Lexical Analysis Phase Report
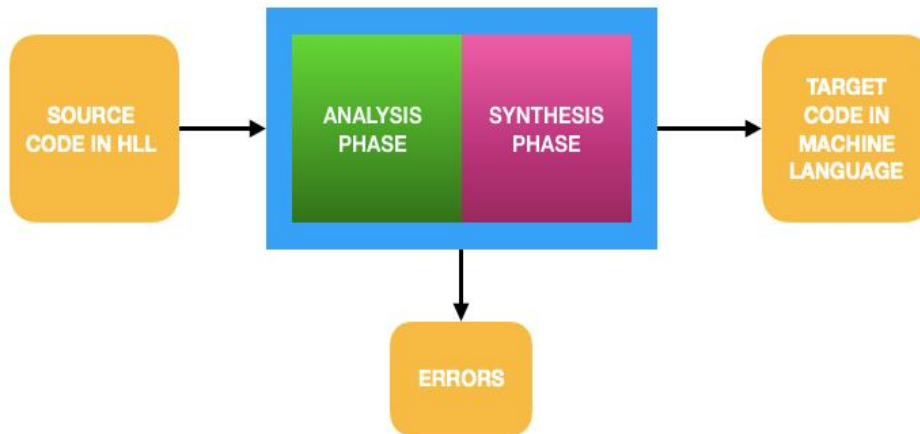
19.01.2018

COMPILER DESIGN PROJECT

—

Harita Reddy - 15CO217

Manali Gala - 15CO214

Introduction to a Compiler

A compiler is a software which converts a source code (program) written in high level language like C, which is also known as **Source Language** to low level language called as **Target Language**, which is machine understandable code. The equivalent and correct machine code that is generated, depends on the machine on which the compiler runs. A compiler also throws errors in the source code.

A **re-targettable or a cross compiler** produces code for some other machine whereas a **native compiler** produces code that can be understood only by the underlying machine. The user is expected to supply a correct source program that is syntactically, semantically and logically correct.

The objectives of a compiler are as follows:

1. **Error Handling Feature:** A compiler generally gives the line where the error is found, i.e, the line number, error message and a possible suggestion. The error message has to be given in a way that programmer can understand.
2. **Machine Code should be proportional to size of code:** The machine code should not be very large compared to the source code; it will affect both storage and the speed.
3. **Should be fast:** The compiler itself should run fast enough.

A compiler is different from interpreter in the fact that interpreters don't generate machine code and thus interpreters are more portable. However an interpreter has a disadvantage that whenever we give a source program, it has to understand the code again and run it from beginning to end. In C program, however, we can compile it once and run the executable again and again.

Phases of a Compiler

The compiler is mainly divided into two phases-

i) **Frontend**- This comprises of the Lexical Analyzer, Syntax Analyzer, Semantic Analyzer and the Intermediate Code Generator. It is also known as the **Analysis Phase.**

ii) **Backend**- This comprises of Code Optimization and Final Code Generation. This is also known as **Synthesis Phase.**
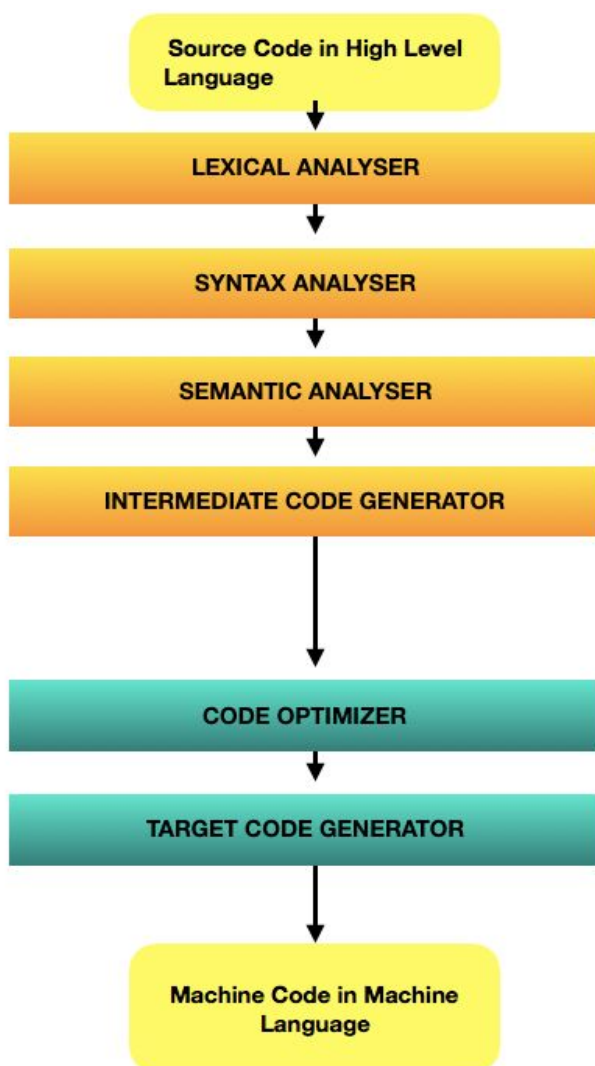
Analysis Phase

