COMPILER DESIGN PROJECT REPORT - 1

SCANNER FOR C

PROGRAMMING LANGUAGE

Submitted By :

Arvind Ramachandran - 15CO111

Aswanth P P - 15CO112

DATE : 20/01/2018

INTRODUCTION

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. We basically have two phases of compilers, namely **Analysis phase** and **Synthesis phase.**

Analysis phase creates an intermediate representation from the given source code. It is also termed as front end of the compiler. Analysis phase consists of :

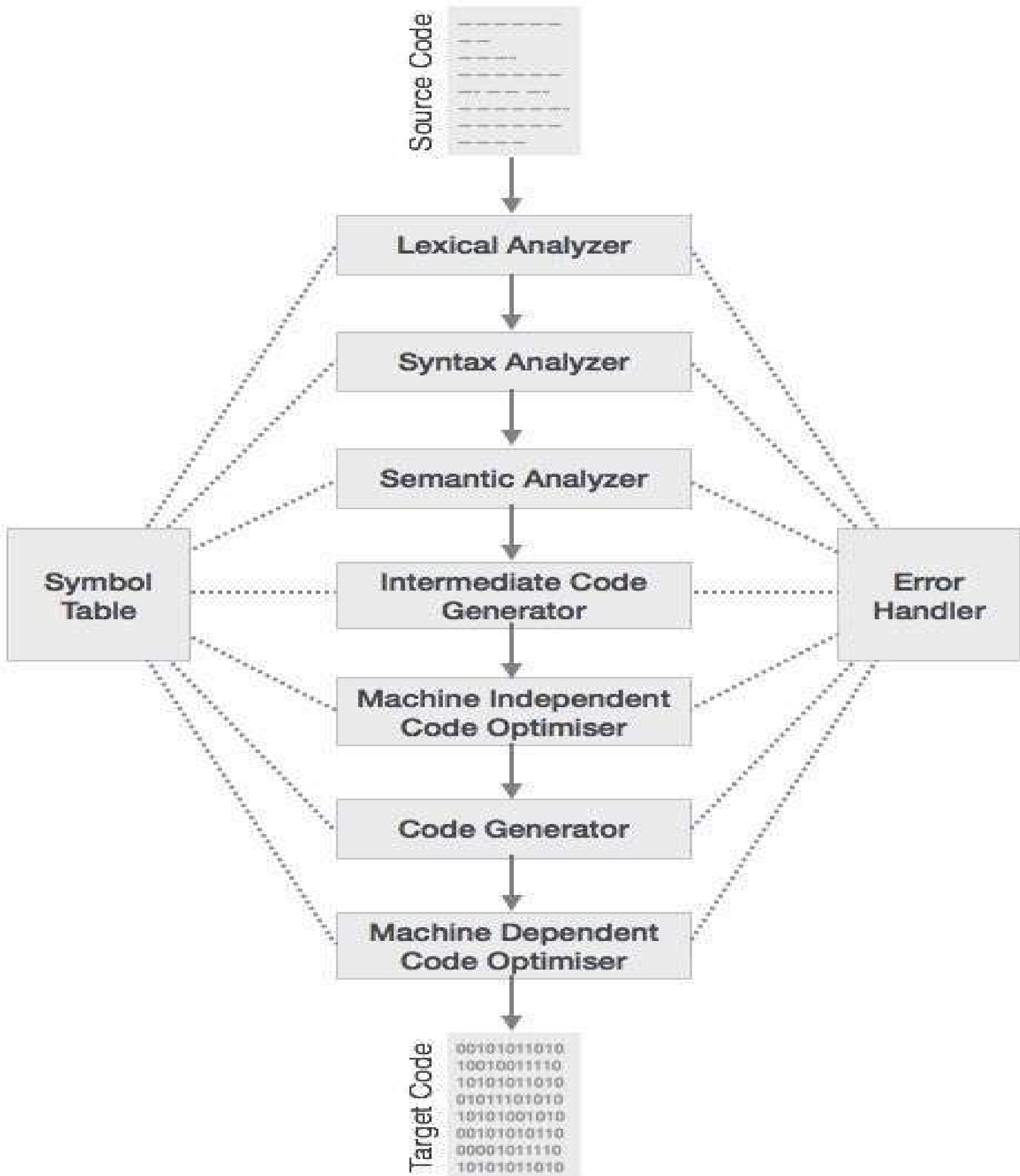
1. **Lexical Analysis** :
    a. The lexical phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as, an identifier, a keyword, a punctuation character.
    b. The character sequence forming a token is called the lexeme for the token.
2. **Syntax Analysis** :
    a. Syntax analysis imposes a hierarchical structure on the token stream. This hierarchical structure is called syntax tree.
    b. A syntax tree has an interior node is a record with a field for the operator and two fields containing pointers to the records for the left and right children.
    c. A leaf is a record with two or more fields, one to identify the token at the leaf, and the other to record information about the token.
3. **Semantic Analysis** :
    a. This phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
    b. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
    c. An important component of semantic analysis is type checking.

