Lexical Analyzer for the C Language

National Institute of Technology Karnataka, Surathkal

Date: 22 August, 2020

Submitted To:

Dr. P Santhi Thilagam
Department of Computer Science and Engineering
NITK, Surathkal

Group Members:

K Krishna Swaroop - 181CO125
Shreeraksha R Aithal - 181CO149
Swathi J S - 181CO155
Yerramaddu Jahnavi - 181CO260

ABSTRACT

Our aim is to design a compiler for a subset of C Programming Language. Lexical Scanner is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences . In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.

Programs that perform lexical analysis are called lexical analyzers or lexers. A lexer contains a tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Our scanner takes in a C program and build - symbol table, constant table and lists all tokens in the input program along with their line number. It also parses both kinds of comments (single line and multi lined) and reports any error with invalid/incomplete comments and strings (and character constant declaration).

Contents

INTRODUCTION

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 a 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 a 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 a 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, groups them into lexemes and produces the token as output.
● Enters the identified token into the symbol table.
● Strips out white spaces and comments from the source program.
● Correlates error messages with the source program i.e., displays error messages 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 SCRIPT

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:

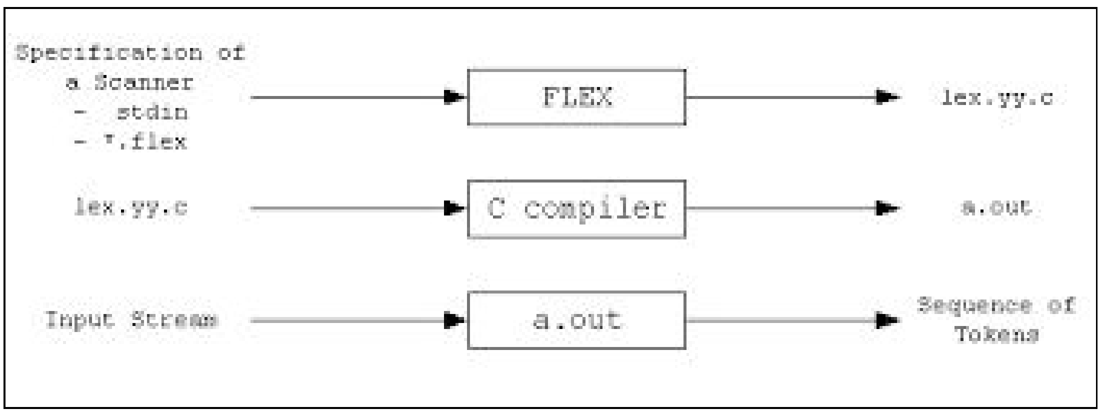

Fig.1

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.

● *.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 section

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 cause compile-time errors.

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

DESIGN OF PROGRAMS

LexicalScanner.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. The program code (LexicalScanner.l) analyzes and parses code for tokens and builds - ‘Symbol table’, ‘Constant table’, and ‘Tokes list’. When executing this file, it is first converted to lex.yy.c which is compiled to get the executable a.out .

Code of LexicalScanner.l

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

FILE *Smbl,*Cnst;                                // Smbl for symbol, Cnst for constants
char *Mul_comment,*inputFile, Sng_comment[1000];

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

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

%}

assignment =
arithmatic \+|\-|\*|\/|\%
datatype "int"|"char"|"float"|"void"
digit  [0-9]
keyword "auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"return"|"short"|
letter  [a-zA-Z]
logical \&|\^|\~
modifiers "long"|"short"
multilinecommentstart (\/\*)
multilinecommentend (\*\/)
newline "\n"
```

Fig.1

```
puncuator \(|\)|\[|\]|\;|\,|\:|\.
quote \'|\"|\\
relational >|<|<=|>=|!=|==
sign "signed"|"unsigned"
singlelinecomment (\/\/.*)
whitespace [ \t]+
identifier ({letter}({letter}|{digit})*)|"_"({letter}|{digit})+

%x COMMENT_DETECTION

%%

^#([-a-zA-Z0-9.]|{relational}|{whitespace})* {
        insertToTable(yytext,'d'); //preprocessor directive rule
        printf("%s : %d : Preprocessor rule found - %s\n",inputFile,lineNumber,yytext);
        }

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

^{datatype}{whitespace}*{identifier}\(.*\)  {   int i,j=0;char temp1[50]={'\0'}, temp2[50]={'\0'};
                                                        for(i=0;yytext[i]!=' ';i++)
                                                        {
                                                                temp1[i] = yytext[i];
                                                        }

                                                        insertToTable(temp1,'k');
                                                        for(;yytext[i]!='(';i++){
                                                                temp2[j]=yytext[i];
                                                                j++;
                                                        }

                                                        insertToTable(temp2,'j');  //procedure rule
                                                }
```

Fig.2

```
{identifier}\[{digit}*\] {      int i,j=0;char temp[50]={'\0'};

                                                for(i=0;yytext[i]!='[';i++)
                                                {
                                                        temp[j] = yytext[i];
                                                        j++;
                                                }

                                                        insertToTable(temp,'a'); // array rule
                                        }

\*{identifier} {        int i,j=0;char temp[50]={'\0'};
                                for(i=1;yytext[i]!='\0';i++)
                                {
                                        temp[j++] = yytext[i];
                                }
                                        insertToTable(temp,'q'); // pointer rule
                        }

{identifier} {
        insertToTable(yytext,'i'); // variable rule
        printf("%s : %d : Identifier found - %s\n",inputFile,lineNumber,yytext);
        }
{digit}+ {
        insertToTable(yytext,'c'); //integer constants rule
        printf("%s : %d : Integer found - %s\n",inputFile,lineNumber,yytext);
        }
{digit}+({letter}|{digit})+|"_" { printf("%s : %d : Invalid Identifier - %s\n",inputFile,lineNumber,yytext); } // invalid identifier


