LEXICAL ANALYSER

A MINI PROJECT REPORT

Submitted by

AABHAS DOGRA [RA2111042010022]
DHAWAL SAHU [RA2111042010015]
ASTHA JAISWAL [RA2111042010025]

for the course 18CSC362J –Compiler Design

Under the guidance of

Dr. S. Sharanya

(Associate Professor, Department of Data Science and Business Systems)

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

DATA SCIENCE AND BUSINESS SYSTEM

of

FACULTY OF ENGINEERING AND TECHNOLOGY

S.R.M. Nagar, Kattankulathur, Chengalpattu District

OCTOBER 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur – 603203

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF DATA SCIENCE AND BUSINESS SYSTEMS

BONAFIDE CERTIFICATE

Reg. No:

R	A	2	1	1	1	0	4	2	0	1	0	0	1	5

Certified that this is the bonafide record of work done by **DHAWAL SAHU** of V semester B.Tech. DATA SCIENCE WITH BUSINESS SYSTEM during the academic year 2023-2024 in the 18CSC362J – Compiler Design Laboratory.

------------------------------ ------------------------------------

Dr. S. Sharanya **Dr. M. Lakshmi**

Faculty - Incharge **HOD/DSBS**

Submitted for the practical examination held on_____________ at SRM Institute of Science and Technology, Kattankulathur, Chennai-603203.

------------------------------ ------------------------------

Examiner-1 **Examiner-2**

ACKNOWLEDGEMENT

We express our humble gratitude to Dr. C. Muthamizhchelvan, Vice Chancellor (I/C), SRM Instituteof Science and Technology, for the facilities extended for the project work and his continued support.We extend our sincere thanks to Dr. Revathi Venkataraman, Professor & Chairperson, School of Computing, SRM Institute of Science and Technology, for her invaluable support.

We wish to thank Dr. M. Lakshmi, Professor & Head, Department of Data Science and Business System, SRM Institute of Science and Technology, for her valuable suggestions and encouragementthroughout the period of the project work.

We are extremely grateful to our Academic Advisor Dr E. Sasikala, Professor, Departmentof Data Science and Business Systems, SRM Institute of Science and Technology, for her great support at all the stages of project work.

We register our immeasurable thanks to our Faculty Advisors, Dr. Jeba Sonia, and Dr. Mercy Thomas,Assistant Professor, Department of Data Science and Business System, SRM Institute of Science andTechnology, for leading and helping us to complete our course.

Our inexpressible respect and thanks to our guide, Dr. S. Sharanya, Assistant Professor, Departmentof Data Science and Business System, SRM Institute of Science and Technology, for providingus an opportunity to pursue our project under her mentorship. She provided the freedom and support to explore the research topics of our interest. Her passion for solving real problems and making a difference in the world has always been inspiring.

We sincerely thank staff and students of the Data Science and Business System Department, SRM Institute of Science and Technology, for their help during my research.

Finally, we would like to thank my parents, our family members and our friends for their unconditional love, constant support, and encouragement.

ABSTRACT

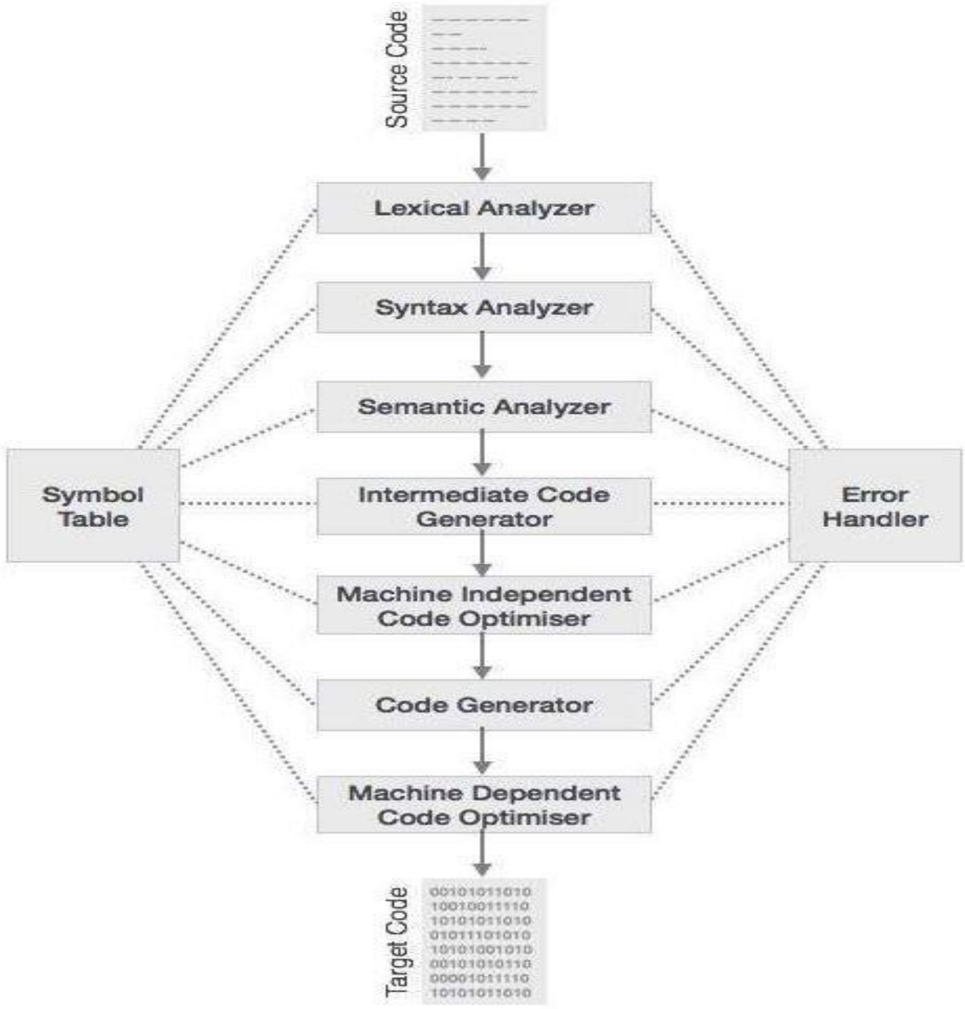
A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or code that a computer's processors use. The file used for writing a C-language contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. The output of the compilation is called object code or sometimes an object module. Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. We have designed a lexical analyzer for the C language using lex. It takes as input a C code and outputs a stream of tokens. The tokens displayed as part of the output include keywords, identifiers, signed/unsigned integer/floating point constants, operators, special characters, headers, data-type specifiers, array, single-line comment, multi-line comment, preprocessor directive, pre-defined functions (printf and scanf), user-defined functions and the main function. The token, the type of token and the line number of the token in the C code are being displayed. The line number is displayed so that it is easier to debug the code for errors. Errors in single-line comments, multi-line comments are displayed along with line numbers. The output also contains the symbol table which contains tokens and their type. The symbol table is generated using the hash organisation.