4. **Intermediate Code Generation** :
    a. The syntax and semantic analysis generate a explicit intermediate representation of the source program.
    b. The intermediate representation should have two important properties:
        i. It should be easy to produce,
        ii. And easy to translate into target program.
    c. Intermediate representation can have a variety of forms. One of the forms is: three address code; which is like the assembly language for a machine in which every location can act like a register.
    d. Three address code consists of a sequence of instructions, each of which has at most three operands.

Synthesis phase creates an equivalent target program from intermediate representation. It is the back end of the compiler. Synthesis phase consists of :

1. **Code Optimization** :
    a. Code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.
2. **Code Generation** :
    a. The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code.
    b. Memory locations are selected for each of the variables used by the program.
    c. Then, the each intermediate instruction is translated into a sequence of machine instructions that perform the same task.

The following diagram shows the various phases of a compiler.


Source Code
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Machine Independent Code Optimiser
Code Generator
Machine Dependent Code Optimiser
Target Code
Symbol Table
Error Handler

LEXICAL ANALYSIS

The word “lexical” in the traditional sense means “pertaining to words”. In terms of programming languages, words are objects like variable names, numbers, keywords etc. Such words are traditionally called tokens.

Lexical analysis is the first phase of compiler which is also termed as scanning. A token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc. Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

A lexical analyser , or lexer for short, will as its input take a string of individual letters and divide this string into tokens. Additionally, it will filter out whatever separates the tokens (the so-called white-space) , i.e., lay-out characters (spaces,newlines,etc.) and comments.

Tokens, Patterns and Lexemes

Token : It is a valid sequence of characters which are given by lexeme. In a programming language, keywords, constant, identifiers, numbers, operators and punctuations symbols are possible tokens to be identified. Example of tokens:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Pattern : A pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules.

For example - [A-Za-z][A-Za-z_0-9]*

Lexeme : A lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a token. Eg: c=a+b*5;

The sequence of tokens produced by lexical analyzer helps the parser in analyzing the syntax of programming languages.

Role of a Lexical Analyzer

A lexical analyzer performs the following tasks :

- Reads the source program, scans the input characters, group them into lexemes and produce the token as output.

- Enters the identified token into the symbol table.
- Strips out white spaces and comments from source program.
- Correlates error messages with the source program i.e., displays error message with its occurrence by specifying the line number.
- Expands the macros if it is found in the source program.

Need of Lexical Analyzer

- **Simplicity of design of compiler -** The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.
- **Compiler efficiency is improved -** Specialized buffering techniques for reading characters speed up the compiler process.
- **Compiler portability is enhanced**

Lexical Errors

A character sequence that cannot be scanned into any valid token is a lexical error. Lexical errors are uncommon, but they still must be handled by a scanner. Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

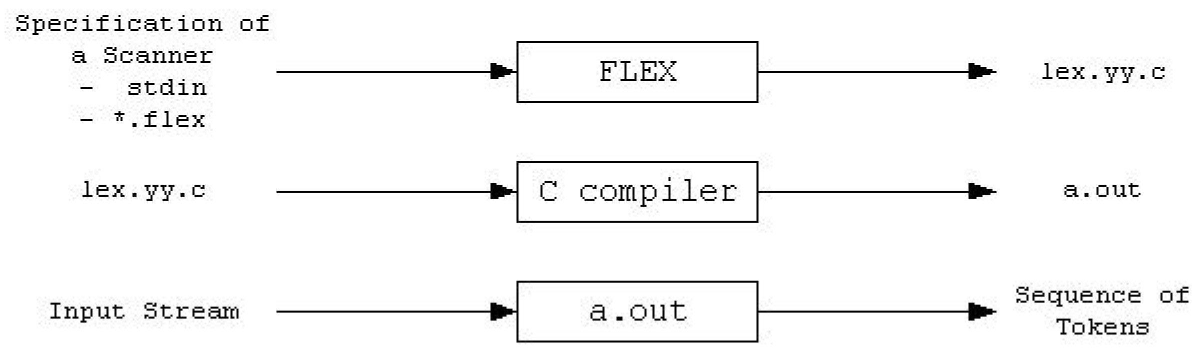
Lexical Analyzer also generates errors in the following cases:

1) **Unterminated String** : When the right number of inverted commas are not provided.
2) **Nested Comments** : Nested comments are not supported.
3) **Unmatched Parenthesis** : If there are missing parenthesis, an error message is generated.
4) **Invalid Identifier** : If the entered identifier does not match the identifier forming rules, an error message is displayed.

FLEX

The scanner performs lexical analysis of a certain program. It reads the source program as a sequence of characters and recognizes "larger" textual units called tokens.

FLEX stands for Fast Lexical Analyzer Generator. It is a tool for generating scanners. Instead of writing a scanner from scratch, you only need to identify the vocabulary of a certain language, write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a scanner for you. FLEX is generally used in the manner depicted here:



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.

- ***.lex** is in the form of pairs of regular expressions and C code.
- **lex.yy.c** defines a routine yylex() that uses the specification to recognize tokens.
- **a.out** is actually the scanner.

