 304
;                               - SPECIAL SYMBOL - 508
int                             - KEYWORD - 100
a                               - IDENTIFIER - 200
=                               - OPERATOR - 414
0x321                           - CONSTANT - 300
;                               - SPECIAL SYMBOL - 508
char                            - KEYWORD - 101
zy                              - IDENTIFIER - 200
=                               - OPERATOR - 414
'a'                             - CONSTANT - 303
;                               - SPECIAL SYMBOL - 508
a                               - IDENTIFIER - 200
+=                              - OPERATOR - 417
5                               - CONSTANT - 302
;                               - SPECIAL SYMBOL - 508
return                          - KEYWORD - 109
0                               - CONSTANT - 302
;                               - SPECIAL SYMBOL - 508
}                               - SPECIAL SYMBOL - 504

SYMBOL TABLE

Symbol                          Type
main                            IDENTIFIER
a                               IDENTIFIER
b                               IDENTIFIER
aString                         IDENTIFIER
zy                              IDENTIFIER


CONSTANT TABLE

Constant                        Type
5                               INTEGER CONSTANT
5.0                             FLOAT CONSTANT
10                              INTEGER CONSTANT
"Hello World"                   STRING
0x321                           HEX CONSTANT
'a'                             CHAR
5                               INTEGER CONSTANT
0                               INTEGER CONSTANT
```

Figure 6: Output for No Errors

Overview:

- ➢ All tokens are identified.
- ➢ Comments are removed.
- ➢ Whitespaces have been removed.
- ➢ No error messages are shown.

WITH ERRORS

C test1.c

```c
/*
 - An All in one test case with all kinds of errors.
*/

/*This should not*/ work */
#include<stdio
.h>
#include<string.h>

int main() {

    int a = 5;
    char aString[10] = "Hello
        World";
    @$
    char zy = 'ab';
    a += 5;
    return 0;
}
```

Figure 7: Code for Errors

Overview:

- The program is fed with the above C program which is erroneous.
- It contains nested multi line comments.
- It contains invalid header file calls.
- It contains invalid string declarations.
- It contains invalid character declarations.
- It contains illegal characters not in the language.
- The output for the following is shown in the following page.

```
work                                    - IDENTIFIER - 200
ERROR AT 5: NESTED COMMENT
ERROR AT 6: ILLEGAL HEADER FORMAT
stdio                                   - IDENTIFIER - 200
ERROR AT 7: ILLEGAL CHARACTER
h                                       - IDENTIFIER - 200
>                                       - OPERATOR - 410
<string.h>                              - HEADER FILE - 305
int                                     - KEYWORD - 100
main                                    - IDENTIFIER - 200
(                                       - SPECIAL SYMBOL - 505
)                                       - SPECIAL SYMBOL - 506
{                                       - SPECIAL SYMBOL - 503
int                                     - KEYWORD - 100
a                                       - IDENTIFIER - 200
=                                       - OPERATOR - 414
5                                       - CONSTANT - 302
;                                       - SPECIAL SYMBOL - 508
char                                    - KEYWORD - 101
aString                                 - IDENTIFIER - 200
[                                       - SPECIAL SYMBOL - 501
10                                      - CONSTANT - 302
]                                       - SPECIAL SYMBOL - 502
=                                       - OPERATOR - 414
ERROR AT 13: UNTERMINATED STRING
World                                   - IDENTIFIER - 200
ERROR AT 13: UNTERMINATED STRING
ERROR AT 13: ILLEGAL CHARACTER
ERROR AT 13: ILLEGAL CHARACTER
char                                    - KEYWORD - 101
zy                                      - IDENTIFIER - 200
=                                       - OPERATOR - 414
ERROR AT 14: UNTERMINATED CHAR
;                                       - SPECIAL SYMBOL - 508
a                                       - IDENTIFIER - 200
+=                                      - OPERATOR - 417
5                                       - CONSTANT - 302
;                                       - SPECIAL SYMBOL - 508
return                                  - KEYWORD - 109
0                                       - CONSTANT - 302
;                                       - SPECIAL SYMBOL - 508
}                                       - SPECIAL SYMBOL - 504

SYMBOL TABLE

Symbol                          Type
work                            IDENTIFIER
stdio                           IDENTIFIER
h                               IDENTIFIER
main                            IDENTIFIER
a                               IDENTIFIER
aString                         IDENTIFIER
World                           IDENTIFIER
zy                              IDENTIFIER