- **LEXICAL ANALYSIS:** In the this phase, the source code is read character by character and line by line to form groups of meaningful words from the code. These meaningful words are supplied to the syntactical analysis stage. It is done generally in one pass and is done by scanner. It breaks the syntax into a series of tokens, and removes white spaces and comments in the source code. It also generates error for invalid tokens.
- **SYNTACTICAL ANALYSIS:** The syntax of a given sentence cannot be checked by a lexical analyser due to the limitations of the regular expressions. This phase uses context free grammar and is also called as **parsing.** It produces syntactical errors and represents the information in the form of tree structure, also known as **parse tree.** Tokens provided by lexical analyser are grouped into meaningful sentences(larger constructs) at this stage.
- **SEMANTIC ANALYSIS:** Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not. Constructs of a language are given a meaning by the semantics. We can interpret symbols, their types and their relationships. The input to the semantic analyser is the parse tree produced by parser, and the output is a semantically verified parse tree.
  - **INTERMEDIATE CODE GENERATOR:** The semantically verified parse tree is given as an input to the intermediate code generator that generates the three address code.

```
Source Code in High Level
Language
        ↓
LEXICAL ANALYSER
        ↓
SYNTAX ANALYSER
        ↓
SEMANTIC ANALYSER
        ↓
INTERMEDIATE CODE GENERATOR
        ↓
CODE OPTIMIZER
        ↓
TARGET CODE GENERATOR
        ↓
Machine Code in Machine
Language
```

Synthesis Phase

CODE OPTIMIZATION: In this phase, the unwanted information is removed and optimization is performed.

The optimization can be machine dependent or machine independent.

FINAL CODE GENERATION: The object code is generated by this phase and this is machine code in machine language. The intermediate code generated by analysis phase is thus converted into machine understandable language.

Lexical Analysis

The lexical analyzer breaks the input code into a series of tokens. This is the only module in the compiler that interacts directly with the source program. A lexical analyzer produces lexical errors. Few important definitions related to this phase are:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,
- Identifiers
- Keywords
- Operators
- Special Characters
- Constants

Pattern: A pattern is a rule associated with a token and is used to match a sequence of characters with a particular token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token. It is an instance of token.

LEXICAL ANALYZER DESIGN ISSUES
Some issues associated with lexical analyzer are:
1. How does a lexical analyzer read a source program?
2. How are tokens specified?
3. Recognition of tokens.
4. Throwing Lexical Errors
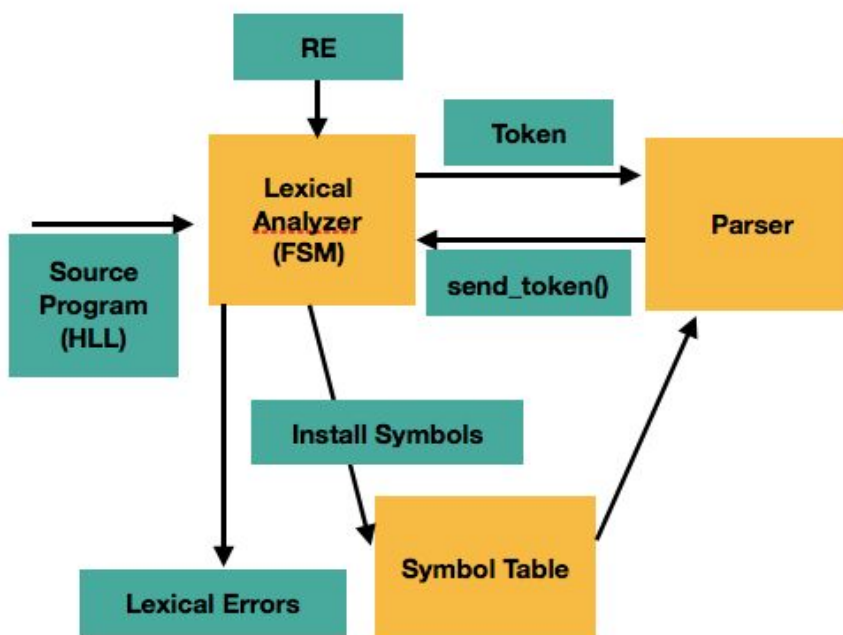
The parser invokes the lexical analyzer to produce tokens. Regular expressions are used to specify the tokens. Finite State Machine(FSM) is used to recognise tokens. A lexical analyzer is nothing but an FSM.