These programs perform character parsing and tokenizing via the use of a deterministic finite automaton (DFA). A DFA is a theoretical machine accepting regular languages. These machines are a subset of the collection of Turing machines. DFAs are equivalent to read-only right moving Turing machines. The syntax is based on the use of regular expressions.

FORMAT OF THE FLEX FILE

The flex input file consists of three sections, separated by a line with just `%%' in it:

definitions
%%
rules
%%
user code

The **definitions** section contains declarations of simple **name** definitions to simplify the scanner specification, and declarations of **start conditions**, which are explained in a later section. Name definitions have the form:

name definition

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-whitespace character following the name and continuing to the end of the line. The definition can subsequently be referred to using "{name}", which will expand to "(definition)". For example,

DIGIT   [0-9]
*ID   [a-z][a-z0-9]**

defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

*{DIGIT}+"."{DIGIT}**

is identical to

*([0-9])+"."([0-9])**

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

The rules section of the flex input contains a series of rules of the form:

pattern   action

where the pattern must be unindented and the action must begin on the same line.

Finally, the user code section is simply copied to `lex.yy.c' verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too.

In the definitions and rules sections, any *indented* text or text enclosed in `%{' and `%}' is copied verbatim to the output (with the `%{}"s removed). The `%{}"s must appear unindented on lines by themselves. In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors.

In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/".

IMPLEMENTATION

scanner.l

This is the lex program that contains various regular expressions for all the specific actions that are to be carried out by a lexical analyzer. This file is converted to **lex.yy.c** which is compiled to get the executable **a.out**.

CODE :

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
int var=0,i,nc=0,cLine=0,flag=0;
int lineNumber=1;
int cBrac=0;

FILE *symbol,*constants;
char *comment,*inputFile, s_comment[1000];

void insertToTable(char *yytext,char type);
void displayComment(char *yytext);
void storeSingleLineComment(char *yytext);

struct Node {
    char *tname;
    int av;
    struct Node *next;
}*head=NULL;

%}

digit  [0-9]
letter  [a-zA-Z]
keyword "auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"return"|
datatype "int"|"char"|"void"
sign "signed"|"unsigned"
modifiers "long"|"short"
relational >|<|<=|>=|!=|==
logical \&|\^|\~
arithmatic \+|\-|\*|\/|\%
puncuator \(|\)|\[|\]|\;|\,|\:|\.
assignment =
quote \'|\"|\\
whitespace [ \t]+
newline "\n"
singlelinecomment (\/\/.*)
multilinecommentstart (\/\*)
multilinecommentend (\*\/)
```

Line 1, Column 1 Tab Size: 4 Lex/Flex

FIG. 1

scanner.l

```
identifier ({letter}({letter}|{digit})*)|"_"({letter}|{digit})+

%x DETECT_COMMENT

%%

^#([-a-zA-Z0-9.]|{relational}|{whitespace})* insertToTable(yytext,'d'); //preprocessor directive rule

{keyword} insertToTable(yytext,'k');
{sign}?{whitespace}{modifiers}?{whitespace}{datatype} insertToTable(yytext,'k'); //keyword rule

^{sign}?{whitespace}*{modifiers}?{whitespace}*{datatype}{whitespace}*{identifier}\(.*\)  insertToTable(yytext,'j');  //procedure rule
{identifier}\[{digit}*\] insertToTable(yytext,'a'); // array rule
\*{identifier} insertToTable(yytext,'q'); // pointer rule
{identifier} insertToTable(yytext,'i'); // variable rule
{digit}+({letter}|{digit})+|"_" { printf("%s : %d : Invalid Identifier\n",inputFile,lineNumber); } // invalid identifier


{relational} insertToTable(yytext,'r'); //operator rules
{logical} insertToTable(yytext,'l');
{arithmatic} insertToTable(yytext,'o');
{assignment} insertToTable(yytext,'e');
{puncuator} insertToTable(yytext,'p');

{digit}+ insertToTable(yytext,'c'); //integer constants rule
\"(.)*\" insertToTable(yytext,'s'); //string constants rule
L?\"(\\.|[^\\"])*    {
                if(nc<=0) //invalid String
                printf("%s : %d : String does not End\n",inputFile,lineNumber);
            }
[-+]?{digit}*\.?{digit}+([eE][-+]?{digit}+)? insertToTable(yytext,'f');  // float constant rule
\'{letter}\' insertToTable(yytext,'z');  // character constant rule

{quote} ;
{whitespace} ;
{newline} lineNumber++;

"{"  {  cBrac++;
    insertToTable(yytext,'p');
     }
_
```

Line 82, Column 1 Tab Size: 4 Lex/Flex

FIG. 2

scanner.l

```
     }

"}"  {  cBrac--;
    insertToTable(yytext,'p');
     }

{singlelinecomment} {storeSingleLineComment(yytext);}

{multilinecommentstart}  {
                BEGIN(DETECT_COMMENT);
                nc++;
                cLine++;
                displayComment("\n\t");
            }

