National Institute of Technology Karnataka Surathkal

Date: 20 January 2018
Submitted To: Ms. Sushmita

Project 1: SCANNER FOR C LANGUAGE

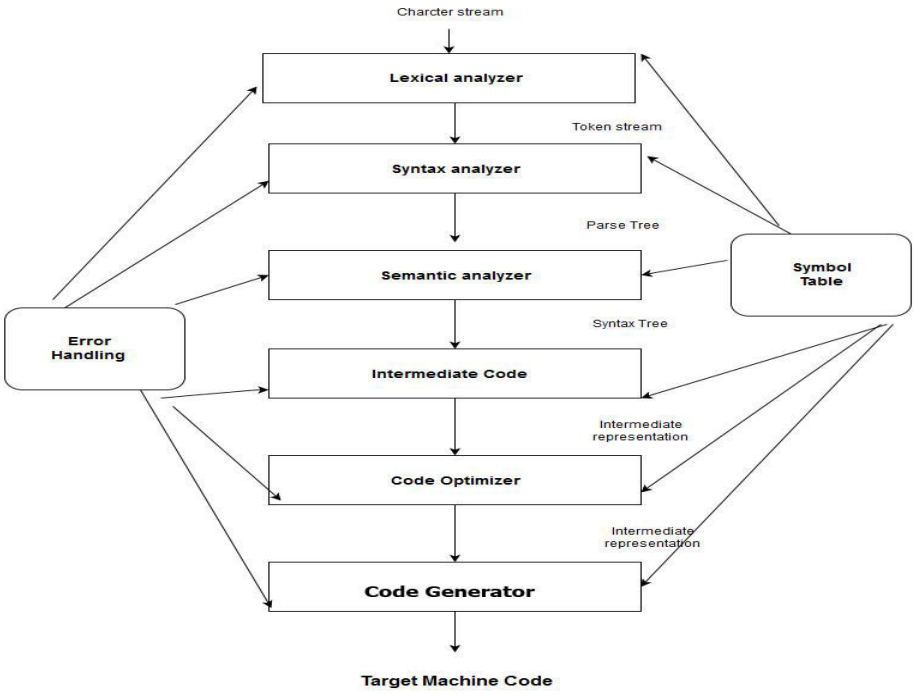
Group Members:

Roll No	Name	E-mail	Contact No.
15CO230	Naladala Indukala	15co230.indukala@nitk.edu.in	9880692703
15CO236	R Aparna	15co236.aparna@nitk.edu.in	8105348941

Table of Contents

1. INTRODUCTION TO COMPILER

Compiler is a software which converts a program written in high level language (source language) to low level language (object/target/machine language).



Analysis Phase: An intermediate representation is created from the given source code. This phase can be further decomposed into:
1. Lexical Analyzer (divides the program into “tokens”)
2. Syntax Analyzer (recognizes “sentences” in the program using syntax of language)
3. Semantic Analyzer (checks static semantics of each construct)

Synthesis phase: Equivalent target program is created from the intermediate representation. It has three parts:
4. Intermediate Code Generator (generates “abstract” code)
5. Code Optimizer (optimizes the abstract code)
6. Final Code Generator (translates abstract intermediate code into specific machine instructions)

2. PHASES OF COMPILER

2.1 LEXICAL ANALYSIS (SCANNER)

The scanner will be developed using flex. It is aimed to support nested comments. The scanner must be able to return appropriate error messages that are as meaningful as possible. A symbol table and a constants table must be generated using hash organisation.

To carry out lexical analysis, the lexical components of the grammar will be identified using language specifications. The desired output of this phase will be a stream of tokens for the input source program. Errors, if any, must be identified and reported appropriately.

At the lexical level, C has the following tokens:

- Operators: *, /, +, -, =, <>, >, <, >=, <=, &, |, *=, ^, +=, -=, !=, ++, --, >>, <<, >>=, <<=.
- Keywords: auto,break,case,char,const,continue,default,do,else,double,enum,extern,float,for,goto,if,long,int,printf,scanf,register,return,short,signed,sizeof,static,struct,switch,typedef,union,unsigned,void,volatile,while
- Identifiers: String of alphanumeric characters that begins with an alphabet character or an underscore character.
- Whitespace: Any whitespace outside of strings is ignored. Whitespace consists of spaces, tab, newline, or comments. A comment starts with /* and ends with */. Comments can be nested. Single line comments start with //.
- Special Characters: \, {, }, (,), [,], ;, :, $, #, ., ”
- Constants: Floating point and integer constants
- String Literals: Sequence of characters from the source character set enclosed in double quotation marks.
- Pre processor Directives: #include<stdio.h>, #include<stdlib.h> , #include<math.h>

2.2 GENERATION OF SYMBOL TABLE:

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities.