CONTENTS

COMPILER DESIGN PHASES

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 processors use. A compiler can broadly be divided into two phases based on the way they compile.

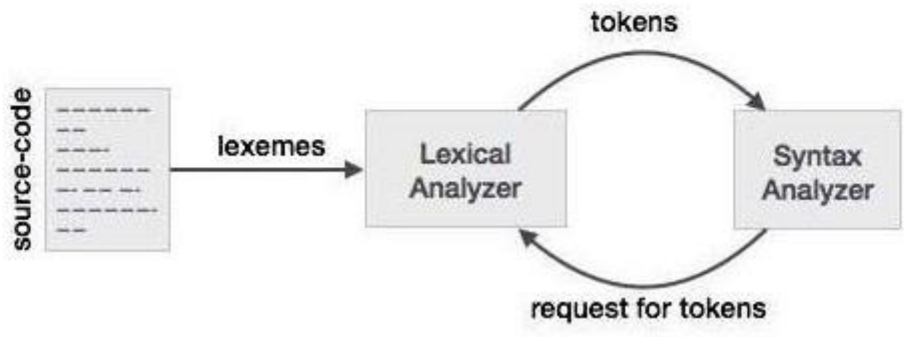
1. **Analysis phase:** Known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table. This phase consists of:
   - ➢ Lexical Analysis
   - ➢ Syntax Analysis
   - ➢ Semantic Analysis
   - ➢ Intermediate Code Generation
2. **Synthesis phase**: Known as the back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table. This phase consists of:
   - ➢ Code Optimization
   - ➢ Code Generator



Lexical Analysis

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

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



Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).

Semantic Analysis

Semantic analysis is the third phase of a compiler. Semantic analyzer checks whether the parse tree constructed by the syntax analyzer follows the rules of language.

Intermediate Code generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

In this phase, code optimization of the intermediate code is done. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.

THE LEXICAL ANALYSIS

In computer science, lexical analysis is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an identified "meaning"). A program that performs lexical analysis may be called a lexer, tokenizer, or scanner (though "scanner" is also used to refer to the first stage of a lexer). Such a lexer is generally combined with a parser, which together analyze the syntax of programming languages, web pages, and so forth.

The script written by us is a computer program called the “lex” program, is the one that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

The structure of the lex program consists of three sections:

{definition section}

%%

{rules section}

%%

{C code section}

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

The lex program, when compiled using the lex command, generates a file called lex.yy.c, which when executed recognizes the tokens present in the input C program.

Lexical analysis only takes care of parsing the tokens and identifying their type. The output of this phase is the stream of tokens as well as the symbol table representing the tokens and their type.