<DETECT_COMMENT>{multilinecommentstart} {
                            nc++;
                            if(nc>1)
                            {
                                printf("%s : %d : Nested Comment\n",inputFile,lineNumber);
                                flag = 1;
                            }
                        }

<DETECT_COMMENT>{multilinecommentend} {
                            if(nc>0)
                                nc--;
                            else
                                printf("%s : %d : */ found before /*\n",inputFile,lineNumber);

                            if(nc==0)
                                BEGIN(INITIAL);
                    }

<DETECT_COMMENT>\n    {
                 cLine++;
                 lineNumber++;
                 displayComment("\n");
                }

<DETECT_COMMENT>. {displayComment(yytext);}
```

Line 81, Column 7 Tab Size: 4 Lex/Flex

FIG. 3

scanner.l

```
<DETECT_COMMENT>. {displayComment(yytext);}


%%

int main(int argc,char **argv)
{
    comment = (char*)malloc(100*sizeof(char));
    yyin=fopen(argv[1],"r");
    inputFile=argv[1];

    symbol=fopen("symbolTable.txt","w"); //File to write symbol table
    fprintf(symbol,"\n Symbol Table:\n \t\tLexeme\t\t\t\tType\t\t\tAttribute Value\t\t\tLine Number\n");

    constants=fopen("constantTable.txt","w"); // File to write constant table
    fprintf(constants,"\n Constants Table:\n \t\tLexeme\t\t\tType\t\t\tAttribute Value\t\t\tLine Number\n");

    yyout=fopen("parsedTable.txt","w"); // File to write all token in source program
    fprintf(yyout,"\n Table:\n \t\tLexeme\t\t\tToken\t\t\t\tAttribute Value\t\t\tLine Number\n");

    yylex();

        if(nc!=0)
            printf("%s : %d : Comment Does Not End\n",inputFile,lineNumber);

        if(cBrac!=0)
            printf("%s : %d : Unbalanced Parenthesis\n",inputFile,lineNumber);

        fprintf(yyout,"\n");
        if(flag==1)
        {
            cLine = 0;
            fprintf(yyout,"\n\nComment (%d lines):\n",cLine);
            printf("%s : %d : Nested Comment\n",inputFile,lineNumber);
        }
        else
        {
            int i;
            fprintf(yyout,"\n\nMultiLineComment (%d lines):",cLine);
            fputs(comment,yyout);
```

Line 81, Column 7 Tab Size: 4 Lex/Flex

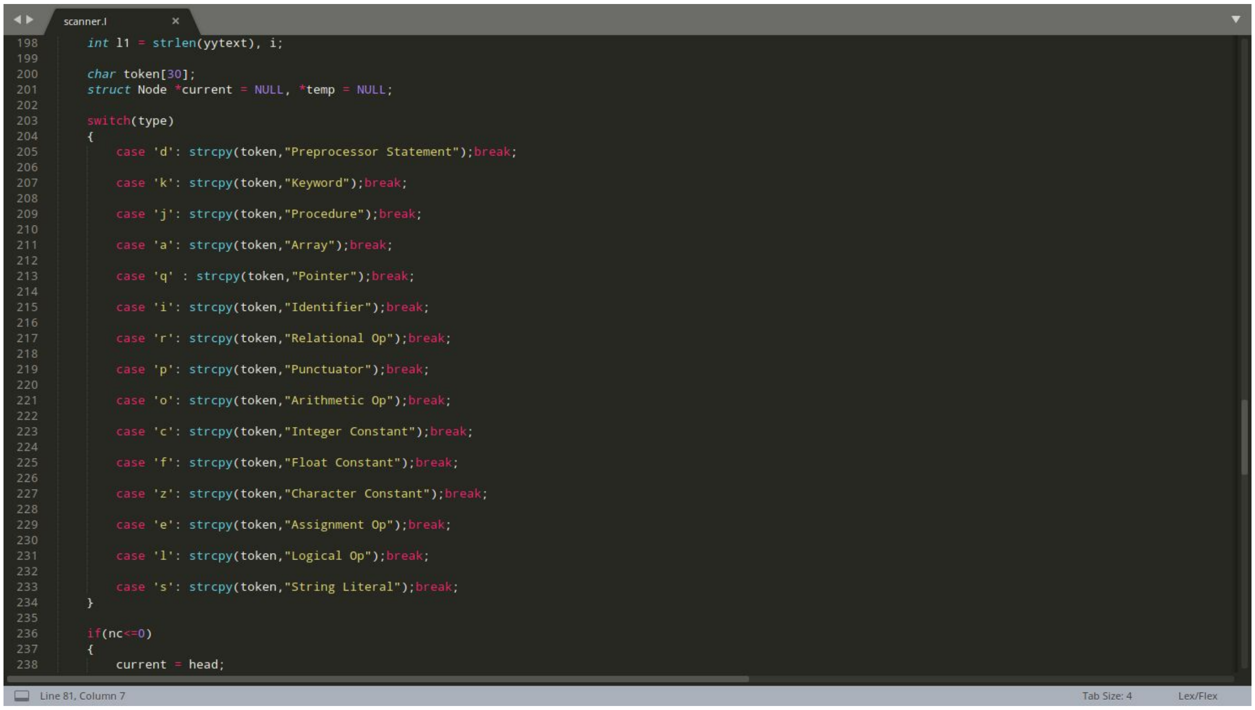
FIG. 4

scanner.l