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

Fig.3

```


\"(.)*\" {
        insertToTable(yytext,'s'); //string constants rule
        printf("%s : %d : String constant found - %s\n",inputFile,lineNumber,yytext);
        }
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}|{digit})\' {        char temp[50]={'\0'};
                                                        temp[0] = yytext[1];
                                                        printf("%s : %d : Character constant found - '%c'\n",inputFile,lineNumber,temp[0]);
                                                        insertToTable(temp,'z');  // character constant rule
                                                }
\'({letter}|{digit})({letter}|{digit})+\' {printf("%s : %d : Invalid Character constant - %s\n",inputFile,lineNumber,yytext);}

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

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

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

{singlelinecomment} {
        printf("%s : %d : Single comment found\n",inputFile,lineNumber);
        storeSingleLineComment(yytext);
        }

```

Fig.4

```
{multilinecommentstart}  {
                BEGIN(COMMENT_DETECTION);
                nc++;
                commLine++;
                storeMultiLineComment("\n\n");
            }

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

<COMMENT_DETECTION>{multilinecommentend} {
                            if(nc>0)
                                nc--;
                            else
                                printf("%s : %d : Error:Closing comment found before opening\n",inputFile,lineNumber);

                            if(nc==0){
                                printf("%s : %d : Multi-line comment found\n",inputFile,lineNumber);
                                BEGIN(INITIAL);
                            }
                    }

<COMMENT_DETECTION>\n   {
                 commLine++;
                 lineNumber++;
                 storeMultiLineComment("\n");
                }
```

Fig.5

```
<COMMENT_DETECTION>. {storeMultiLineComment(yytext);}


%%

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

        Smbl=fopen("Symbol Table.txt","w");                              //File to write and save Symbol Table for given input
        fprintf(Smbl,"\t\t\t\t\t\t\t\t\t\t Symbol Table:\n\n\n\n \t\tLine Number\t\t\tLexeme\t\t\t\tType\t\t\tAttribute Value\n");

        Cnst=fopen("Constant Table.txt","w");                           // File to write and Constant Table for given input
        fprintf(Cnst,"\t\t\t\t\t\t\t\t\t\t Constants Table:\n\n\n\n \t\tLine Number\t\t\tLexeme\t\t\tType\t\t\tAttribute Value\n");

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

        yylex();

        if(nc!=0)
                printf("%s : %d : Comment does not end\n",inputFile,lineNumber);

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

        fprintf(yyout,"\n");
        if(flag > 0)
        {
                        commLine = 0;
                        fprintf(yyout,"\n\nComment (%d lines):\n",commLine);
                        printf("\n%s : %d Nested comment(s) found\n",inputFile,flag);
        }
```

Fig.6

```
        else
        {
                int i;
                fprintf(yyout,"\n\nMultiLineComment (%d lines):\n\n",commLine);
                fputs(Mul_comment,yyout);
                fprintf(yyout,"\n\nSingleLineComment :\n\n");
                fputs(Sng_comment,yyout);
        }

        fclose(yyout);
        fclose(Smbl);
        fclose(Cnst);
}

void storeSingleLineComment(char *yytext)
{
        int length = strlen(yytext);
        int i, j;
        char *extra;
        extra = (char*)malloc((length+1)*sizeof(char));
        for(j=2,i=0;yytext[j]!='\0';j++,i++)
        {
                extra[i] = yytext[j];
        }
        strcat(extra,"\n");     //To print on new line
        strcat(Sng_comment,extra);      //Copy to list of single comments
}
void storeMultiLineComment(char *yytext)
{
    int len1, len2;
    char *extra;

    len1 = strlen(Mul_comment);
    len2 = strlen(yytext);
```

Fig.7

```c
    extra = (char*)malloc((len1+1)*sizeof(char));
    strcpy(extra,Mul_comment);
    Mul_comment = (char*)malloc((len1+len2+1)*sizeof(char));
    strcat(extra,yytext);
    strcpy(Mul_comment,extra);
}
void insertToTable(char *yytext,char type)
{
    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;
```

Fig.8

```

        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;
            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;
```

Fig.9

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

                var++;
            }
    }

        if(type =='i' || type == 'a' || type == 'q' || type=='j')
        {
                fprintf(Smbl,"\n%25d%30s%30s%30d",lineNumber,yytext,token,i);
        }
        switch(type)
        {
                case 'c' : fprintf(Cnst,"\n%25d%30s%20s%30d",lineNumber,yytext,"int",i);
                                break;

                case 'f' : fprintf(Cnst,"\n%25d%30s%20s%30d",lineNumber,yytext,"float",i);
                                break;

                case 'z' : fprintf(Cnst,"\n%25d%30s%20s%30d",lineNumber,yytext,"char",i);
                                break;
        }

        fprintf(yyout,"\n%25d%30s%30s%30d",lineNumber,yytext,token,i);
}

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

Fig.10

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 LexicalScanner.l)
<filename2> is the C source program for which Lexical Analysis is done. (here BasicInput.c)

BasicInput.c

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

Code

```c
#include <stdio.h>
int main()
{
    /* Hello,
    This is
    a sample
    multi-line
    comment*/
    /* This is a basic program to
    check the proper execution
    of Lexical Scanner*/
    struct node{
    int a;
    char name;
    };

    /*This is a another
    multi-line comment*/

    //Following are basic constants and identifiers
    float fl = 5.01;
    char l1 = 'a', l2 ='1';
    int _a=0,b=0,c=5;
    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 = "Hello";
    char *name = "John Doe";

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

Fig.11

Output