THE LEX PROGRAM: (lexer.l)

```
%{
int lineno = 1;
#include<stdio.>
#include<stdlib.>
#include<string>
#define AUTO 1
#define BREAK 2
#define CASE 3
#define CHAR 4
#define CONST 5
#define CONTINUE 6
#define DEFAULT 7
#define DO 8
#define DOUBLE 9
#define ELSE 10
#define ENUM 11
#define EXTERN 12
#define FLOAT 13
#define FOR 14
#define GOTO 15
#define IF 16
#define INT 17
#define LONG 18
#define REGISTER 19
#define RETURN 20
#define SHORT 21
#define SIGNED 22
#define SIZEOF 23
#define STATIC 24
#define STRUCT 25
#define SWITCH 26
#define TYPEDEF 27
#define UNION 28
```

```
#define UNSIGNED 29
#define VOID 30
#define VOLATILE 31
#define WHILE 32
#define IDENTIFIER 33
#define SLC 34
#define MLCS 35
#define MLCE 36
#define LEQ 37
#define GEQ 38
#define EQEQ 39
#define NEQ 40
#define LOR 41
#define LAND 42
#define ASSIGN 43
#define PLUS 44
#define SUB 45
#define MULT 46
#define DIV 47
#define MOD 48
#define LESSER 49
#define GREATER 50
#define INCR 51
#define DECR 52
#define COMMA 53
#define SEMI 54
#define HEADER 55
#define MAIN 56
#define PRINTF 57
#define SCANF 58
#define DEFINE 59
#define INT_CONST 60
#define FLOAT_CONST 61
```

```
#define TYPE_SPEC 62
#define DQ 63
#define OBO 64
#define OBC 65
#define CBO 66
#define CBC 67
#define HASH 68
#define ARR 69
#define FUNC 70
#define NUM_ERR 71
#define UNKNOWN 72
#define CHAR_CONST 73
#define SIGNED_CONST 74
#define STRING_CONST 75%}
alpha [A-Za-z]
digit [0-9]
und [_]
space [ ]
tab [ ]
line [\n]
char \'.\'
at [@]

string \"(.^([%d]|[%f]|[%s]|[%c]))\"

%%
{space}* {}
{tab}* {}

{string} return STRING_CONST;
{char} return CHAR_CONST;
{line} {lineno++;} auto
return AUTO;
break return BREAK;
case return CASE;
char return CHAR;
const return CONST;
continue return CONTINUE;
```

```
default return DEFAULT;
do return DO;
double return DOUBLE;
else return ELSE;
enum return ENUM;
extern return EXTERN;
float return FLOAT;
for return FOR;
goto return GOTO;
if return IF;
int return INT;
long return LONG;
register return REGISTER;
return return RETURN;
short return SHORT;
signed return SIGNED;
sizeof return SIZEOF;
static return STATIC;
struct return STRUCT;
switch return SWITCH;
typedef return TYPEDEF;
union return UNION;
unsigned return UNSIGNED;
void return VOID;
volatile return VOLATILE;
while return WHILE;
printf return PRINTF;
scanf return SCANF;
{alpha}({alpha}|{digit}|{und})* return IDENTIFIER;
[+-][0-9]{digit}*(\.{digit}+)? return SIGNED_CONST;
"//" return SLC;
"/*" return MLCS;
"*/" return MLCE;
"<=" return LEQ;
">=" return GEQ;
"==" return EQEQ;
"!=" return NEQ;
"||" return LOR;
"&&" return LAND;
"=" return ASSIGN;
"+" return PLUS;
"-" return SUB;
"*" return MULT;
"/" return DIV;
"%" return MOD;
"<" return LESSER;
">" return GREATER;
"++" return INCR;
"--" return DECR;
"," return COMMA;
";" return SEMI;
"#include<stdio.h>" return HEADER;
"#include <stdio.h>" return HEADER;
"main()" return MAIN;
{digit}+ return INT_CONST;
```

```
({digit}+)\.({digit}+) return FLOAT_CONST;
"%d"|"%f"|"%u"|"%s" return TYPE_SPEC;
"\"" return DQ;
"(" return OBO;
")" return OBC;
"{" return CBO;
"}" return CBC;
"#" return HASH;
{alpha}({alpha}|{digit}|{und})*\[{digit}*\] return ARR;
{alpha}({alpha}|{digit}|{und})*\(({alpha}|{digit}|{und}|{space})*\) return FUNC;
({digit}+)\.({digit}+)\.({digit}|\.)* return NUM_ERR;
({digit}|{at})+({alpha}|{digit}|{und}|{at})* return UNKNOWN;
%%
struct node
{
char token[100];
char attr[100];
struct node *next;
};
struct hash
{
struct node *head;
int count;
};
struct hash hashTable[1000];
int eleCount = 1000;

struct node * createNode(char *token, char *attr)
{
struct node *newnode;
newnode = (struct node *) malloc(sizeof(struct node));
strcpy(newnode->token, token);
strcpy(newnode->attr, attr);
newnode->next = NULL;
return newnode;
}
int hashIndex(char *token)
{
int hi=0;
int l,i;
for(i=0;token[i]!='\0';i++)
{
hi = hi + (int)token[i];
}
hi = hi%eleCount;
return hi;
}
void insertToHash(char *token, char *attr)
{
int flag=0;
int hi;
hi = hashIndex(token);
struct node *newnode = createNode(token, attr);
/* head of list for the bucket with index "hashIndex" */
if (hashTable[hi].head==NULL)
```

```
{
hashTable[hi].head = newnode;
hashTable[hi].count = 1;
return;
}
struct node *myNode;
myNode = hashTable[hi].head;
while (myNode != NULL)
{
if (strcmp(myNode->token, token)==0)
{
flag = 1;
break;
}
myNode = myNode->next;
}
if(!flag)
{
//adding new node to the list
newnode->next = (hashTable[hi].head);
//update the head of the list and no of nodes in the current bucket
hashTable[hi].head = newnode;
hashTable[hi].count++;
}
return;
}
void display()
{
struct node *myNode;
int i,j, k=1;
printf("-------------------------------------------------------------------");
printf("\nSNo \t|\tToken \t\t|\tToken Type \t\n");
printf("-------------------------------------------------------------------\n");
for (i = 0; i < eleCount; i++)
{
if (hashTable[i].count == 0)
continue;
myNode = hashTable[i].head;
if (!myNode)
continue;
while (myNode != NULL)
{
printf("%d\t\t", k++);
printf("%s\t\t\t", myNode->token);
printf("%s\t\n", myNode->attr);
myNode = myNode->next;
}
}
return;
}
int main()
{
int scan, slcline=0, mlc=0, mlcline=0, dq=0, dqline=0;
yyin = fopen("isPrime.c","r");
printf("\n\n");
```

```
scan = yylex();
while(scan)
{
if(lineno == slcline)
{
scan = yylex();
continue;
}
if(lineno!=dqline && dqline!=0)
{
if(dq%2!=0)
printf("\n******** ERROR!! INCOMPLETE STRING at Line %d
********\n\n", dqline);
dq=0;
}
if((scan>=1 && scan<=32) && mlc==0)
{
printf("%s\t\t\tKEYWORD\t\t\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "KEYWORD");
}
if(scan==33 && mlc==0)
{
printf("%s\t\t\tIDENTIFIER\t\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "IDENTIFIER");
}
if(scan==34)
{
printf("%s\t\t\tSingleline Comment\t\tLine %d\n", yytext, lineno);
slcline = lineno;
}
if(scan==35 && mlc==0)
{
printf("%s\t\t\tMultiline Comment Start\t\tLine %d\n", yytext, lineno);
mlcline = lineno;
mlc = 1;
}
if(scan==36 && mlc==0)
{
printf("\n******** ERROR!! UNMATCHED MULTILINE COMMENT END %s at
Line %d ********\n\n", yytext, lineno);
}
if(scan==36 && mlc==1)
{
mlc = 0;
printf("%s\t\t\tMultiline Comment End\t\tLine %d\n", yytext, lineno);
}
if((scan>=37 && scan<=52) && mlc==0)
{
printf("%s\t\t\tOPERATOR\t\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "OPERATOR");
}
if((scan==53||scan==54||scan==63||(scan>=64 && scan<=68)) && mlc==0)
{
printf("%s\t\t\tSPECIAL SYMBOL\t\t\tLine %d\n", yytext, lineno);
if(scan==63)
```

```
{
dq++;
dqline = lineno;
}
insertToHash(yytext, "SPECIAL SYMBOL");
}
if(scan==55 && mlc==0)
{
printf("%s\tHEADER\t\t\t\tLine %d\n",yytext, lineno);
}
if(scan==56 && mlc==0)
{
printf("%s\t\t\tMAIN FUNCTION\t\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "IDENTIFIER");
}
if((scan==57 || scan==58) && mlc==0)
{
printf("%s\t\t\tPRE DEFINED FUNCTION\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "PRE DEFINED FUNCTION");
}
if(scan==59 && mlc==0)
{
printf("%s\t\t\tPRE PROCESSOR DIRECTIVE\t\tLine %d\n", yytext, lineno);
}
if(scan==60 && mlc==0)
{
printf("%s\t\t\tINTEGER CONSTANT\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "INTEGER CONSTANT");
}
if(scan==61 && mlc==0)
{
printf("%s\t\t\tFLOATING POINT CONSTANT\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "FLOATING POINT CONSTANT");
}
if(scan==62 && mlc==0)
{
printf("%s\t\t\tTYPE SPECIFIER\t\t\tLine %d\n", yytext, lineno);
}
if(scan==69 && mlc==0)
{
printf("%s\t\t\tARRAY\t\t\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "ARRAY");
}
if(scan==70 && mlc==0)
{
printf("%s\t\t\tUSER DEFINED FUNCTION\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "USER DEFINED FUNCTION");
}
if(scan==71 && mlc==0)
{
printf("\n******** ERROR!! CONSTANT ERROR %s at Line %d
********\n\n", yytext, lineno);
}
if(scan==72 && mlc==0)
{
```

```
printf("\n******** ERROR!! UNKNOWN TOKEN %s at Line %d
********\n\n", yytext, lineno);
}
if(scan==73 && mlc==0)
{
printf("%s\t\t\tCHARACTER CONSTANT\t\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "CHARACTER CONSTANT");
}
if(scan==74 && mlc==0)
{
printf("%s\t\t\tSIGNED CONSTANT\t\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "SIGNED CONSTANT");
}
if(scan==75 && mlc==0)
{
printf("%s\t\t\tSTRING CONSTANT\t\t\tLine %d\n", yytext, lineno);
insertToHash(yytext, "STRING CONSTANT");
}
scan = yylex();
}
if(mlc==1)
printf("\n******** ERROR!! UNMATCHED COMMENT STARTING at Line %d
********\n\n",mlcline);
printf("\n");
printf("\n\t******** SYMBOL TABLE ********\t\t\n");
display();
printf("-------------------------------------------------------------------\n\n");
}
int yywrap()
{
return 1;
}
```