```
            fprintf(yyout,"\n\nMultiLineComment (%d lines):",cLine);
            fputs(comment,yyout);
            fprintf(yyout,"\n\nSingleLineComment :\n");
            fputs(s_comment,yyout);
        }

    fclose(yyout);
    fclose(symbol);
    fclose(constants);
}

void storeSingleLineComment(char *yytext)
{
    int len = strlen(yytext);
    int i, j=0;
    char *temp;
    temp = (char*)malloc((len+1)*sizeof(char));
    for(i=2;yytext[i]!='\0';i++)
    {
        temp[j++] = yytext[i];
    }
    strcat(temp,"\n");
    strcat(s_comment,temp);
}
void displayComment(char *yytext)
{
    int l1, l2;
    char *temp;

    l1 = strlen(comment);
    l2 = strlen(yytext);
    temp = (char*)malloc((l1+1)*sizeof(char));
    strcpy(temp,comment);
    comment = (char*)malloc((l1+l2+1)*sizeof(char));
    strcat(temp,yytext);
    strcpy(comment,temp);
}
void insertToTable(char *yytext,char type)
{
    int l1 = strlen(yytext), i;

```

Line 81, Column 7 Tab Size: 4 Lex/Flex

FIG. 5


scanner.l
int l1 = strlen(yytext), i;

char token[30];
struct Node *current = NULL, *temp = NULL;

switch(type)
{
case 'd': strcpy(token,"Preprocessor Statement");break;

case 'k': strcpy(token,"Keyword");break;

case 'j': strcpy(token,"Procedure");break;

case 'a': strcpy(token,"Array");break;

case 'q' : strcpy(token,"Pointer");break;

case 'i': strcpy(token,"Identifier");break;

case 'r': strcpy(token,"Relational Op");break;

case 'p': strcpy(token,"Punctuator");break;

case 'o': strcpy(token,"Arithmetic Op");break;

case 'c': strcpy(token,"Integer Constant");break;

case 'f': strcpy(token,"Float Constant");break;

case 'z': strcpy(token,"Character Constant");break;

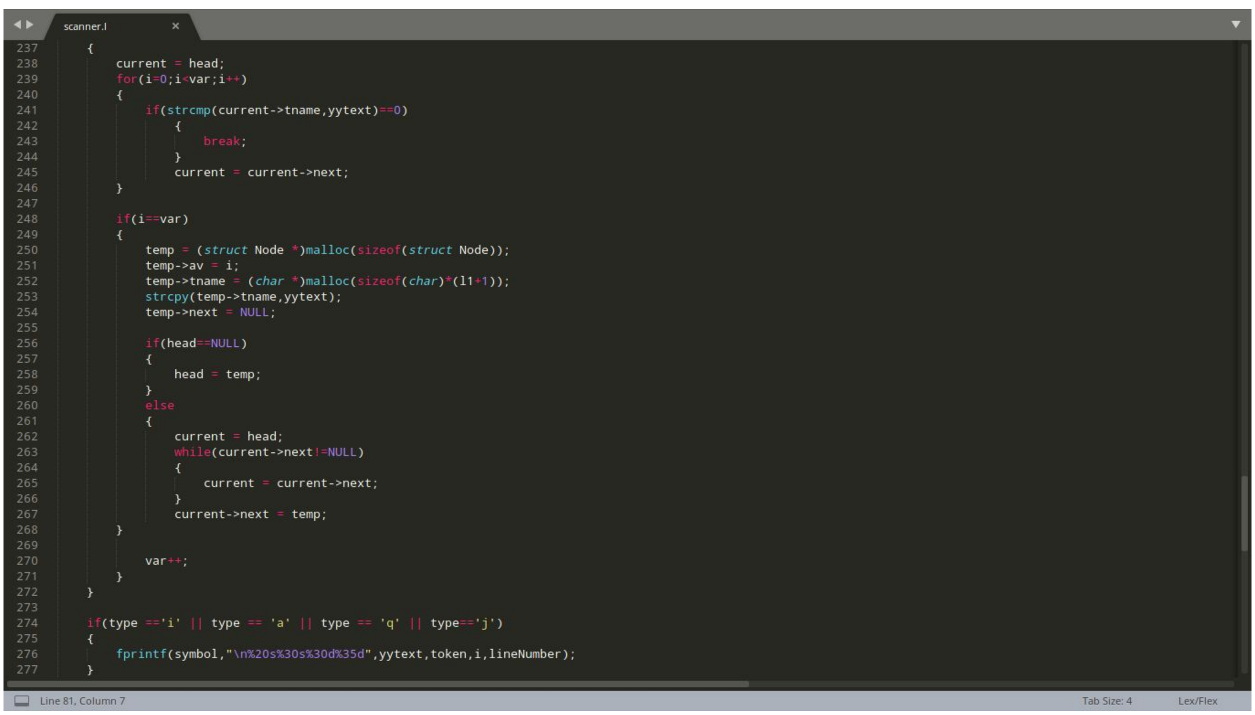
case 'e': strcpy(token,"Assignment Op");break;

case 'l': strcpy(token,"Logical Op");break;

case 's': strcpy(token,"String Literal");break;
}

if(nc<=0)
{
current = head;
Line 81, Column 7
Tab Size: 4
Lex/Flex

FIG. 6


scanner.l
{
current = head;
for(i=0;i<var;i++)
{
if(strcmp(current->tname,yytext)==0)
{
break;
}
current = current->next;
}

if(i==var)
{
temp = (struct Node *)malloc(sizeof(struct Node));
temp->av = i;
temp->tname = (char *)malloc(sizeof(char)*(l1+1));
strcpy(temp->tname,yytext);
temp->next = NULL;

if(head==NULL)
{
head = temp;
}
else
{
current = head;
while(current->next!=NULL)
{
current = current->next;
}
current->next = temp;
}

var++;
}
}

if(type =='i' || type == 'a' || type == 'q' || type=='j')
{
fprintf(symbol,"\n%20s%30s%30d%35d",yytext,token,i,lineNumber);
}
Line 81, Column 7
Tab Size: 4
Lex/Flex

FIG. 7

scanner.l