```
E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>flex LexicalScanner.l

E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>gcc lex.yy.c

E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>a.exe BasicInput.c
BasicInput.c : 1 : Preprocessor rule found - #include <stdio.h>
BasicInput.c : 8 : Multi-line comment found
BasicInput.c : 11 : Multi-line comment found
BasicInput.c : 12 : Identifier found - node
BasicInput.c : 13 : Identifier found - a
BasicInput.c : 14 : Identifier found - name
BasicInput.c : 18 : Multi-line comment found
BasicInput.c : 20 : Single comment found
BasicInput.c : 21 : Identifier found - f1
BasicInput.c : 22 : Identifier found - l1
BasicInput.c : 22 : Character constant found - 'a'
BasicInput.c : 22 : Identifier found - l2
BasicInput.c : 22 : Character constant found - '1'
BasicInput.c : 23 : Identifier found - _a
BasicInput.c : 23 : Integer found - 0
BasicInput.c : 23 : Identifier found - b
BasicInput.c : 23 : Integer found - 0
BasicInput.c : 23 : Identifier found - c
BasicInput.c : 23 : Integer found - 5
BasicInput.c : 26 : Identifier found - scanf
BasicInput.c : 26 : String constant found - "%d %d"
BasicInput.c : 26 : Identifier found - a
BasicInput.c : 26 : Identifier found - b
BasicInput.c : 27 : Identifier found - sum
BasicInput.c : 27 : Integer found - 0
BasicInput.c : 28 : Identifier found - sum
BasicInput.c : 28 : Identifier found - a
BasicInput.c : 28 : Identifier found - b
BasicInput.c : 29 : Identifier found - printf
BasicInput.c : 29 : String constant found - "\n Sum : %d \n"
BasicInput.c : 29 : Identifier found - sum
BasicInput.c : 31 : Single comment found
BasicInput.c : 32 : String constant found - "Hello"
BasicInput.c : 33 : String constant found - "John Doe"
BasicInput.c : 35 : Single comment found
BasicInput.c : 36 : Integer found - 1
BasicInput.c : 40 : Identifier found - printf
BasicInput.c : 40 : String constant found - "\nTest function "
BasicInput.c : 41 : Identifier found - printf
BasicInput.c : 41 : String constant found - "\nInside abc() function\n"

E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>_
```

Fig.12

Symbol Table.txt:

Symbol Table - Notepad

File Edit Format View Help

Symbol Table:

Line Number	Lexeme	Type	Attribute Value
2	main	Procedure	2
12	node	Identifier	5
13	a	Identifier	6
14	name	Identifier	9
21	f1	Identifier	12
22	l1	Identifier	15
22	l2	Identifier	17
23	_a	Identifier	19
23	b	Identifier	21
23	c	Identifier	22
24	arr	Array	24
25	ptr	Pointer	25
26	scanf	Identifier	26
26	a	Identifier	6
26	b	Identifier	21
27	sum	Identifier	31
28	sum	Identifier	31
28	a	Identifier	6
28	b	Identifier	21
29	printf	Identifier	33
29	sum	Identifier	31
32	name	Pointer	9
33	name	Pointer	9
38	abc	Procedure	39
40	printf	Identifier	33
41	printf	Identifier	33

Fig.13

Constant Table.txt:

Constant Table - Notepad

File Edit Format View Help

Constants Table:

Line Number	Lexeme	Type	Attribute Value
21	5.01	float	14
22	a	char	6
22	1	char	18
23	0	int	20
23	0	int	20
23	5	int	23
27	0	int	20
36	1	int	18

Fig.14

Token List.txt:

Tokens List - Notepad

File Edit Format View Help

Token List:

Line Number	Lexeme	Token	Attribute Value
1	#include <stdio.h>	Preprocessor Statement	0
2	int	Keyword	1
2	main	Procedure	2
3	{	Punctuator	3
12	struct	Keyword	4
12	node	Identifier	5
12	{	Punctuator	3
13	int	Keyword	1
13	a	Identifier	6
13	;	Punctuator	7
14	char	Keyword	8
14	name	Identifier	9
14	;	Punctuator	7
15	}	Punctuator	10
15	;	Punctuator	7
21	float	Keyword	11
21	fl	Identifier	12
21	=	Assignment Op	13
21	5.01	Float Constant	14
21	;	Punctuator	7
22	char	Keyword	8
22	l1	Identifier	15
22	=	Assignment Op	13
22	a	Character Constant	6
22	,	Punctuator	16
22	l2	Identifier	17
22	=	Assignment Op	13
22	1	Character Constant	18
22	;	Punctuator	7
23	int	Keyword	1
23	_a	Identifier	19
23	=	Assignment Op	13
23	0	Integer Constant	20
23	,	Punctuator	16
23	b	Identifier	21
23	=	Assignment Op	13

Fig.15

Tokens List - Notepad

File Edit Format View Help

22	=	Assignment Op	13
22	1	Character Constant	18
22	;	Punctuator	7
23	int	Keyword	1
23	_a	Identifier	19
23	=	Assignment Op	13
23	0	Integer Constant	20
23	,	Punctuator	16
23	b	Identifier	21
23	=	Assignment Op	13
23	0	Integer Constant	20
23	,	Punctuator	16
23	c	Identifier	22
23	=	Assignment Op	13
23	5	Integer Constant	23
23	;	Punctuator	7
24	int	Keyword	1
24	arr	Array	24
24	;	Punctuator	7
25	char	Keyword	8
25	ptr	Pointer	25
25	;	Punctuator	7
26	scanf	Identifier	26
26	(	Punctuator	27
26	"%d %d"	String Literal	28
26	,	Punctuator	16
26	&	Logical Op	29
26	a	Identifier	6
26	,	Punctuator	16
26	&	Logical Op	29
26	b	Identifier	21
26	)	Punctuator	30
26	;	Punctuator	7
27	int	Keyword	1
27	sum	Identifier	31
27	=	Assignment Op	13
27	0	Integer Constant	20
27	;	Punctuator	7
28	sum	Identifier	31
28	=	Assignment Op	13
28	a	Identifier	6
28	+	Arithmetic Op	32

Fig.16

*Tokens List - Notepad

File Edit Format View Help

26	,	Punctuator	16
26	&	Logical Op	29
26	a	Identifier	6
26	,	Punctuator	16
26	&	Logical Op	29
26	b	Identifier	21
26	)	Punctuator	30
26	;	Punctuator	7
27	int	Keyword	1
27	sum	Identifier	31
27	=	Assignment Op	13
27	0	Integer Constant	20
27	;	Punctuator	7
28	sum	Identifier	31
28	=	Assignment Op	13
28	a	Identifier	6
28	+	Arithmetic Op	32
28	b	Identifier	21
28	;	Punctuator	7
29	printf	Identifier	33
29	(	Punctuator	27
29	"\n Sum : %d \n"	String Literal	34
29	,	Punctuator	16
29	sum	Identifier	31
29	)	Punctuator	30
29	;	Punctuator	7
32	char	Keyword	8
32	name	Pointer	9
32	=	Assignment Op	13
32	"Hello"	String Literal	35
32	;	Punctuator	7
33	char	Keyword	8
33	name	Pointer	9
33	=	Assignment Op	13
33	"John Doe"	String Literal	36
33	;	Punctuator	7
36	return	Keyword	37
36	1	Integer Constant	18
36	;	Punctuator	7
37	}	Punctuator	10
38	void	Keyword	38
38	abc	Procedure	39

Fig.17