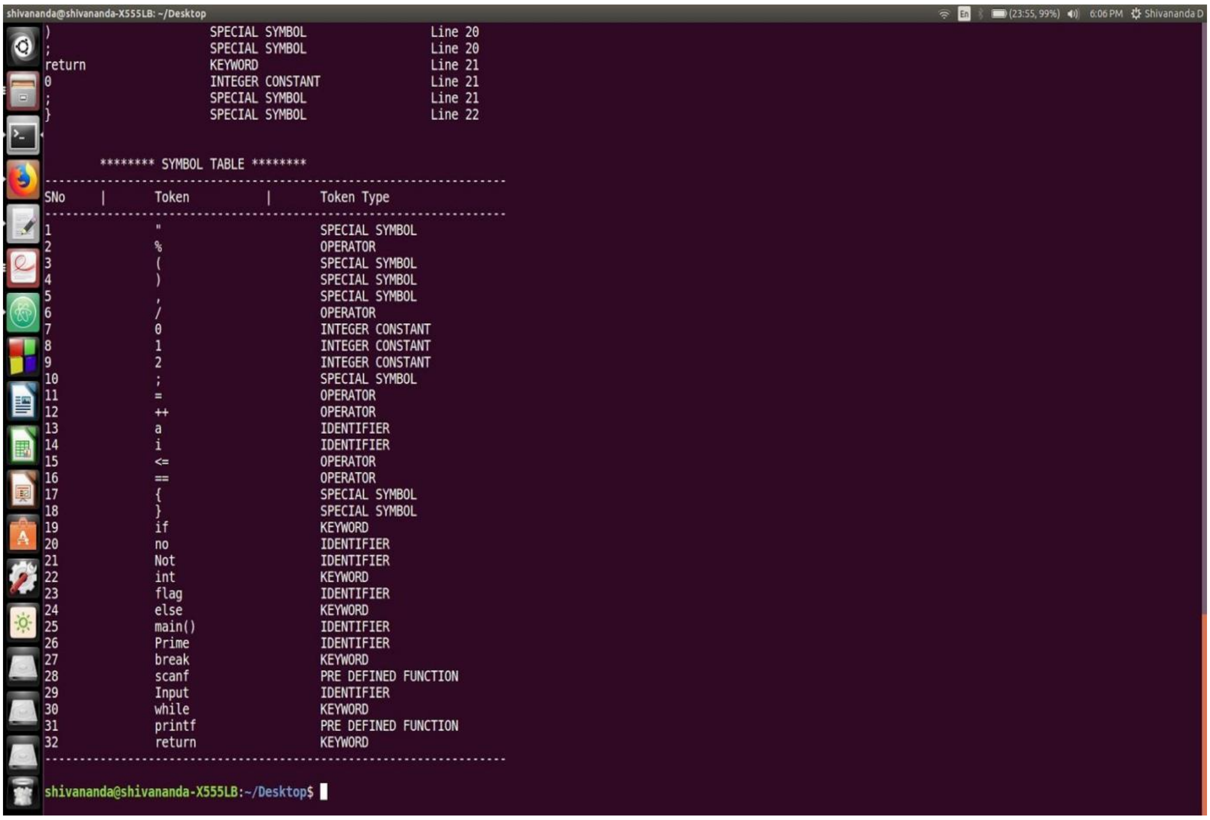
Input file (isPrime.c)

```
#include<stdio.h>
int main()
{
int a,i,flag=0;
printf("Input no");
scanf("%d",&a);
i=2;
while(i <= a/2)
{
if(a%i == 0)
{
flag=1;
break;
}
i++;
}
if(flag==0)
printf("%d Prime", a);
else
printf("Not Prime");
return 0;
}
```