Generated tokens are used to create symbol table. It can be used to store names of all entities in a structured form at one place, check if the variable is declared or not, implement type checking by verifying whether assignments and expressions in the source code are semantically correct, determine the scope of an entity (scope resolution).

The symbol table provides the following functions:

- insert(): This operation is used to add information in the symbol table about unique names occurring in the source code. An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope and type of the symbol. We use this operation to place unique tokens in the symbol table which are identified using the analysis phase.
- lookup(): lookup() operation is used to search an entity name in the symbol table, determine if the symbol exists in the table, if it is declared before it is being used, if the name is used in the scope, if the symbol is initialized, if the symbol is declared multiple times.

We use the lookup function at various stages of the compiler to get information about various attributes.

2.3 SYNTAX ANALYSIS (PARSER)

The syntax analysis will be carried out using Yacc/Bison parser generator and the appropriate grammar specifications. The next step would be to integrate the scanner and parser which will involve updating the respective tables whenever a new token is identified in the source code. Errors must be reported with the line number and the program must be aborted.

2.4 SEMANTIC ANALYSIS (SEMANTIC CHECKER)

The semantic checker must primarily perform two functions, namely:

- Use of Symbol table for checking and code generation: This could be carried out with the use of a stack. The symbol table should now include information about symbols like variables, procedures and parameters.

- Implementation of type checking: Type checking, scope analysis and declaration processing will be focused on here.

Semantic errors like errors in variable declarations, function declarations, call expressions etc are identified and reported along with the line numbers, and consequently, the program is aborted.

2.5 INTERMEDIATE CODE GENERATION

The three-address code for the original grammar will be generated. This, in turn, would be converted to MIPS assembly code, if required.

3. LEXICAL ANALYSIS

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function that performs lexical analysis is called a **lexical analyzer**, **lexer**, **tokenizer** or **scanner**.

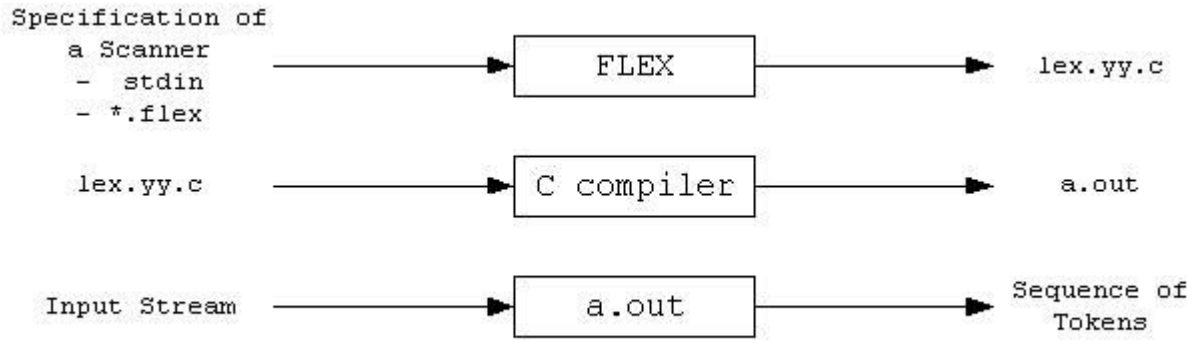
Our scanner supports the following features:

- Single line/multiline comments are identified and ignored. Nested comments are supported.
- Whitespaces, newlines are ignored.
- Keywords, identifiers, pre-processor directives, operators and special characters are detected.
- It identifies constants and classifies them as float or integer.
- It returns appropriate error message when string literal does not end till EOF.
- It returns appropriate error message if comment does not end till EOF.
- It identifies invalid character sequences (improper identifier names).

Our lexical analyzer also detects functions and arrays though the actual lexical analyzer for C language doesn't.