```
*Tokens List - Notepad
File  Edit  Format  View  Help
        36                        1            Integer Constant          18
        36                        ;            Punctuator                7
        37                        }            Punctuator                10
        38                     void            Keyword                   38
        38                      abc            Procedure                 39
        39                        {            Punctuator                3
        40                   printf            Identifier                33
        40                        (            Punctuator                27
        40        "\nTest function "           String Literal            40
        40                        )            Punctuator                30
        40                        ;            Punctuator                7
        41                   printf            Identifier                33
        41                        (            Punctuator                27
        41   "\nInside abc() function\n"       String Literal            41
        41                        )            Punctuator                30
        41                        ;            Punctuator                7
        42                        }            Punctuator                10


MultiLineComment (10 lines):

Â

 Hello,
        This is
        a sample
        multi-line
        comment

 This is a basic program to
        check the proper execution
        of Lexical Scanner

This is a another
        multi-line comment

SingleLineComment :

Following are basic constants and identifiers
Enter Name
End the main function
```

Fig.18

Test Cases:

Input for: For loop with valid and invalid strings

```
12 lines (11 sloc)  135 Bytes

#include<stdio.h>

int main(){
        int a;
        char str[24] = "Hi!";
        for(a = 0; a<4; a++){
                if(a == 3){
                        str = "Bye!;
                };
        }
        return 0;
}
```

Fig.19

Output for: For loop with valid and invalid strings

Fig.20

Input for: Various forms of multi-line comment

20 lines (15 sloc) | 186 Bytes

```
#include<stdio.h>

int main(){
	//This is a simple single comment

	/* This is a
	multi-line comment */

	char c = 'C';

	/*
	This is
	a
	/* nested
	comment*/
	*/

	int a;
	return 0;
}
```

Fig.21

Output for: Various forms of multi-line comment

```
E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>flex LexicalScanner.l

E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>gcc lex.yy.c

E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>a.exe Input2.c
Input2.c : 1 : Preprocessor rule found - #include<stdio.h>
Input2.c : 4 : Single comment found
Input2.c : 7 : Multi-line comment found
Input2.c : 9 : Identifier found - c
Input2.c : 9 : Character constant found - 'C'
Input2.c : 14 : Nested Comment found
Input2.c : 16 : Multi-line comment found
Input2.c : 18 : Identifier found - a
Input2.c : 19 : Integer found - 0

Input2.c : 1 Nested comment(s) found

E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>flex LexicalScanner.l
```

Fig.22

Input for: Sample C program for binary search.

```c
#include<stdio.h>

int main(){
	int left, target, right, mid;
	int array[5] = { 0, 1, 5, 7, 10};
	left = 0;
	right = 5;
	target = 1;
	while(left < right){
		mid = (left + right)/2;
		if(array[mid] == target){
			printf("Found");
			break;
		}
		if(array[mid]>target)
			left = mid + 1;
		right = mid - 1;
	}
	if(left>right)
		printf("Not found");
	return 0;
}
```

Fig.23

Output for: Sample C program for binary search.

```
E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>gcc lex.yy.c

E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>a.exe Input3.c
Input3.c : 1 : Preprocessor rule found - #include<stdio.h>
Input3.c : 4 : Identifier found - left
Input3.c : 4 : Identifier found - target
Input3.c : 4 : Identifier found - right
Input3.c : 4 : Identifier found - mid
Input3.c : 5 : Integer found - 0
Input3.c : 5 : Integer found - 1
Input3.c : 5 : Integer found - 5
Input3.c : 5 : Integer found - 7
Input3.c : 5 : Integer found - 10
Input3.c : 6 : Identifier found - left
Input3.c : 6 : Integer found - 0
Input3.c : 7 : Identifier found - right
Input3.c : 7 : Integer found - 5
Input3.c : 8 : Identifier found - target
Input3.c : 8 : Integer found - 1
Input3.c : 9 : Identifier found - left
Input3.c : 9 : Identifier found - right
Input3.c : 10 : Identifier found - mid
Input3.c : 10 : Identifier found - left
Input3.c : 10 : Identifier found - right
Input3.c : 10 : Integer found - 2
Input3.c : 11 : Identifier found - array
Input3.c : 11 : Identifier found - mid
Input3.c : 11 : Identifier found - target
Input3.c : 12 : Identifier found - printf
Input3.c : 12 : String constant found - "Found"
Input3.c : 15 : Identifier found - array
Input3.c : 15 : Identifier found - mid
Input3.c : 15 : Identifier found - target
Input3.c : 16 : Identifier found - left
Input3.c : 16 : Identifier found - mid
Input3.c : 16 : Integer found - 1
Input3.c : 17 : Identifier found - right
Input3.c : 17 : Identifier found - mid
Input3.c : 17 : Integer found - 1
Input3.c : 19 : Identifier found - left
Input3.c : 19 : Identifier found - right
Input3.c : 20 : Identifier found - printf
Input3.c : 20 : String constant found - "Not found"
Input3.c : 21 : Integer found - 0