Symbol Table is a common data structure (usually implemented by hashing) used to store the identifiers and their related information.

WORKING OF A LEXICAL ANALYZER

1. Lexical analyzer strips out all the unwanted information from the source program like comments, blank spaces, new line characters and tabs.
2. It scans the source program from left to right and top to bottom, character by character.
3. According to the specifications given by the regular expressions, the characters are grouped into tokens. It inserts tokens into the symbol table.
4. It always goes for the longest match.
5. Ill-formed tokens(to which no rule matches) or illegal characters are printed on the screen.
6. Rule priority is important part of lexical analyzer.

Our Implementation of Lexical Analyzer

We have implemented the lexical analysis phase of the compiler using lex (a computer program that generates lexical analyzers and is commonly used with yacc parser generator). The lexer is implemented through C language ad reads an input stream. From the input stream, based on the specified regular expressions, the stream of characters are matched with the patterns and classified into tokens. These tokens are inserted into symbol and constant tables, which are printed at the end.

Comments

Comments are ignored and no tokens are taken from them.

Single Line Comments

Single line comments are represented by the regular expression: [/][/].*[\n]. Any line starting with // is ignored completely and no action is taken by the scanner.

Multi Line Comments

Multiple line comments are represented by the regular expression: [/][*](.*[\n].*)*[*][/]. Since these comments can span multiple lines, anything between /* and */ is ignored completely.

Because of the longest match property, all the comments get matched by their respective regular expressions first.

CHECKING FOR INCOMPLETE COMMENTS

If a multiline comment that has started is not terminated with a '*/' then the lexical analyzer throws an unterminated comment error.

Data Types

1. **FLOAT:** Our lexical analyzer supports float data type and throws error as well as suggestion when this keyword is slightly misspelled. The suggestion message is something like: Do you mean 'float'? It also recognizes erroneous identifiers that don't follow the prescribed format for identifiers.
2. **CHAR:** Our lexical analyzer supports char data type, char* as well as char** data types. It also recognizes erroneous identifiers that don't follow the prescribed format for identifiers.

The data type is specified as keyword and the variable declared with it is specified as an identifier if it conforms to rule of identifiers. Else, it gives **invalid identifier error**.

Constants

Constants are detected and stored in constant table along with their type.

1. **STRING CONSTANTS:** String literals or constants are a group of characters beginning with " and ending with ". The regular expression to recognise string constants is : **["][a-zA-Z0-9]*["]**
2. **CHARACTER CONSTANTS:** Character constants begin with ' and end with ', with a single character in between the single quotes. The regular expression to recognise character constants is : **[\'][a-zA-Z][\']**
3. **INTEGER AND FLOAT CONSTANTS:** Floating point constants have a decimal point where as integer constants don't. They can be either negative or positive. The regular expression to recognise integer or float constants is : **-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)**

Keywords

We support a range of keywords in C through our lexical analyzer through the following regular expression:

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

Loops

The while loop is detected in our lexical analyser.

Note : Function detection feature was not added since it conflicts with the while loop and also with some keywords like if, switch, for. Functions names are considered as identifiers.

Identifiers

The analyzer can detect identifier tokens, and throw error if it is an incorrect identifier format (Eg: 8a is an incorrect identifier as identifiers have to start with alphabet). Identifier tokens can be anything ranking from data type variables to function names.

Operators and Special Characters

Some other features that our lexical analyzer supports are:

1. **Binary Operators:** +,-,/,*,%,=,<,>
2. **Unary Operators:** --,++
3. **Special Characters:** {}()
4. **Delimiters:** ; and ,
5. Newline, Tabs and Spaces are ignored by the lexical analyzer.
6. Stray characters lead to error message. Stray character can also be an '*/' without a comment starting.