CONSTANT TABLE

Constant                        Type
5                               INTEGER CONSTANT
10                              INTEGER CONSTANT
5                               INTEGER CONSTANT
0                               INTEGER CONSTANT
```

Figure 8: Output for Errors

Overview:

- ➢ Appropriate error messages have been shown for invalid header files.
- ➢ Appropriate error messages have been shown for invalid nested comments.
- ➢ Appropriate error messages have been shown for invalid character declarations.
- ➢ Appropriate error messages have been shown for invalid string declarations.
- ➢ Appropriate error messages have been shown for invalid tokens.

OTHER CASE - 1

```c
/*
 - Test for single line comments
 - Test for multi-line comments
 - Test for single line nested comments
 - Test for multiline nested comments

The output in lex should remove all the comments including this one
*/

#include<stdio.h>

void main(){s
    // Single line comment

    /* Multi-line comment
       Like this */

    /* here */ int a; /* "int a" should be untouched */

    // This nested comment // This comment should be removed should be removed



    /* To make things /*  nested multi-line comment */ interesting */

    return 0;
}
```

Figure 9: Code for Other Test Case -1

Overview:

- The program is fed with the above C program which is erroneous.
- It contains single and multiline comments.
- The output for the following is shown in the following page.

```
<stdio.h>                                - HEADER FILE - 305
void                                     - KEYWORD - 116
main                                     - IDENTIFIER - 200
(                                        - SPECIAL SYMBOL - 505
)                                        - SPECIAL SYMBOL - 506
{                                        - SPECIAL SYMBOL - 503
s                                        - IDENTIFIER - 200
int                                      - KEYWORD - 100
a                                        - IDENTIFIER - 200
;                                        - SPECIAL SYMBOL - 508
interesting                              - IDENTIFIER - 200
ERROR AT 24: NESTED COMMENT
return                                   - KEYWORD - 109
0                                        - CONSTANT - 302
;                                        - SPECIAL SYMBOL - 508
}                                        - SPECIAL SYMBOL - 504

SYMBOL TABLE

Symbol                          Type
main                           IDENTIFIER
s                              IDENTIFIER
a                              IDENTIFIER
interesting                    IDENTIFIER


CONSTANT TABLE

Constant                        Type
0                              INTEGER CONSTANT
```

Figure 10: Output for Other Test Case -1

Overview:

- ➢ Appropriate error messages have been shown for all cases.
- ➢ Appropriate token messages have shown.

OTHER CASE - 2

```
/*
 - Test for multi-line comment that doesn't end till EOF

The output in lex should print as error message when the comment does not terminate
It should remove the comments that terminate
s*/

#include<stdio.h>

void main(){

    // This is fine
    /* This as well
    like we know */

    /* This is not fine since
    this comment has to end somewhere

    return 0;
}
```

Figure 11: Code for Other Test Case -2

Overview:

- ➢ The program is fed with the above C program which does not end till EOF.
- ➢ The output for the following is shown in the following page.

```
<stdio.h>                                   - HEADER FILE - 305
void                                        - KEYWORD - 116
main                                        - IDENTIFIER - 200
(                                           - SPECIAL SYMBOL - 505
)                                           - SPECIAL SYMBOL - 506
{                                           - SPECIAL SYMBOL - 503
ERROR AT 16: UNTERMINATED COMMENT

SYMBOL TABLE

Symbol                          Type
main                           IDENTIFIER


CONSTANT TABLE

Constant                        Type
```

Figure 12: Output for Other Test Case -2

Overview:

- ➢ Appropriate error messages have been shown for all cases.
- ➢ Appropriate token messages have shown.

OTHER CASE - 3

```c
/*
Identifying tokens and displaying symbol and constants table

Following tokens must be detected
 - Keywords   (int, long int, long long int, main include)
 - Identifiers (main,total,x,y,printf),
 - Constants  (-10, 20, 0x0f, 123456l)
 - Strings ("Total = %d \n")
 - Special symbols and Brackets ( (), {}, ;, ,)
 - Operators (+,-,=,*,/,%,--,++)

The output should display appropriate tokens with their type and also the symbol and constants table
*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int x, y;
  long long int total, diff;
  int *ptr;
  float lk = 0.123456;
  unsigned int a = 0x0f;
  long int mylong = 123456;
  long int i, j;
  for(i=0; i < 10; i++){
    for(j=10; j > 0; j--){
    printf("%d",i);
    }
  }
  x = -10, y = 20;
  x=x*3/2;
  total = x + y;
  diff = x - y;
  int rem = x % y;
  printf ("Total=%d \n", total);
}
```

Figure 13: Code for Other Test Case -3

Overview:

- ➢ The program is fed with the above C program which does not end till EOF.
- ➢ The output for the following is shown in the following page.