E:\Github\Mini-C-Compiler\Part 1 - Lexical Analyser>
```

Fig.24

The flex script recognises 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 unclean integers and floating point numbers
- ❏ Errors for incomplete strings
- ❏ Keywords
- ❏ Identifiers

DFA for the above mentioned regex

1. Digit

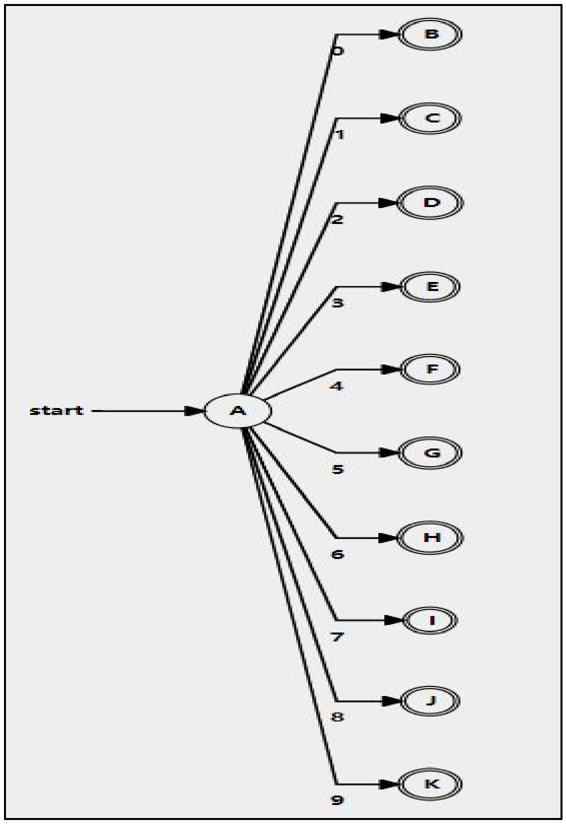

Fig.25

2. Letter

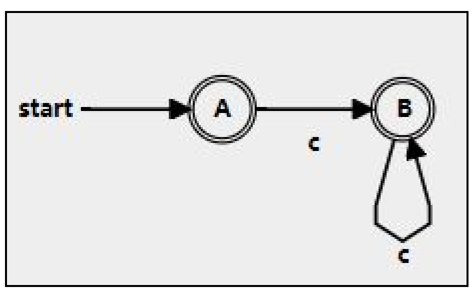

Fig.26

3. Datatype

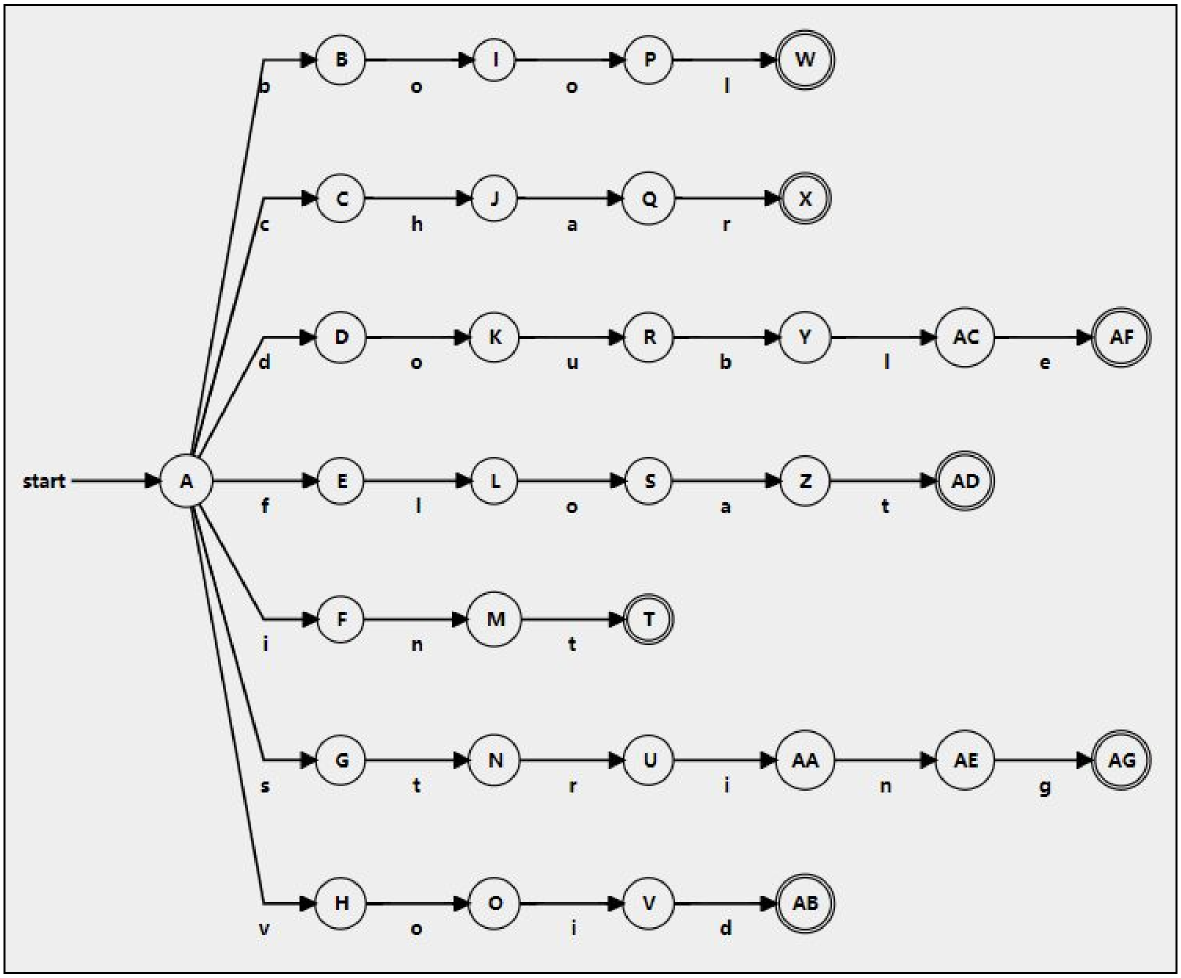

Fig.27

4. Sign

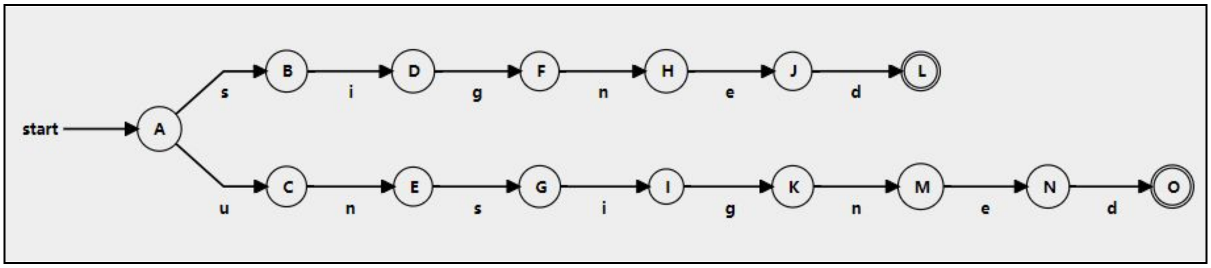

Fig.28

5. Modifiers

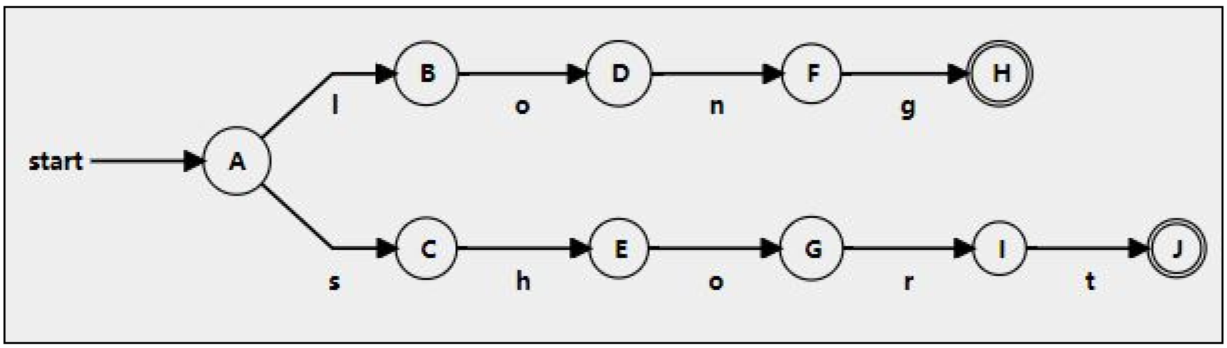

Fig.29

6. Relational

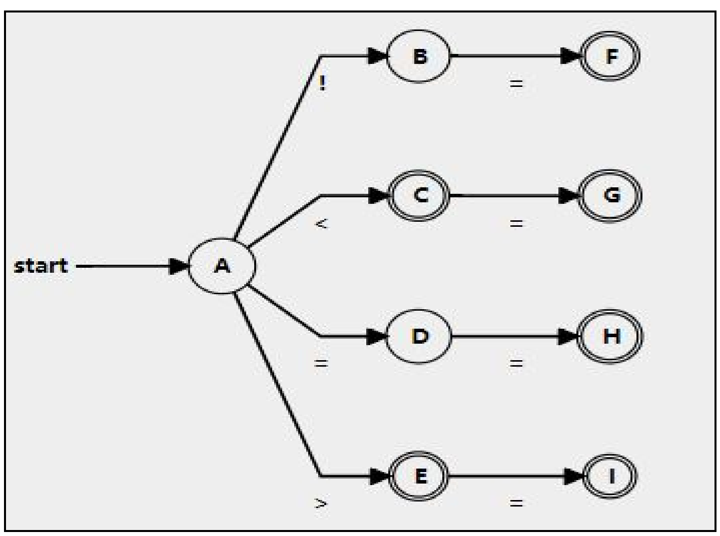

Fig.30

7. Logical

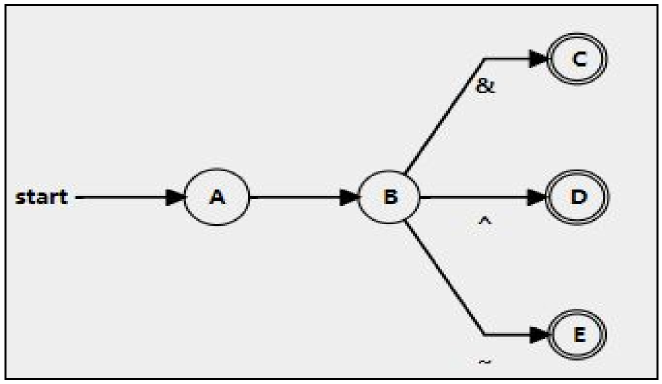

Fig.31

8. Arithmetic

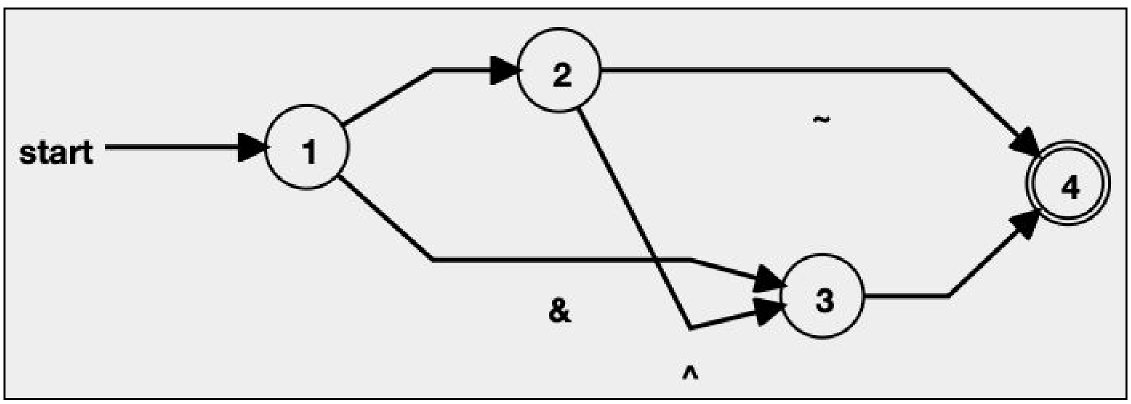

Fig.32

9. Assignment

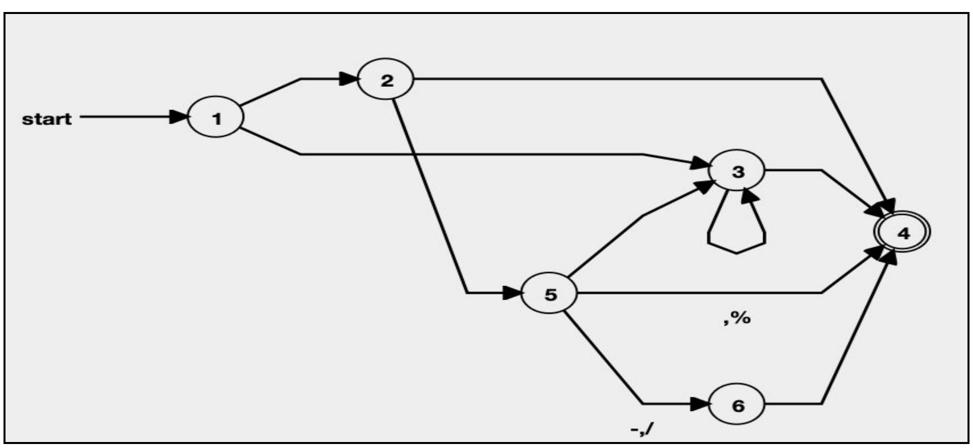

Fig.33

10. Quote

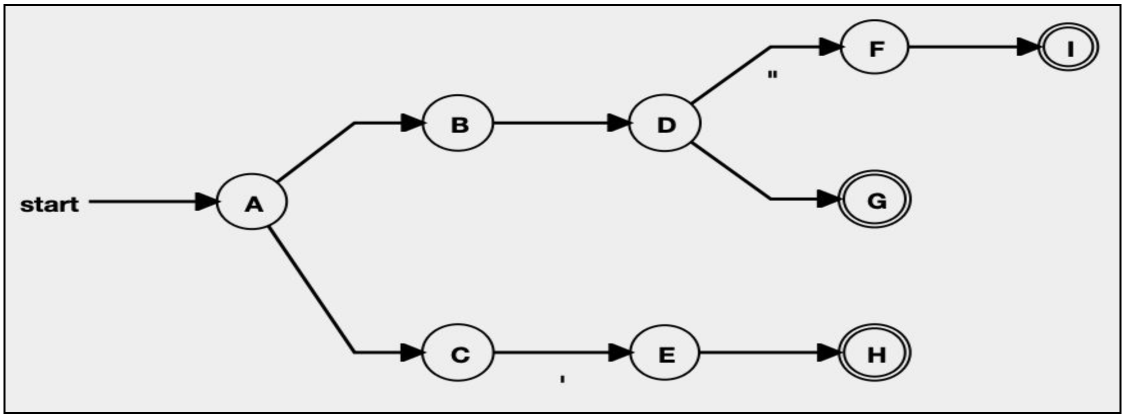

Fig.34

11. Whitespace

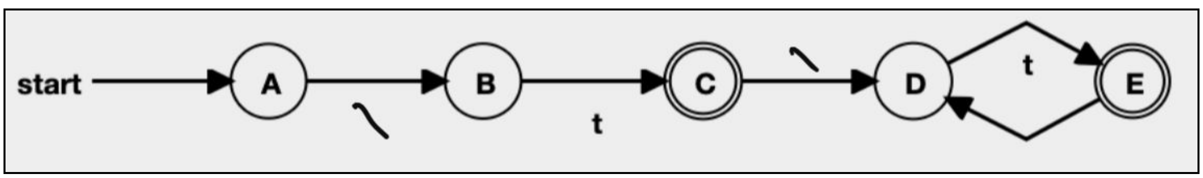

Fig.35

12. Newline

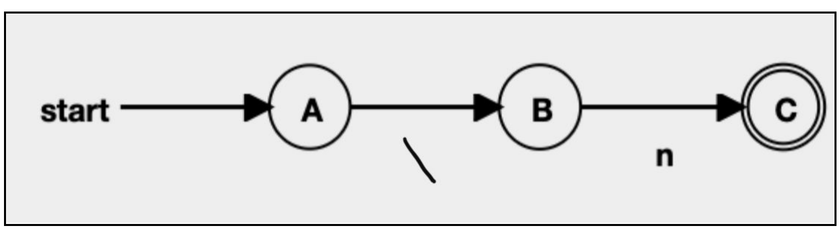

Fig.36

13. Single line comment

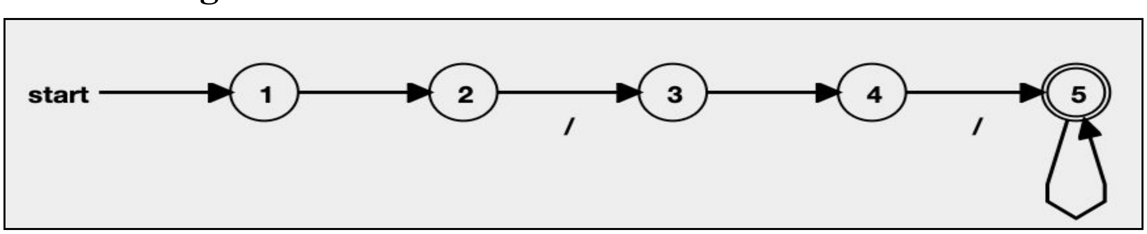

Fig.37

14. Multiline comment start

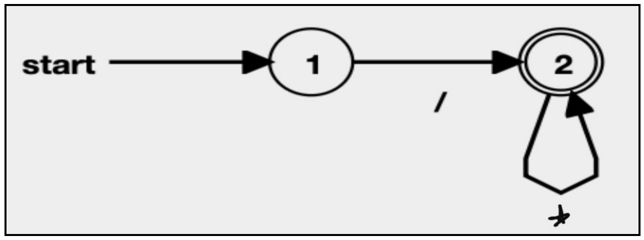

Fig.38

15. Multiline comment end

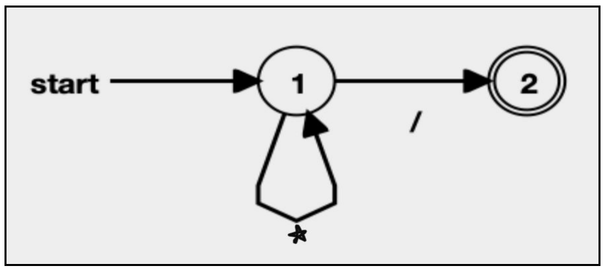

Fig.39

Keywords accounted for:

Following are the keywords recognised by the Lexical Analyser:

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