Output:

```
shivananda@shivananda-X555LB:~/Desktop$ lex lexer.l
shivananda@shivananda-X555LB:~/Desktop$ cc lex.yy.c
shivananda@shivananda-X555LB:~/Desktop$ ./a.out


#include<stdio.h>       HEADER                  Line 1
int                     KEYWORD                 Line 2
main()                  MAIN FUNCTION           Line 2
{                       SPECIAL SYMBOL          Line 3
int                     KEYWORD                 Line 4
a                       IDENTIFIER              Line 4
,                       SPECIAL SYMBOL          Line 4
i                       IDENTIFIER              Line 4
,                       SPECIAL SYMBOL          Line 4
flag                    IDENTIFIER              Line 4
=                       OPERATOR                Line 4
0                       INTEGER CONSTANT        Line 4
;                       SPECIAL SYMBOL          Line 4
printf                  PRE DEFINED FUNCTION    Line 5
(                       SPECIAL SYMBOL          Line 5
"                       SPECIAL SYMBOL          Line 5
Input                   IDENTIFIER              Line 5
no                      IDENTIFIER              Line 5
"                       SPECIAL SYMBOL          Line 5
)                       SPECIAL SYMBOL          Line 5
;                       SPECIAL SYMBOL          Line 5
scanf                   PRE DEFINED FUNCTION    Line 6
(                       SPECIAL SYMBOL          Line 6
"                       SPECIAL SYMBOL          Line 6
%d                      TYPE SPECIFIER          Line 6
"                       SPECIAL SYMBOL          Line 6
,                       SPECIAL SYMBOL          Line 6
&a                      IDENTIFIER              Line 6
)                       SPECIAL SYMBOL          Line 6
;                       SPECIAL SYMBOL          Line 6
i                       IDENTIFIER              Line 7
=                       OPERATOR                Line 7
2                       INTEGER CONSTANT        Line 7
;                       SPECIAL SYMBOL          Line 7
while                   KEYWORD                 Line 8
(                       SPECIAL SYMBOL          Line 8
i                       IDENTIFIER              Line 8
<=                      OPERATOR                Line 8
a                       IDENTIFIER              Line 8
/                       OPERATOR                Line 8
2                       INTEGER CONSTANT        Line 8
)                       SPECIAL SYMBOL          Line 8
```

```
{                       SPECIAL SYMBOL          Line 9
if                      KEYWORD                 Line 10
(                       SPECIAL SYMBOL          Line 10
a                       IDENTIFIER              Line 10
%                       OPERATOR                Line 10
i                       IDENTIFIER              Line 10
==                      OPERATOR                Line 10
0                       INTEGER CONSTANT        Line 10
)                       SPECIAL SYMBOL          Line 10
{                       SPECIAL SYMBOL          Line 11
flag                    IDENTIFIER              Line 12
=                       OPERATOR                Line 12
1                       INTEGER CONSTANT        Line 12
;                       SPECIAL SYMBOL          Line 12
break                   KEYWORD                 Line 13
;                       SPECIAL SYMBOL          Line 13
}                       SPECIAL SYMBOL          Line 14
i                       IDENTIFIER              Line 15
++                      OPERATOR                Line 15
;                       SPECIAL SYMBOL          Line 15
}                       SPECIAL SYMBOL          Line 16
if                      KEYWORD                 Line 17
(                       SPECIAL SYMBOL          Line 17
flag                    IDENTIFIER              Line 17
==                      OPERATOR                Line 17
0                       INTEGER CONSTANT        Line 17
)                       SPECIAL SYMBOL          Line 17
printf                  PRE DEFINED FUNCTION    Line 18
(                       SPECIAL SYMBOL          Line 18
"                       SPECIAL SYMBOL          Line 18
%d                      TYPE SPECIFIER          Line 18
Prime                   IDENTIFIER              Line 18
"                       SPECIAL SYMBOL          Line 18
,                       SPECIAL SYMBOL          Line 18
a                       IDENTIFIER              Line 18
)                       SPECIAL SYMBOL          Line 18
;                       SPECIAL SYMBOL          Line 18
else                    KEYWORD                 Line 19
printf                  PRE DEFINED FUNCTION    Line 20
(                       SPECIAL SYMBOL          Line 20
"                       SPECIAL SYMBOL          Line 20
Not                     IDENTIFIER              Line 20
Prime                   IDENTIFIER              Line 20
"                       SPECIAL SYMBOL          Line 20
)                       SPECIAL SYMBOL          Line 20
;                       SPECIAL SYMBOL          Line 20
return                  KEYWORD                 Line 21
```



```
shivananda@shivananda-X555LB: ~/Desktop
)                       SPECIAL SYMBOL                  Line 20
;                       SPECIAL SYMBOL                  Line 20
return                  KEYWORD                         Line 21
0                       INTEGER CONSTANT                Line 21
;                       SPECIAL SYMBOL                  Line 21
}                       SPECIAL SYMBOL                  Line 22


        ******** SYMBOL TABLE ********
-------------------------------------------------------------------
SNo     |       Token           |       Token Type
-------------------------------------------------------------------
1               "                       SPECIAL SYMBOL
2               %                       OPERATOR
3               (                       SPECIAL SYMBOL
4               )                       SPECIAL SYMBOL
5               ,                       SPECIAL SYMBOL
6               /                       OPERATOR
7               0                       INTEGER CONSTANT
8               1                       INTEGER CONSTANT
9               2                       INTEGER CONSTANT
10              ;                       SPECIAL SYMBOL
11              =                       OPERATOR
12              ++                      OPERATOR
13              a                       IDENTIFIER
14              i                       IDENTIFIER
15              <=                      OPERATOR
16              ==                      OPERATOR
17              {                       SPECIAL SYMBOL
18              }                       SPECIAL SYMBOL
19              if                      KEYWORD
20              no                      IDENTIFIER
21              Not                     IDENTIFIER
22              int                     KEYWORD
23              flag                    IDENTIFIER
24              else                    KEYWORD
25              main()                  IDENTIFIER
26              Prime                   IDENTIFIER
27              break                   KEYWORD
28              scanf                   PRE DEFINED FUNCTION
29              Input                   IDENTIFIER
30              while                   KEYWORD
31              printf                  PRE DEFINED FUNCTION
32              return                  KEYWORD
-------------------------------------------------------------------

shivananda@shivananda-X555LB:~/Desktop$
```