Flex Script

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. Instead of writing a scanner from scratch, you only need to identify the *vocabulary* of C language and write a specification of patterns using regular expressions 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 then, 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 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

Longest Matching Rule

- When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.
- There are some predefined rules for every lexeme (a sequence of characters (alphanumeric) in a token scanned) to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.
- The Longest Match Rule states that the lexeme should be determined based on the longest match among all the tokens available.

- And when there is more than one longest match, we will give more preference to the one which was written first while writing the rules.

4. LEXICAL ANALYZER CODE

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct symbol
{
char values[100];
int line_number;
char type[15];
};
int k=0, j=0, i=0;
typedef struct symbol s;
s ident[100];
s constant[100];
int line=1, comment=0, quote_error=0;
void insert(int a, char val[], char type[])
{
if(a==0)
{
for(k=0; k<j; k++)
{
if(strcmp(val, constant[k].values)==0)
return;
}
strcpy(constant[j].values, val);
constant[j].line_number=line;
```

```
strcpy(constant[j].type, type);
j++;
}
else if(a==1)
{
for(k=0; k<i; k++)
{
if(strcmp(val, ident[k].values)==0)
return;
}
strcpy(ident[i].values, val);
strcpy(ident[i].type, type);
i++;
}
}
%}

digit [0-9]
alpha [a-zA-Z]
alphanum [0-9a-zA-Z]
keyword
auto|break|case|char|const|continue|default|do|else|double|enum|extern|floa
t|for|goto|if|long|int|printf|scanf|register|return|short|signed|sizeof|sta
tic|struct|switch|typedef|union|unsigned|void|volatile|while
identi ({alpha}|_)({alpha}|{digit}|_)*
white [ \t]+
sign [\+\-]?
invalid_identifier {digit}+{alpha}({alpha}|{digit})*
%%

\n {line++;}
```

```
"//".* {
printf("Single line comment\n");
}
\/\* {
comment++;
printf("Multiline comment starts\n");
}

\*\/ {
if(comment>0)
{
comment--;
printf("Closing of multiline comment\n");
}
else
printf("Multiline comment doesn't have a start, Error occured at line %d\n",line);
}

#include<.*> {
if(comment<=0)
printf("Preprocessor directive:\t\t\t\t%s\n", yytext);
}

{keyword} {
if(comment<=0)
{
insert(1, yytext, "keyword");
printf("Keyword:\t\t\t\t%s\n", yytext);
}
}
```

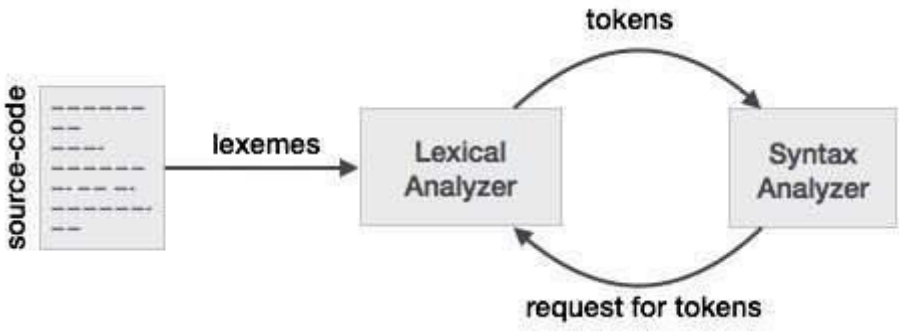
```
{white} {;}


"\\t"|"\\n"|"\\b"|"\\a" {

if(comment<=0)

printf("Escape character:\t\t\t\t%s\n", yytext);

}


{identi} {

if(comment<=0)

{

insert(1, yytext, "identifier");

printf("Identifier:\t\t\t\t%s\n", yytext);

}

}


"*"|"+"|"-"|"++"|"--"|"!"
|"~"|"<<"|">>"|"<="|"<"|">"|"=="|"!="|"&"|"^"|"|"|"&&"|"\||"|"*="|"+="|"-
="|"="|"&="|"^="|","|">>="|"<<="|"="|"%"|"=" {

if(comment<=0)

printf("Operator is found:\t\t\t\t%s\n", yytext);

}


"."|"'"|"\"|"#"|"("|")"|"["|"]"|"$"|";"|":"|"{"|"}" {

if(comment<=0)

printf("Special character:\t\t\t\t%s\n", yytext);

}


{sign}?{digit}*\.{digit}+ {

if(comment<=0)

{
```

```
insert(0, yytext, "float");
printf("Floating point constant:\t\t\t\t%s\n", yytext);
}
}