```
                current->next = temp;
            }

            var++;
        }
    }

    if(type =='i' || type == 'a' || type == 'q' || type=='j')
    {
        fprintf(symbol,"\n%20s%30s%30d%35d",yytext,token,i,lineNumber);
    }
    switch(type)
    {
        case 'c' : fprintf(constants,"\n%20s%20s%30d%35d",yytext,"int",i,lineNumber);
                   break;

        case 'f' : fprintf(constants,"\n%20s%20s%30d%35d",yytext,"float",i,lineNumber);
                   break;

        case 'z' : fprintf(constants,"\n%20s%20s%30d%35d",yytext,"char",i,lineNumber);
                   break;
    }

    fprintf(yyout,"\n%20s%30s%30d%35d",yytext,token,i,lineNumber);
}

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

Line 296, Column 2   Tab Size: 4   Lex/Flex

FIG. 8

Input.c

This is the C source program for which the lexical analysis is done. Based on this, we generate the Symbol and Constants Table.

CODE :

```
#include <stdio.h>
int main()
{
    /* My name is Arvind.
    This is
    a sample
    multi-line
    comment*/
    struct node{
    int a;
    char name;
    };

    float fl = 5.01;
    char letter = 'a';
    int _a=0,b=0,c=5;
    unsigned int arr[50];
    char *ptr;
    scanf("%d %d",&a,&b);
    int sum=0;
    sum=a+b;
    printf("\n Sum : %d \n",sum);

    //Enter Name
    char *name = "Arvind";

    //End the main function
    return 1;
}
void abc()
{
    printf("\nTest function ");
}
```

EXECUTION OF CODE :

The lex code can be executed by the following commands :

- lex <filename1>
- cc lex.yy.c
- ./a.out <filename2>

Where <filename1> is the lex program (here **scanner.l**)

<filename2> is the C source program for which Lexical Analysis is done. (here **input.c**)

OUTPUT :

symbolTable.txt :

Symbol Table:

Lexeme	Type	Attribute Value	Line Number
int main()	Procedure	1	2
node	Identifier	4	9
a	Identifier	6	10
name	Identifier	9	11
fl	Identifier	12	14
letter	Identifier	15	15
_a	Identifier	17	16
b	Identifier	20	16
c	Identifier	21	16
arr[50]	Array	24	17
*ptr	Pointer	25	18
scanf	Identifier	26	19
a	Identifier	6	19
b	Identifier	20	19
sum	Identifier	31	20
sum	Identifier	31	21
a	Identifier	6	21
b	Identifier	20	21
printf	Identifier	33	22
sum	Identifier	31	22
*name	Pointer	35	25
void abc()	Procedure	39	30
printf	Identifier	33	32

constantTable.txt

```
Constants Table:
        Lexeme          Type            Attribute Value         Line Number

            5.01            float               14                      14
             'a'             char               16                      15
               0              int               18                      16
               0              int               18                      16
               5              int               22                      16
               0              int               18                      20
               1              int               38                      28
```