Test cases:

❏ Without Errors:

Serial No	Test Case	Expected Output	Status
1.PreProcessor Statements	#include<stdio.h>	input.c : 1 : Preprocessor rule found - #include <stdio.h>	Passed
2.Constants Keywords Punctuators	int a = 25; char b = ‘c’;	Keyword:int,char Identifier:a,b Operator:=	Passed

<table>
<tr><td>Variables</td><td></td><td>Integer:25
Character:c</td><td></td></tr>
<tr><td>3.Operators</td><td>int a=1,b=2;
a++;
--b;
a=a+b;
a=a||b;
b=a>b;</td><td>Keyword:int
Identifier:a,b
Operator:=,++,--,+,||,>
Integer:1,2</td><td>Passed</td></tr>
<tr><td>4.Comments</td><td>//SingleLine
/* Multi-line
Comment
*/</td><td>Singlelinecomment:
SingleLine
MultilineComment:
Multi-line
Comment</td><td>Passed</td></tr>
<tr><td>5.Control Statements</td><td>Int a=1,b=5;
if(a>b)
printf(“A is greater”);
Else
printf(“B is greater”);</td><td>Keyword: int,if,else
Identifier:a,b
Operator:>
Procedures:printf()</td><td>Passed</td></tr>
<tr><td>6.Function Declaration</td><td>int fun(){
printf(“Hello World”);
return 2;
}
int main(){
fun();
return 1;
}</td><td>Procedures:fun(),printf(),
main()</td><td>Passed</td></tr>
</table>

- With Errors:

Serial No	Test case	Expected Output	Status
1.Invalid String	a=”hello;	Identifier:a String does not end: ”hello	Passed

2. Invalid Character Constant	A = ‘Sam’	Identifier:A Invalid Character constant: ‘Sam’	Passed
3.Invalid Comment	/*Comment	Comment does not end:/*Comment	Passed