{sign}?{digit}+ {
if(comment<=0)
{
insert(0, yytext, "integer");
printf("Integer constant:\t\t\t\t%s\n", yytext);
}
}

\"(\\.|[^"\\])*\" {
if(comment<=0)
printf("String literal:\t\t\t\t%s\n", yytext);
}

{identi}\(({alphanum}|{white}|\,)*\) {
if(comment<=0)
printf("Function:\t\t\t\t%s\n", yytext);
}

{identi}(\[{digit}+\])+ {
if(comment<=0)
printf("Array:\t\t\t\t%s\n", yytext);
}

\"(\\.|[^"\\])* {
if(comment<=0)
```

```
printf(“String literal is not ending until EOF\n”);
}
{invalid_identifier} {
if(comment<=0)
printf("Invalid identifier name %s at line number %d\n", yytext, line);
}

%%
int main()
{
FILE *file;
file=fopen("abc.txt", "r");
yyin=file;
yylex();
if(comment!=0)
printf("Multiline comment error found\n");
printf("\n___________CONSTANTS TABLE___________\n");
printf("\nName\t\t\tType\t\t\tLine\n");
for(k=0; k<j; k++)
printf("%s\t\t\t%s\t\t\t%d\n",     constant[k].values,     constant[k].type,
constant[k].line_number);
printf("\n");
printf("\n___________SYMBOLS TABLE_____________\n");
printf("\nName\t\t\tType\n");
for(k=0; k<i; k++)
printf("%s\t\t\t%s\n", ident[k].values, ident[k].type);
printf("\n");
}
int yywrap()
{ return(1); }
```

5. EXPLANATION

C program is a normal procedure oriented language which contains keywords, comments (single line and multiline), functions, special characters etc.
Lex program mainly contains regular expressions that analyses the C code, identifies lexemes and categorizes them into tokens. 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.



Lexical Error Detection:

- When the scanner comes across a double quotation mark (“) in the input file and doesn’t find a double quotation mark till the end of the file, then lexical analyzer detects an error and displays the message “String literal does not end until the end of file”.
- When the scanner comes across a multiline comment starting symbol (/*) in the input file and doesn’t find a multiline comment ending symbol (*/) till the end of the file, then lexical analyzer detects an error and displays the message “Multiline comment error found”.
- When the scanner comes across an invalid token, such as 09b, which is an invalid identifier name, then suitable error message is displayed. A regular expression is used for this.

S. no	Regular Expression	Token
1	{sign}?{digit}+	Integer constants
2	{sign}?{digit}*\.{digit}+	Floating constants
3	\"(\\.|[^"\\])*\"	String literal
4	"//".*	Detects single-line comments
5	{identi}\(({alphanum}|{white}|\,)*\)	Detects function
6	{identi}(\[{digit}+\])+	Detects array

6. TEST CASES

Test Case #1

Input

```
int i;
// Working of for loop
for (i=0; i<5; i++)
{
    if(i%2==0)
      {
          i++;
          continue;
      }
    else
      {
         i+=2;
         break;
      }
}
```

Output

```
Keyword:                        int
Identifier:                     i
Special character:                      ;
Single line comment
Keyword:                        for
Special character:                      (
Identifier:                     i
Operator is found:                      =
Integer constant:                       0
Special character:                      ;
Identifier:                     i
Operator is found:                      <
Integer constant:                       5
Special character:                      ;
Identifier:                     i
Operator is found:                      ++
Special character:                      )
Special character:                      {
Keyword:                        if
Special character:                      (
Identifier:                     i
Operator is found:                      %
Integer constant:                       2
Operator is found:                      ==
Integer constant:                       0
Special character:                      )
Special character:                      {
Identifier:                     i
Operator is found:                      ++
Special character:                      ;
Keyword:                        continue
Special character:                      ;
Special character:                      }
Keyword:                        else
Special character:                      {
Identifier:                     i
Operator is found:                      +=
Integer constant:                       2
Special character:                      ;
Keyword:                        break
Special character:                      ;
Special character:                      }
Special character:                      }

__________CONSTANTS TABLE__________

Name                    Type                    Line
0                       integer                 3
5                       integer                 3
2                       integer                 5


__________SYMBOLS TABLE__________

Name                    Type
int                     keyword
i                       identifier
for                     keyword
if                      keyword
continue                        keyword
else                    keyword
break                   keyword
```

Test- Case #2

Input