parsedTable.txt

```
Table:
   Lexeme      Token       Attribute Value     Line Number

 #include <stdio.h>       Preprocessor Statement          0          1
         int main()                    Procedure          1          2
                  {                   Punctuator          2          3
             struct                      Keyword          3          9
               node                   Identifier          4          9
                  {                   Punctuator          2          9
                int                      Keyword          5         10
                  a                   Identifier          6         10
                  ;                   Punctuator          7         10
               char                      Keyword          8         11
               name                   Identifier          9         11
                  ;                   Punctuator          7         11
                  }                   Punctuator         10         12
                  ;                   Punctuator          7         12
              float                      Keyword         11         14
                 fl                   Identifier         12         14
                  =                Assignment Op         13         14
               5.01               Float Constant         14         14
                  ;                   Punctuator          7         14
               char                      Keyword          8         15
             letter                   Identifier         15         15
                  =                Assignment Op         13         15
                'a'           Character Constant         16         15
                  ;                   Punctuator          7         15
                int                      Keyword          5         16
                 _a                   Identifier         17         16
                  =                Assignment Op         13         16
                  0             Integer Constant         18         16
                  ,                   Punctuator         19         16
                  b                   Identifier         20         16
                  =                Assignment Op         13         16
                  0             Integer Constant         18         16
                  ,                   Punctuator         19         16
                  c                   Identifier         21         16
                  =                Assignment Op         13         16
                  5             Integer Constant         22         16
```

FIG. 1

,	Punctuator	19	16
c	Identifier	21	16
=	Assignment Op	13	16
5	Integer Constant	22	16
;	Punctuator	7	16
unsigned	Keyword	23	17
int	Keyword	5	17
arr[50]	Array	24	17
;	Punctuator	7	17
char	Keyword	8	18
*ptr	Pointer	25	18
;	Punctuator	7	18
scanf	Identifier	26	19
(	Punctuator	27	19
"%d %d"	String Literal	28	19
,	Punctuator	19	19
&	Logical Op	29	19
a	Identifier	6	19
,	Punctuator	19	19
&	Logical Op	29	19
b	Identifier	20	19
)	Punctuator	30	19
;	Punctuator	7	19
int	Keyword	5	20
sum	Identifier	31	20
=	Assignment Op	13	20
0	Integer Constant	18	20
;	Punctuator	7	20
sum	Identifier	31	21
=	Assignment Op	13	21
a	Identifier	6	21
+	Arithmetic Op	32	21
b	Identifier	20	21
;	Punctuator	7	21
printf	Identifier	33	22
(	Punctuator	27	22
"\n Sum : %d \n"	String Literal	34	22
,	Punctuator	19	22
sum	Identifier	31	22
)	Punctuator	30	22

Line 1, Column 1 Spaces: 2 Plain Text

FIG. 2

;	Punctuator	7	21
printf	Identifier	33	22
(	Punctuator	27	22
"\n Sum : %d \n"	String Literal	34	22
,	Punctuator	19	22
sum	Identifier	31	22
)	Punctuator	30	22
;	Punctuator	7	22
char	Keyword	8	25
*name	Pointer	35	25
=	Assignment Op	13	25
"Arvind"	String Literal	36	25
;	Punctuator	7	25
return	Keyword	37	28
1	Integer Constant	38	28
;	Punctuator	7	28
}	Punctuator	10	29
void abc()	Procedure	39	30
{	Punctuator	2	31
printf	Identifier	33	32
(	Punctuator	27	32
"\nTest function "	String Literal	40	32
)	Punctuator	30	32
;	Punctuator	7	32
}	Punctuator	10	33

```
MultiLineComment (5 lines):
   My name is Arvind.
  This is
  a sample
  multi-line
  comment

SingleLineComment :
Enter Name
End the main function
```

FIG. 3

TEST CASES

Test Case Filename	Test Case Type	Code	Status
Case1.c	PreProcessor statements	#include<stdio.h> #define count 10	Passed

Case2.c	Constants
	Keywords
	Punctuators
	Variables
		int a=25;
		
		char c='h';
		
		char arr1[10]="hello";
		
		char arr2[10]="hello;
			Passed
			
			Constants :
			
			Integer : 25
			
			Character :’h’
			
			String :“Hello”
			
			KeyWords :
			int char
			
			Identifiers:
			a c arr1 arr2
			
			Invalid String
			“hello

Case3.c	Relational Operators
	
	Logical Operators
	
	Arithmetic Operators
		int a=1,b=2,c=3,d;
		a++;
		--b;
		c=a+b;
		d+=a;
		d=a||b;
		c=a%b;
		d=a++c;
		a=c+/c;
		d=a>b;
			Passed
			
			Relational :
			>
			Logical :
			||
			
			Arithmetic :
			+ - / % \

Case4.c	Single Line Comments Multi Line Comments	// single line comment /// this is valid comment /* this is a multi line comment */	Passed Passed Passed

		/* this is a /* nested */ comment */ invalid comment */ /* invalid comment	Passed Error Passed Error Passed Error

Case5.c	Control Statements
	Looping Statements
		int a=1,b=2,c=10;
		
		if(a>b){
		printf("\nInside if");
		}
		else{
		printf("\nInside else");
		}
		
		if(b>a){
		if(b<c){
		printf("\nNested if ");
		}
		}
		
		for(int i=1;i<=10;i++){
		
		printf("\nIteration %d",i);
		}
		
		for(int i=0;i<10;i++){
		for(int J=0;J<20;J++){
		printf("\nNestedloop ");
		}
		}
			Passed
			