Symbol Table and Constant Table

We have implemented the symbol and constant table using hashing. In case of collision, chaining (using linked lists) is done. The hash function is based on the sum of the ASCII values of the characters in the tokens. Based on that, the position of a token in the hash table is calculated. The token name, its type and the number of times it has appeared in the source code is stored in the symbol table. We have used an array of structure pointers which are further chained at their respective places using structure pointers. Whenever a token is identified, we check if it is already present in the symbol table and simply increment the count if it does. Else, we insert the token into the table. Chaining is done through dynamic memory allocation.

Constants and their specific type (string, character, int or float constant) are stored in the constant table in similar fashion.

CONSTANT	CONSTANT TYPE	NUMBER OF INSTANCES

Code

```
%{
  #include<string.h>
  /* Buffers for storing float, char, char pointers declarations */
  char *floatbuffer[10];
  int floati=0;
  int floatn=0;
  int floatlengths[50];

  char *charbuffer[10];
  int chari=0;
  int charn=0;
  int charlengths[50];

  char *charptrbuffer[10];
  int charptri=0;
  int charptrn=0;
  int charptrlengths[50];
  /* Structure for implementing symbol and constant table */
  struct node
  {
        char name[50];
        char class[50];
        int count;
        struct node* next;
  } *symtable[53],*constable[53];

  int poscalc(char* name) //Calculates position of token in symbol table( hash function)
  {
        int i,pos=0;
        for(i=0;i<strlen(name);i++)
        {
                pos=pos+name[i]; //ASCII values of chacacters added
        }
        pos=pos%53;
        return pos; //returns hash value

  }
  int poscalccons(char* name) //Calculates position of constant in constant table( hash function)
  {
```

```
        int i,pos=0;
        for(i=0;i<strlen(name);i++)
        {
                pos=pos+name[i]; //ASCII values of chacacters added
        }
        pos=pos%53;
        return pos; //returns hash value

}
void insert(char* name,char* class) //Inserts the token in symbol table
{
        int pos=poscalc(name);
        if(symtable[pos]==NULL) //If there is no element already at that index
        {
                symtable[pos]=(struct node*)malloc(sizeof(struct node));
                strcpy(symtable[pos]->name,name);

                strcpy(symtable[pos]->class,class);
                symtable[pos]->next=NULL;
                symtable[pos]->count=1;
        }
        else //Chaining is required
        {
                int flag=0;
                struct node* check=symtable[pos];
                while(check!=NULL)
                {
                        if(strcmp(check->name,name)==0) //If that token already exists, increments count
                        {
                        flag=1;
                        check->count++;
                        break;
                }
                        check=check->next;
                }
                if(flag==1)
                return;
                struct node* ptr=(struct node*)malloc(sizeof(struct node));
                strcpy(ptr->name,symtable[pos]->name);
                strcpy(ptr->class,symtable[pos]->class);
                ptr->count=symtable[pos]->count;
                ptr->next=symtable[pos]->next;
                strcpy(symtable[pos]->name,name);
                strcpy(symtable[pos]->class,class);
                symtable[pos]->next=ptr;
                symtable[pos]->count=1;
        }
}
void insertincons(char* name,char* class) //Inserts token into constant table
{
        int pos=poscalc(name);
        if(constable[pos]==NULL) //If there is no element already at that index
        {
                constable[pos]=(struct node*)malloc(sizeof(struct node));
```

```
                        strcpy(constable[pos]->name,name);

                        strcpy(constable[pos]->class,class);
                        constable[pos]->next=NULL;
                        constable[pos]->count=1;
                }
            else //Chaining is required
            {
                        int flag=0;
                        struct node* check=constable[pos];
                        while(check!=NULL)
                        {
                                    if(strcmp(check->name,name)==0) //If that constant already exists, increments count
                                    {
                                    flag=1;
                                    check->count++;
                                    break;
                        }
                                    check=check->next;
                        }
                        if(flag==1)
                        return;
                        struct node* ptr=(struct node*)malloc(sizeof(struct node));
                        strcpy(ptr->name,constable[pos]->name);