```
#define pi 3.14
int r;
/* r is the radius of the circle. This program computes
 the circumference of a circle.*/
printf("Enter the radius of the circle");
scanf("%d",&r);
printf("The circumference of the circle is %f", 2*pi*r);
printf("Program has ended);
```

Output

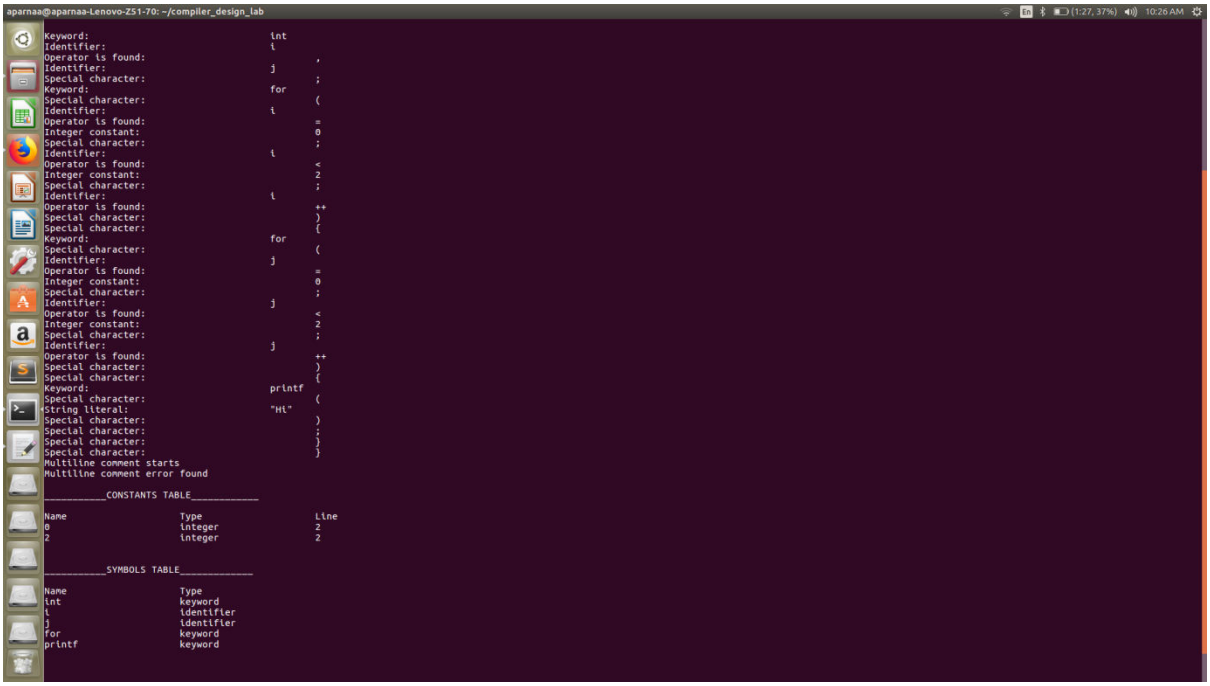
```
Special character:                      #
Identifier:                             define
Identifier:                             pi
Floating point constant:                        3.14
Keyword:                                int
Identifier:                             r
Special character:                      ;
Multiline comment starts
Closing of multiline comment
Keyword:                                printf
Special character:                      (
String literal:                         "Enter the radius of the circle"
Special character:                      )
Special character:                      ;
Keyword:                                scanf
Special character:                      (
String literal:                         "%d"
Operator is found:                      ,
Operator is found:                      &
Identifier:                             r
Special character:                      )
Special character:                      ;
Keyword:                                printf
Special character:                      (
String literal:                         "The circumference of the circle is %f"
Operator is found:                      ,
Integer constant:                       2
Operator is found:                      *
Identifier:                             pi
Operator is found:                      *
Identifier:                             r
Special character:                      )
Special character:                      ;
Keyword:                                printf
Special character:                      (
String literal is not ending until EOF

__________CONSTANTS TABLE__________

Name                    Type                    Line
3.14                    float                   1
2                       integer                 7


__________SYMBOLS TABLE__________

Name                    Type
define                  identifier
pi                      identifier
int                     keyword
r                       identifier
printf                  keyword
scanf                   keyword
```

Test- Case #3

Input