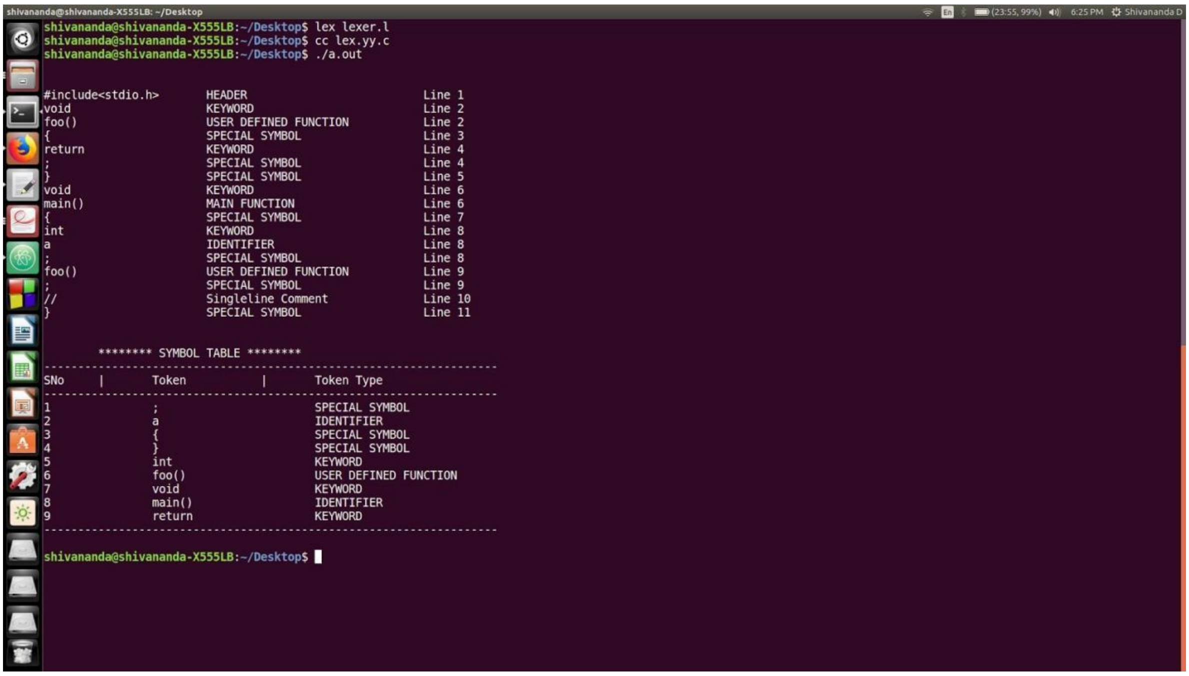
TEST CASES

Without errors

1. Input file (test5.c) :

```
#include<stdio.h>
void foo()
{
    return;
}
void main()
{
    int a;
    foo();
    //user defined function
}
```

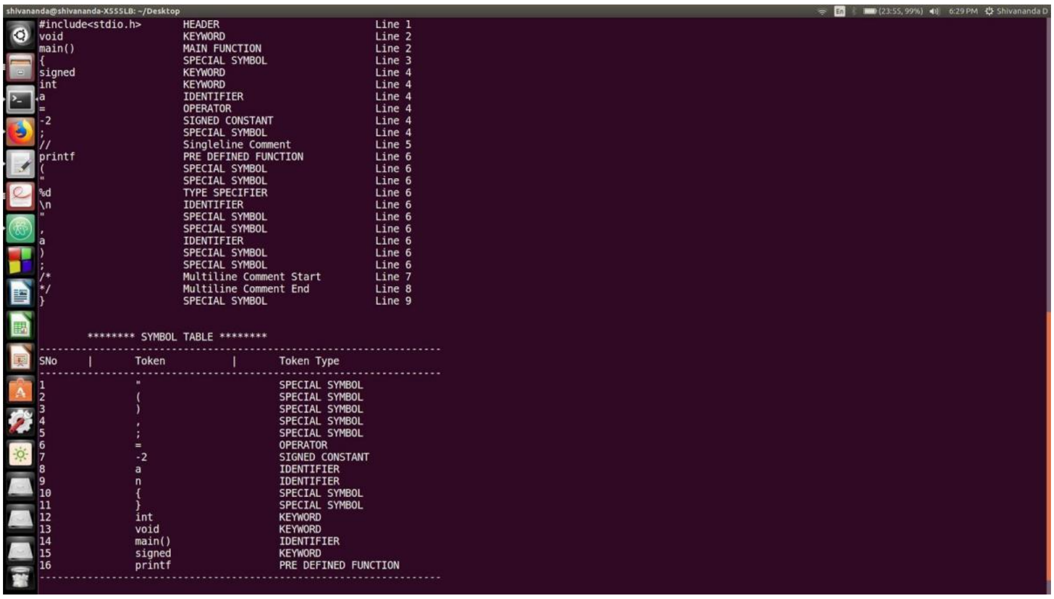
Output :



2.Input file (test2.c) :

```
#include<stdio.h> void
main()
 {
    signed int a = -2;
    //Single line comment
    a++;
    printf("%d\n", a);
    /* Multiline
    comment */
 }
```

Output :