			Keywords
			If else for
			
			Datatype
			Int
			
			Identifiers
			J a b c i
			
			Procedures :
			
			printf ()

Case6.c	Function Declaration	struct student{
		int a;
		char c;
		};
		union teacher{
		int q;
		};
		void abc(){
		printf("\nHello World");
		}
		void main(){
		student *S;
		teacher *T;
			Passed
			
			Procedures :
			
			main()
			abc()
			printf()
			
			Identifiers :
			
			Student
			teacher

		abc(); }	

RESULTS

Case1.c

Purpose : Test case to check Preprocessor Directives declarations

Source Code :

```c
// Test case to check Pre Processor Directives declarations

#include <stdio.h>
#include "userHeader.h"
#define size 10

#include<<math.h>

int main(){
	int a=5;
	char c='H';
	char str[]="Hello World";
	return 1;

}
```

Output:

```
aswanth@hp-notebook: ~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ lex scanner.l
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ cc lex.yy.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ ./a.out Case1.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ |
```

Case2.c

Purpose : Test case to check int and char constants,string constants,keywords, punctuators and variables

Source Code :

```
// Test case to check int and char constants,string constants,
// keywords, punctuators and variables

#include<stdio.h>
void main(){
	int a=25;
	char c='h';
	char c='abc
	char arr1[10]="hello World";
	char arr2[10]="hello;
	return 1;


}
```

Output :

```
aswanth@hp-notebook: ~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ lex scanner.l
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ cc lex.yy.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ ./a.out Case2.c
Case2.c : 5 : Invalid Identifier
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ 
```

Case3.c

Purpose : Test case to check relational, logical and arithmetic operators

Source Code :

```
// Test case to check relational, logical and arithmetic operators

#include<stdio.h>
void main(){

		int a=1,b=2,c=3,d;
		a++;
		--b;
		c=a+b;
		d+=a;
		d=a||b;
		c=a%b;
		d=a++c;
		a=c+/c;

		d=a>b;


}
```

Output :

```
aswanth@hp-notebook: ~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ lex scanner.l
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ cc lex.yy.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ ./a.out Case3.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ |
```

Case4.c

Purpose : Test case to check single and multi line comments

Source Code :

```c
// test case to check single and multi line comments

#include<stdio.h>
void main()
{
	// this is a single line comment

	/// this is also valid comment

	/* this is
	a multi line
	comment */

	/* this is
	a /* nested */
	comment */

	invalid comment */

	/* invalid comment


}
```

Output :

```
aswanth@hp-notebook: ~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ lex scanner.l
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ cc lex.yy.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ ./a.out Case4.c
Case4.c : 15 : Nested Comment
Case4.c : 24 : Comment Does Not End
Case4.c : 24 : Nested Comment
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ |
```

Case5.c

Purpose : Test case to check control and looping statements

Source Code :

```c
1  // test case to check control and looping statements
2
3  #include<stdio.h>
4  void main()
5  {
6      int a=1,b=2,c=10;
7
8      if(a>b){
9          printf("\nInside if");
10     }
11     else{
12         printf("\nInside else");
13     }
14
15     if(b>a){
16         if(b<c){
17             printf("\nNested if ");
18         }
19     }
20     for(int i=1;i<=10;i++){
21         printf("\nIteration %d",i);
22     }
23     for(int i=0;i<10;i++){
24         for(int j=0;j<20;j++){
25             printf("\nNested loop ");
26          }
27     }
28
29 }
30
```

Output :

```
aswanth@hp-notebook: ~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ lex scanner.l
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ cc lex.yy.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ ./a.out Case5.c
Case5.c : 6 : Invalid Identifier
Case5.c : 20 : Invalid Identifier
Case5.c : 23 : Invalid Identifier
Case5.c : 24 : Invalid Identifier
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$
```

Case6.c

Purpose : Test case to check function declaration ,struct and union declaration

Source Code :

```c
// test case to check function decleration ,struct and union decleration

#include<stdio.h>
struct student{
    int a;
    char c;
};
union teacher{
    int q;
    int p;
}
void abc(){
    printf("\nHello World");
}
void main(){
    student *S;
    teacher *T;
    abc();
}
```

Output :

aswanth@hp-notebook: ~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases

```
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ lex scanner.l
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ cc lex.yy.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ ./a.out Case7.c
aswanth@hp-notebook:~/Desktop/Sem6/CD/Compiler Design Project/Lexical Analyzer/test cases$ 
```

CONCLUSION

We have made a scanner for C Programming Language. The following features have been taken care of :

1) Identification of Keywords, Identifiers, Operators (Relational, Logical and Arithmetic), Punctuators, Constants (Integer, Character and Float) and String Literals.
2) Arrays
3) Pointers
4) Single and Multi Line Comments
5) Data Types (int, float and char) with modifiers (unsigned and signed) and types (short,long).
6) Procedures

The following lexical errors have been handled :

1) Nested Comments
2) Invalid Identifiers
3) Invalid String
4) Balancing of Parentheses