                        strcpy(ptr->class,constable[pos]->class);
                        ptr->count=constable[pos]->count;
                        ptr->next=constable[pos]->next;
                        strcpy(constable[pos]->name,name);
                        strcpy(constable[pos]->class,class);
                        constable[pos]->next=ptr;
                        constable[pos]->count=1;
                }
    }

    int commstart=0; //For keeping track of multiline comments and checking if they have ended properly
%}
alpha [a-zA-Z]
digit [0-9]
scomment [/][/].*[\n]
mcommentstart [/][*]
mcommentend [*][/]
charconstant [\'][a-zA-Z][\']
preprocessor [#].+
floatvar [f][l][o][a][t][ |\n|\t]+[a-zA-Z0-9_]+
charvar [c][h][a][r][ |\n|\t]+[a-zA-Z0-9_]+
floatcap [Ff][Ll][Oo][Aa][Tt][ |\n|\t]+[a-zA-Z0-9_]+
charcap [Cc][Hh][Aa][Rr][ |\n|\t]+[a-zA-Z0-9_]+
charptr [c][h][a][r][*]+[ |\n|\t]+[a-zA-Z0-9_]+
kw
("float")|("char")|("auto")|("break")|("case")|("const")|("continue")|("main")|("default")|("do")|("double")|("else")|("enum")|("extern")|(
"for")|("goto")|("if")|("int")|("long")|("register")|("return")|("short")|("signed")|("sizeof")|("static")|("struct")|("switch")|("typedef")|("uni
on")|("unsigned")|("void")|("volatile")|("while")
whilevar [w][h][i][l][e]
strvar ["][a-zA-Z0-9]*["]
```

```
id [a-zA-Z][a-zA-Z0-9_]*
unaryop ("++")|("--")
op [+\-*%=<>/]|"=="
spchar [{}()]
strlenvar [s][t][r][l][e][n][(]([")][{digit}|{alpha}]*["]|[{alpha}]+[{digit}|{alpha}]*)[)]
constant -?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
wrongid [0-9_][a-zA-Z0-9_]+
delimiter [;,]
stray [$&`@~]

%%
{scomment} {
    printf("%-20s : Comment\n",yytext);

}

{mcommentstart} {
    if(commstart==0)
    commstart++;

}

{mcommentend} {
    if(commstart==1)
    commstart--;
    else printf("\nStray symbols found : %s",yytext);
}
{preprocessor} {

    if(commstart==0)
    printf("%-20s : Preprocessor directive: \n",yytext);
}

{floatvar} {
    if(commstart==0){
                    floatbuffer[floati]=yytext;
                    floatlengths[floati]=yyleng;
                    floati++;
                    floatn++;
                    insert("float","keyword");
                    printf("%s : datatype declaration\n",yytext);
                    }
            }
            {floatcap} {
                    if(commstart==0)
    printf("Error : %s : Did you mean 'float'?\n",yytext);
}
{charvar} {
    if(commstart==0){
                    charbuffer[chari]=yytext;
                    charlengths[chari]=yyleng;
                    chari++;
                    charn++;
                    insert("char","keyword");
```

```
            printf("%s : datatype declaration\n",yytext);
            }
        }


        {charcap} {
                if(commstart==0)
    printf("Error : %s : Did you mean 'char'?\n",yytext);
}
{charptr} { if(commstart==0){
                charptrbuffer[charptri]=yytext;
                charptrlengths[charptri]=yyleng;
                charptri++;
                charptrn++;
                }
        }
        {charconstant} {
                if(commstart==0){
    printf("%-20s : character constant\n",yytext);
    insert(yytext,"constant");
    insertincons(yytext,"char");
}
}
{whilevar} {
    if(commstart==0){
                printf("\n%-20s : keyword(loop)\n",yytext);
                insert(yytext,"keyword");
                }
        }
        {strvar} {
                if(commstart==0){
    printf("\n%-20s : string\n",yytext);
    insert(yytext,"string literal");
    insertincons(yytext,"string");
}
}

{kw} {
    if(commstart==0){
    printf("\n%-20s : keyword\n",yytext);
    insert(yytext,"keyword");
}
}
{id} { if(commstart==0)
    {
    printf("\n%-20s : identifier\n",yytext);
    insert(yytext,"identifier");
}
}
{delimiter} {
    if(commstart==0){
    printf("\n%-20s : separator\n",yytext);
    insert(yytext,"separator");
}
```

```
}
{unaryop} {
   if(commstart==0){
   printf("\n%-20s : unary operator\n",yytext);
   insert(yytext,"unary operator");
}
}
{op} {
   if(commstart==0){
   printf("\n%-20s : operator\n",yytext);
   insert(yytext,"binary operator");
}
}

{spchar} {
   if(commstart==0){
   printf("\n%-20s : special character\n",yytext);
   insert(yytext,"sp. character");
}
}

{strlenvar} {
}

{constant} {
   if(commstart==0){
   int i=0;
   printf("\n%-20s : constant\n",yytext);
   insert(yytext,"constant");
   int isfloat=0;
   for(i=0;i<yyleng;i++)
   {
           if(yytext[i]=='.')
           {
                   isfloat=1;
                   break;
           }
   }
   if(isfloat==1)
   insertincons(yytext,"float");
   else
   insertincons(yytext,"int");
}

}

{wrongid} {    if(commstart==0)

           printf("%s  : Invalid identifer\n",yytext);

}

{stray} {
   printf("%s : Stray character\n",yytext);
```

```
}


%%
int main()
{
    yyin = fopen("program.txt","r");
    yylex();

printf("\n\n\n\nRESULT\n\n");
int k,j;

if(commstart==1)
{
        printf("\nError : Unterminated Comment\n\n"); // The multi line comment hasn't been terminated