```
int i,j;
for(i=0;i<2;i++)
 {
   for(j=0;j<2;j++)
     {
        printf("Hi");
     }
 }
/*This program uses nested loop to print
Hi four times./
```

Output



Test-Case #4

Input

```
/*/*This is a nested comment*/*/
int find_double(int a)
{
  int b= 2*a;
  return b;
}
9ab;
```

Output

```
Multiline comment starts
Multiline comment starts
Closing of multiline comment
Closing of multiline comment
Keyword:                        int
Function:                       find_double(int a)
Special character:                      {
Keyword:                        int
Identifier:                     b
Operator is found:                      =
Integer constant:                       2
Operator is found:                      *
Identifier:                     a
Special character:                      ;
Keyword:                        return
Identifier:                     b
Special character:                      ;
Special character:                      }
Invalid identifier name 9ab at line number 7
Special character:                      ;

__________CONSTANTS TABLE___________

Name                    Type                    Line
2                       integer                 4


__________SYMBOLS TABLE_____________

Name                    Type
int                     keyword
b                       identifier
a                       identifier
return                  keyword
```

Test-Case #5

Input

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main()
{
int a=-1;
float b=-3.26;

a++;   // // a++ increments value of a by 1

b=b*b;
}
```

Output

```
Preprocessor directive:                 #include<stdio.h>
Preprocessor directive:                 #include<stdlib.h>
Preprocessor directive:                 #include<math.h>
Keyword:                        int
Function:                       main()
Special character:                      {
Keyword:                        int
Identifier:                     a
Operator is found:                      =
Integer constant:                       -1
Special character:                      ;
Keyword:                        float
Identifier:                     b
Operator is found:                      =
Floating point constant:                        -3.26
Special character:                      ;
Identifier:                     a
Operator is found:                      ++
Special character:                      ;
Single line comment
Identifier:                     b
Operator is found:                      =
Identifier:                     b
Operator is found:                      *
Identifier:                     b
Special character:                      ;
Special character:                      }

__________CONSTANTS TABLE__________

Name                    Type                    Line
-1                      integer                 7
-3.26                   float                   8


__________SYMBOLS TABLE__________

Name                    Type
int                     keyword
a                       identifier
float                   keyword
b                       identifier
```

Test-Case #6

Input

```
#include<stdio.h>
#include<abc.h>

int main()
{
int arr[1];
int array[2][2];
int sun]3[;
int find_prod(3,6);
int find_sum)3,6(;
}
```

Output

```
Preprocessor directive:                         #include<stdio.h>
Preprocessor directive:                         #include<abc.h>
Keyword:                                int
Function:                               main()
Special character:                              {
Keyword:                                int
Array:                          arr[1]
Special character:                              ;
Keyword:                                int
Array:                          array[2][2]
Special character:                              ;
Keyword:                                int
Identifier:                             sun
Special character:                              ]
Integer constant:                               3
Special character:                              [
Special character:                              ;
Keyword:                                int
Function:                               find_prod(3,6)
Special character:                              ;
Keyword:                                int
Identifier:                             find_sum
Special character:                              )
Integer constant:                               3
Operator is found:                              ,
Integer constant:                               6
Special character:                              (
Special character:                              ;
Special character:                              }

__________CONSTANTS TABLE__________

Name                    Type                    Line
3                       integer                 8
6                       integer                 10


__________SYMBOLS TABLE__________

Name                    Type
int                     keyword
sun                     identifier
find_sum                        identifier
```

7. RESULTS/FUTURE WORK

The result of the project will be in the form of a flex file which acts as scanner for C language. The lexical analyser reads the source program as a sequence of characters and recognizes tokens as mentioned earlier in the introduction section. The future work of our project is to develop a fully robust scanner with a better level of functionality and better error detection capacity as compared to the present one. We plan to develop a parser using YACC, a semantic checker and an intermediate code generator for C language and make sure that all the components of the project work with each other to develop a good compiler. Our future work will aim to identify and correct syntax and semantic errors with appropriate suggestions to correct errors using flex and YACC.

8. REFERENCES

http://alumni.cs.ucr.edu/~lgao/teaching/ [2]http://www.codeproject.com/Articles/ [3]
https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html
http://www.isi.edu/~pedro/Teaching/CSCI565-Fall15/Materials/LexAndYaccTutorial.pdf
http://www.tldp.org/HOWTO/Lex-YACC-HOWTO-3.html