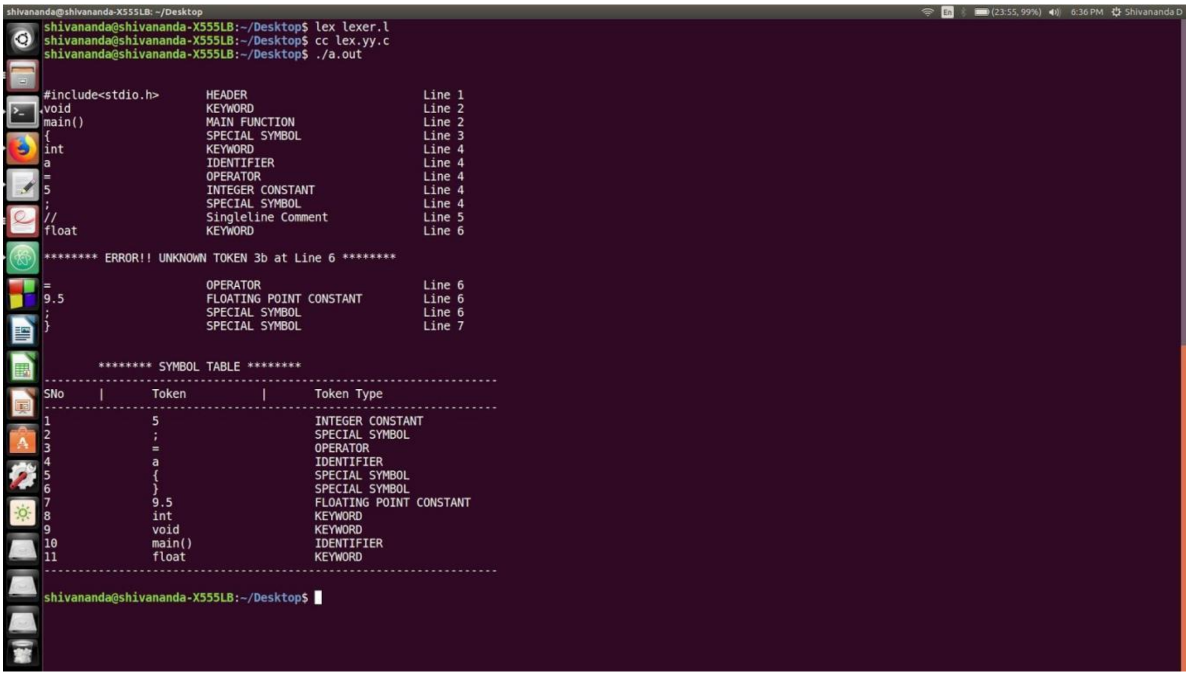


With errors

1. Input file (test3.c) :

```
#include<stdio.h>

void main()

{

   int a=5;

   //identifier rule broken

   float 3b = 9.5;

}
```

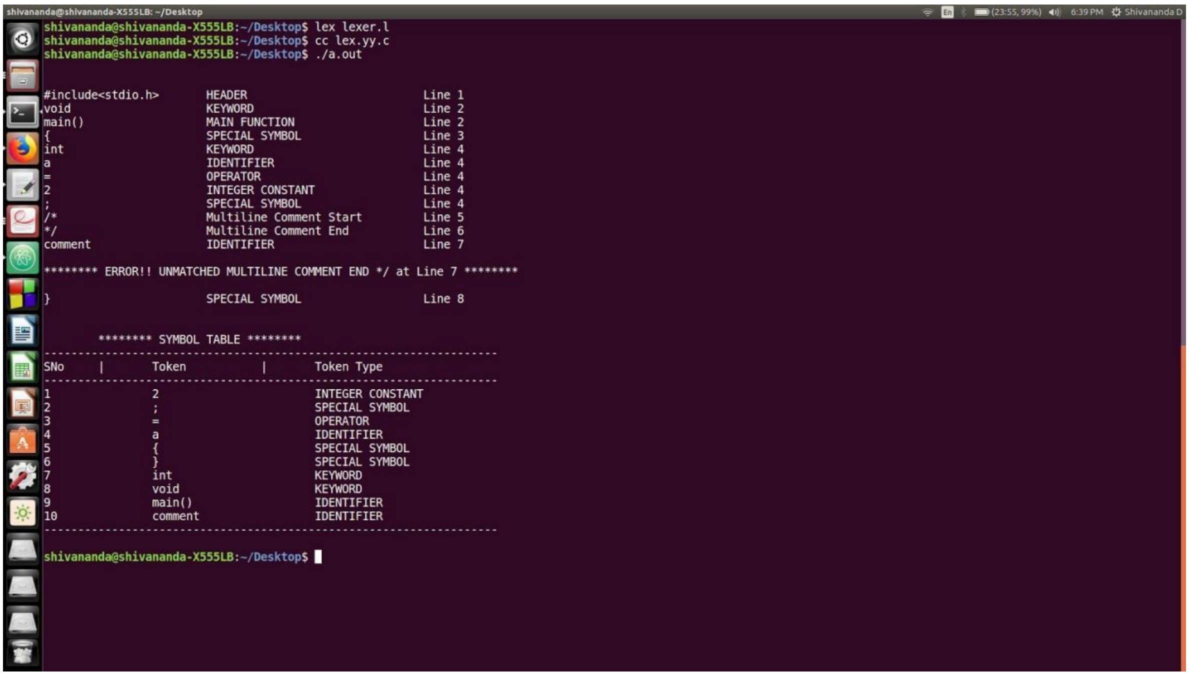
Output :



2. Input file (test4.c) :

```
#include<stdio.h>
void main()
{
   int a=2;
   /*Unmatched
   /*Multiline*/
   comment*/
}
```

Output :



3. Input file (test6.c) :

#include<stdio.h> void

main()