    }
    for(k=0;k<floatn;k++)
    {

        for(j=0;j<floatlengths[k];j++)
        {

                printf("%c", floatbuffer[k][j]);
        }
        printf("\n");
        printf("float : keyword\n");
        int p;
        for(p=5;p<floatlengths[k];p++)
        {
                if(floatbuffer[k][p]!=' ')
                break;
        }
        if(floatbuffer[k][p]>=48&&floatbuffer[k][p]<=57||floatbuffer[k][p]=='_')
        printf("Error: Invalid Identifier\n"); //Starting character of identifier should be alphabet
        else
        {
        for(j=5;j<floatlengths[k];j++)
        {
                if(floatbuffer[k][j]!=' '&&floatbuffer[k][j]!='\t'&&floatbuffer[k][j]!='\n')
                break;
        }
        char arr[32];
        int s=0;
        for(j=j;j<floatlengths[k];j++)
        {
                printf("%c",floatbuffer[k][j]);
                arr[s]=floatbuffer[k][j];
                s++;
        }
        arr[s]='\0';
        insert(arr,"identifier");
        printf(" : float identifier\n");
        printf("\n\n");
```

```
    }
}

    for(k=0;k<charn;k++)
    {

            for(j=0;j<charlengths[k];j++)
            {

                    printf("%c", charbuffer[k][j]);
            }
            printf("\n");
            printf("char : keyword\n");
            int p;
            for(p=4;p<charlengths[k];p++)
            {
                    if(charbuffer[k][p]!=' ')
                    break;
            }
            if(charbuffer[k][p]>=48&&charbuffer[k][p]<=57||charbuffer[k][p]=='_')
            printf("Error: Invalid Identifier\n"); //Starting character of identifier should be alphabet
            else
            {
            for(j=4;j<charlengths[k];j++)
            {
                    if(charbuffer[k][j]!=' '&&charbuffer[k][j]!='\t'&&charbuffer[k][j]!='\n')
                    break;
            }
            char arr[32];
            int s=0;
            for(j=j;j<charlengths[k];j++)
            {
                    printf("%c",charbuffer[k][j]);
                    arr[s]=charbuffer[k][j];
                    s++;
            }
            arr[s]='\0';
            insert(arr,"identifier");
            printf(" : char identifier\n");
            printf("\n\n");

    }
}

    for(k=0;k<charptrn;k++)
    {
            for(j=0;j<charptrlengths[k];j++)
            {

                    printf("%c", charptrbuffer[k][j]);
            }
            printf("\n");
            int starcount=0;
            for(j=0;j<charptrlengths[k];j++)
```

```
                {

                        if(charptrbuffer[k][j]=='*')
                        starcount++;
                }
                if(starcount==1)
                printf("char* : char pointer\n");
                else
                printf("char** : char pointer\n");
                int p;
                for(p=6;p<charptrlengths[k];p++)
                {

                        if(charptrbuffer[k][p]!=' ')
                        break;
                }
                if(charptrbuffer[k][p]>=48&&charptrbuffer[k][p]<=57||charptrbuffer[k][p]=='_')
                printf("Error: Invalid Identifier\n"); //Starting character of identifier should be alphabet
                else
                {
                for(j=6;j<charptrlengths[k];j++)
                {

                        if(charptrbuffer[k][j]!=' '&&charptrbuffer[k][j]!='\t'&&charptrbuffer[k][j]!='\n')
                        break;
                }
                char arr[32];
                int s=0;
                for(j=j;j<charptrlengths[k];j++)
                {

                        printf("%c",charptrbuffer[k][j]);
                        arr[s]=charptrbuffer[k][j];
                        s++;
                }
                arr[s]='\0';
                insert(arr,"identifier");
                printf(" : char pointer identifier\n");
                printf("\n\n");

        }
}

    printf("--SYMBOL TABLE--\n"); //Printing Symbol Table
    for(j=0;j<53;j++)
    {

            struct node* ptr=symtable[j];
            if(ptr==NULL)
            continue;
            printf("Position: %d\t",j);
            while(ptr!=NULL)
            {

                    printf("%-10s %-15s\t\t%-2d|\t",ptr->name,ptr->class,ptr->count);
                    ptr=ptr->next;
            }
```

```
            printf("\n");
    }
    printf("\n\n--CONSTANTS TABLE--\n"); //Printing Constant Table
    for(j=0;j<53;j++)
    {

            struct node* ptr=constable[j];
            if(ptr==NULL)
            continue;
            printf("Position: %d\t",j);
            while(ptr!=NULL)
            {

                    printf("%-10s %-15s\t\t%-2d|\t",ptr->name,ptr->class,ptr->count);
                    ptr=ptr->next;
            }
            printf("\n");
    }

}

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

Output Screenshots

The symbol table and constant tables' screenshots are attached below along with the test cases.

Test Cases