```
 <stdio.h>                              - HEADER FILE - 305
 <stdlib.h>                             - HEADER FILE - 305
int                                     - KEYWORD - 100
main                                    - IDENTIFIER - 200
(                                       - SPECIAL SYMBOL - 505
)                                       - SPECIAL SYMBOL - 506
{                                       - SPECIAL SYMBOL - 503
int                                     - KEYWORD - 100
x                                       - IDENTIFIER - 200
,                                       - SPECIAL SYMBOL - 507
y                                       - IDENTIFIER - 200
;                                       - SPECIAL SYMBOL - 508
long long                               - KEYWORD - 107
int                                     - KEYWORD - 100
total                                   - IDENTIFIER - 200
,                                       - SPECIAL SYMBOL - 507
diff                                    - IDENTIFIER - 200
;                                       - SPECIAL SYMBOL - 508
int                                     - KEYWORD - 100
*                                       - OPERATOR - 405
ptr                                     - IDENTIFIER - 200
;                                       - SPECIAL SYMBOL - 508
float                                   - KEYWORD - 102
lk                                      - IDENTIFIER - 200
=                                       - OPERATOR - 414
0.123456                                - CONSTANT - 301
;                                       - SPECIAL SYMBOL - 508
unsigned                                - KEYWORD - 104
int                                     - KEYWORD - 100
a                                       - IDENTIFIER - 200
=                                       - OPERATOR - 414
0x0f                                    - CONSTANT - 300
;                                       - SPECIAL SYMBOL - 508
long                                    - KEYWORD - 106
int                                     - KEYWORD - 100
mylong                                  - IDENTIFIER - 200
=                                       - OPERATOR - 414
123456                                  - CONSTANT - 302
;                                       - SPECIAL SYMBOL - 508
long                                    - KEYWORD - 106
int                                     - KEYWORD - 100
i                                       - IDENTIFIER - 200
,                                       - SPECIAL SYMBOL - 507
j                                       - IDENTIFIER - 200
;                                       - SPECIAL SYMBOL - 508
for                                     - KEYWORD - 112
(                                       - SPECIAL SYMBOL - 505
i                                       - IDENTIFIER - 200