4.Missing closing braces	int main(){	Unbalanced Braces {}	Passed
5. Missing Parenthesis	int main){ }	(No error is shown)	Failed

Implementation

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

- **The Regex for Identifiers:** The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.
- **Multiline comments should be supported:** 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, character constants etc.
- **Error Handling for Incomplete String:** Open and close quote(both single and double) missing, both kinds of errors have been handled in the rules written in the script.
- **Error Handling for Invalid Constant Character:** Constants of size greater than one inside single quotes are termed as invalid character constants and the scanner throws error for the same.
- **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.

To list all tokens, identifiers and constants in a given file, we have designed a separate table which records respective values when encountered.

- Each token is associated with an attribute value - a value to recognise the repetition of the same token elsewhere in the file. This is maintained using a linked list using the structure Node.
- For constants, we create a new ‘Constant Table.txt’ at the beginning of execution. Every constant parsed gets written into this file along with its line number, type and attribute number.
- For identifiers, we create a new ‘Symbol Table.txt’ at the beginning of execution. Every identifier parsed gets written into this file along with its line number, type and attribute number.
- For tokens, we create a new ‘Token List.txt’ at the beginning of execution. Every token parsed gets written into this file along with its line number, type and attribute number.
- In the end, after EOF is encountered, all the single and multi-line comments are appended in ‘Token List.txt’.

Results:

The scanner is currently able to detect nested comments, basic identifiers, string and character, integer and floating point constant literals and other special symbols. It throws error for imbalance/incomplete braces, strings and invalid character constants.

Future work

The flex script presented in this report takes care of all the rules of 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 C language and making it more efficient.

We plan to extend the scanner by adding additional features like parsing for other kinds of parenthesis, able to verify for data-types associated with an identifier and its assigned value, etc.

References

1) Aho A.V, Sethi R, and Ullman J.D. **Compilers: Principles, Techniques, and Tools**. Addison-Wesley, 1986.
2) Appel A.W., and Palsberg J. **Modern Compiler Implementation in Java.** Cambridge University Press, 2002.
3) **Lex and Yacc Tutorial**, Tom Niemann