{

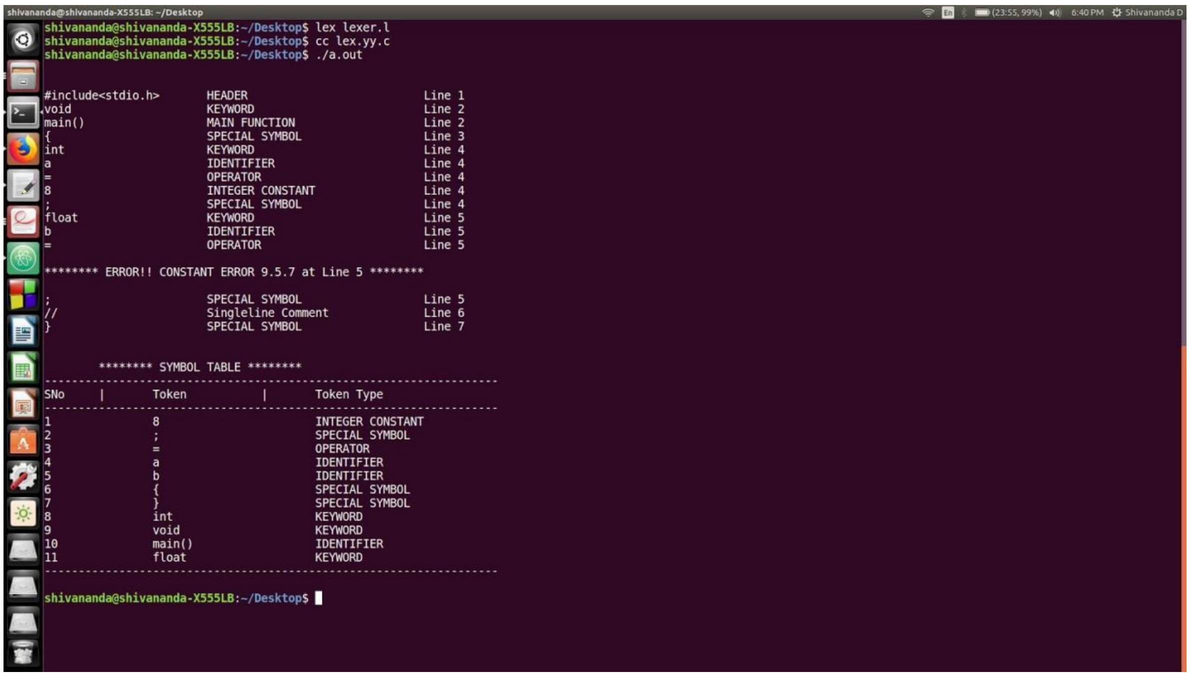
int a = 8;

float b = 9.5.7;

//constant error

}

Output :



```
shivananda@shivananda-X555LB:~/Desktop$ lex lexer.l
shivananda@shivananda-X555LB:~/Desktop$ cc lex.yy.c
shivananda@shivananda-X555LB:~/Desktop$ ./a.out


#include<stdio.h>       HEADER                          Line 1
void                    KEYWORD                         Line 2
main()                  MAIN FUNCTION                   Line 2
{                       SPECIAL SYMBOL                  Line 3
int                     KEYWORD                         Line 4
a                       IDENTIFIER                      Line 4
=                       OPERATOR                        Line 4
8                       INTEGER CONSTANT                Line 4
;                       SPECIAL SYMBOL                  Line 4
float                   KEYWORD                         Line 5
b                       IDENTIFIER                      Line 5
=                       OPERATOR                        Line 5

******** ERROR!! CONSTANT ERROR 9.5.7 at Line 5 ********

;                       SPECIAL SYMBOL                  Line 5
//                      Singleline Comment              Line 6
}                       SPECIAL SYMBOL                  Line 7


        ******** SYMBOL TABLE ********
-------------------------------------------------------------------
SNo     |       Token           |       Token Type
-------------------------------------------------------------------
1               8                       INTEGER CONSTANT
2               ;                       SPECIAL SYMBOL
3               =                       OPERATOR
4               a                       IDENTIFIER
5               b                       IDENTIFIER
6               {                       SPECIAL SYMBOL
7               }                       SPECIAL SYMBOL
8               int                     KEYWORD
9               void                    KEYWORD
10              main()                  IDENTIFIER
11              float                   KEYWORD
-------------------------------------------------------------------

shivananda@shivananda-X555LB:~/Desktop$
```

CONCLUSION

Here is a sample conclusion for a project report on lexical analysis:

In conclusion, this project successfully developed a lexical analyzer for a subset of the C programming language using Python. The key achievements include:

- Designing the lexical grammar by defining tokens and regular expressions based on language grammar rules.
- Implementing lexical analyzer logic using Python libraries to match input strings, extract lexemes, and identify tokens.
- Building a SymbolTable to hold identifiers and assign semantic information.
- Handling whitespace, comments, operators, delimiters, identifiers, keywords, constants, and literals.
- Performing error handling through reporting and recovery from illegal characters.
- Demonstrating functionality and validating correctness through sample inputs and test cases.
- Generating a token stream output file as interface with subsequent parser stage.
- Improving efficiency by optimizing regular expressions and analyzer logic.
- Documenting specifications, flowcharts, sample outputs and operational guidelines.

The coding methodology, data structures, and algorithm design skills applied during this project provide a foundation for building more complex compiler components like parsers and semantic analyzers. Insights on formally defining language grammar rules and converting into automated token recognition will be valuable for future language processing tasks.

Some potential extensions could be expanding the scope to the full C language, adding GUI interface, visualizing token streams, and integrating the lexical analyzer with parsing modules. Overall, this project served as an excellent learning experience in building foundational lexical analysis skills which can be applied to many language processing domains such as compilers, interpreters, static code analyzers, and text analytics tools.

FUTURE WORK

We have implemented the parser and Lexical analyzer for only a subset of C language. The future work may include defining the grammar and specifying the semantics for switch statements, predefined functions (like string functions, fileread and write functions), jump statements and enumerations.

REFERENCES

1. http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf - Lex and YaccTutorial by Tom Nieman
2. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, RaviSethi, Jeffrey D. Ullman
3. https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm - Compiler Design Semantic Analysis by TutorialsPoint