=                                       - OPERATOR - 414
0                                       - CONSTANT - 302
;                                       - SPECIAL SYMBOL - 508
i                                       - IDENTIFIER - 200
<                                       - OPERATOR - 411
10                                      - CONSTANT - 302
;                                       - SPECIAL SYMBOL - 508
```

Figure 14: Output for Other Test Case –3.1

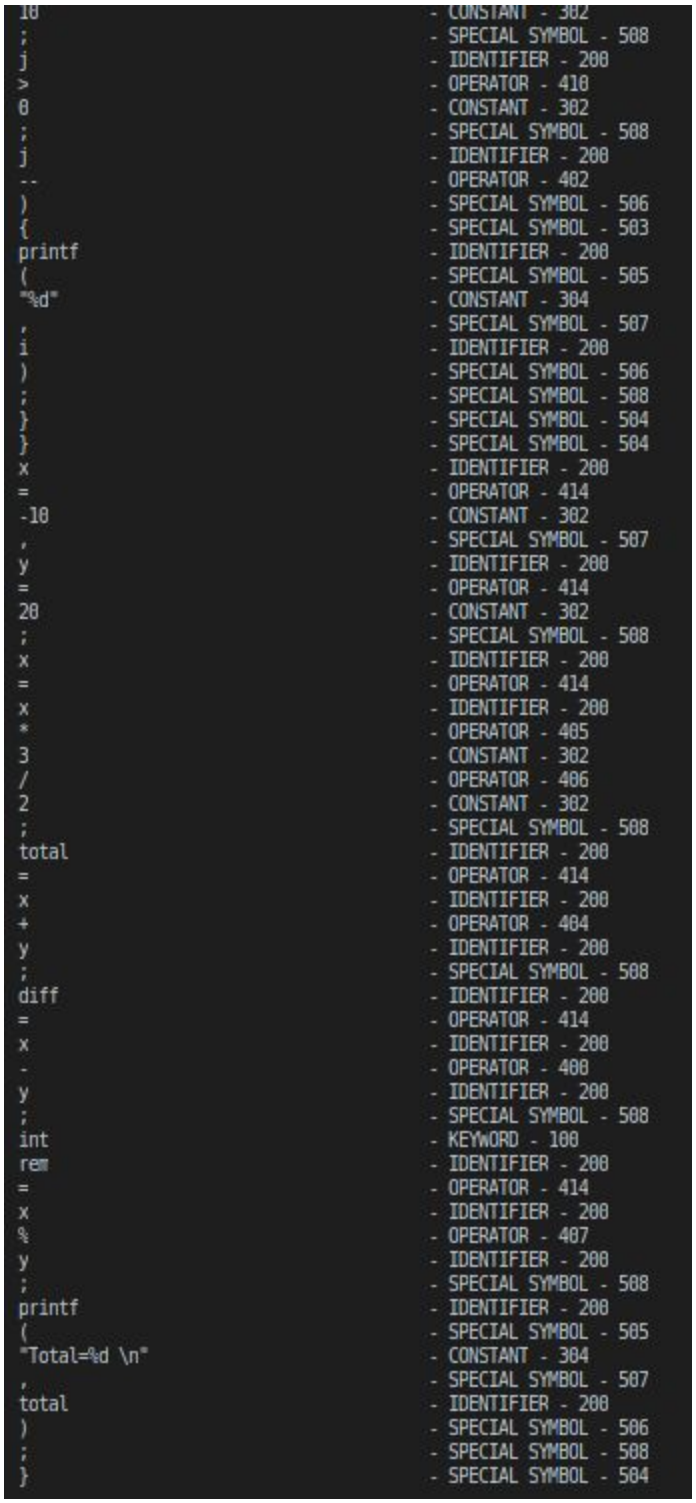
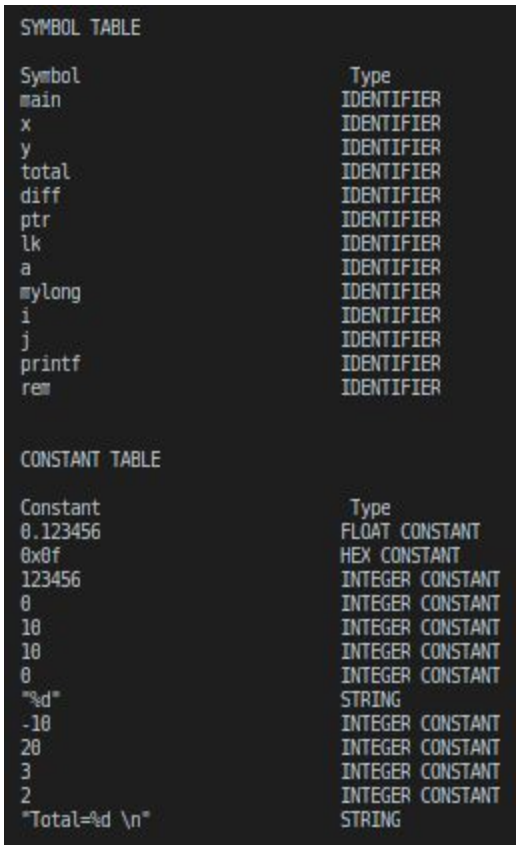


Figure 15: Output for Other Test Case -3.2



SYMBOL TABLE

Symbol	Type
main	IDENTIFIER
x	IDENTIFIER
y	IDENTIFIER
total	IDENTIFIER
diff	IDENTIFIER
ptr	IDENTIFIER
lk	IDENTIFIER
a	IDENTIFIER
mylong	IDENTIFIER
i	IDENTIFIER
j	IDENTIFIER
printf	IDENTIFIER
rem	IDENTIFIER

CONSTANT TABLE

Constant	Type
0.123456	FLOAT CONSTANT
0x0f	HEX CONSTANT
123456	INTEGER CONSTANT
0	INTEGER CONSTANT
10	INTEGER CONSTANT
10	INTEGER CONSTANT
0	INTEGER CONSTANT
"%d"	STRING
-10	INTEGER CONSTANT
20	INTEGER CONSTANT
3	INTEGER CONSTANT
2	INTEGER CONSTANT
"Total=%d \n"	STRING

Figure 16: Output for Other Test Case -3.3

Overview:

- ➢ Appropriate error messages have been shown for all cases.
- ➢ Appropriate token messages have shown.
- ➢ Symbols and Constants are shown respectively.

IMPLEMENTATION

OVERVIEW

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:*** This has been supported by using custom regular algorithm especially robust in cases where tricky characters like * or / are used within the comments.
- ***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 kinds of errors have been handled in the rules written in the script.
- ***Error Handling for Nested Comments:*** This use-case has been handled by the custom defined regular expressions which help throw errors when comment opening or closing is missing.

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

- A token.h header file was created which contains enumerated objects for each token.
- Once a token is matched with a particular regex, that is, it gets identified, then the token along with its unique ID obtained from the header file is displayed in the terminal.
- Error messages are displayed with reason along with the line number.

Results Future Work & References

Results

A scanner has been created which is capable of taking a program written in C language as an input and give out a stream of tokens present in the program as the output. It removes whitespaces, comments and shows necessary error messages.

Future Work

Furthermore, the flex script presented in this report takes care of most of the rules of the C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of the C language and making it more efficient.

References

1. Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986
2. http://www.csd.uwo.ca/~moreno/CS447/Lectures/Lexical.html/node11.html