```
1.
#include<stdio.h>
int main()
{
        float a=9;
        char aa1='m';
        floatb=7;
        int i=4;
        while(a)
        {
                a--;
        }
        if(i%2==0)
        {
                continue;
        }
        else
        {
```

```
                rreturn(0);
        }
        return 0;
}
```

```
--SYMBOL TABLE--

Format : Lexeme, Class, Count(no. of times it has been repeated)


Position: 1     else       keyword                   1 |
Position: 2     7          constant                  1 |
Position: 3     m          identifier                1 |
Position: 4     9          constant                  1 |   float      keyword                   1 |
Position: 6     ;          separator                 8 |
Position: 7     while      keyword                   1 |
Position: 8     =          binary operator           4 |
Position: 13    int        keyword                   2 |
Position: 16    ==         binary operator           1 |
Position: 17    {          sp. character             4 |
Position: 19    }          sp. character             4 |
Position: 21    continue   keyword                   1 |
Position: 31    aa1        identifier                1 |
Position: 36    return     keyword                   1 |
Position: 37    %          binary operator           1 |   --         unary operator            1 |
Position: 40    (          sp. character             4 |
Position: 41    )          sp. character             4 |
Position: 43    char       keyword                   1 |
Position: 44    rreturn    identifier                1 |   a          identifier                3 |
Position: 48    0          constant                  3 |   if         keyword                   1 |
Position: 49    floatb     identifier                1 |
Position: 50    2          constant                  1 |   main       keyword                   1 |
Position: 52    4          constant                  1 |   i          identifier                2 |
```

```
--CONSTANTS TABLE--

Format : Constant, Datatype, Count(no. of times it has been repeated)


Position: 2     7          int                       1 |
Position: 4     9          int                       1 |
Position: 48    0          int                       3 |
Position: 50    2          int                       1 |
Position: 52    4          int                       1 |
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$
```

2.
```
#include<stdio.h>
int main()
{
        char 8a;
        float b10=5;
        char** var="NITK"
        if(a<0)
        {
                while(b10)
                {
                b10--;
                8a='c'+ int(b10);
                }
```

```
        }
        else
        {
                float a_=10;
                a_=a_/10;
        }
        return 0;
}
```

```
--SYMBOL TABLE--

Format : Lexeme, Class, Count(no. of times it has been repeated)


Position: 0     5          constant           1 |
Position: 1     else       keyword            1 |
Position: 4     float      keyword            2 |
Position: 6     ;          separator          7 |
Position: 7     while      keyword            1 |  <        binary operator      1 |  "NITK"    string literal       1 |
Position: 8     =          binary operator    5 |
Position: 11    var        identifier         1 |
Position: 13    int        keyword            2 |
Position: 17    {          sp. character      4 |
Position: 18    'c'        constant           1 |
Position: 19    }          sp. character      4 |
Position: 33    a_         identifier         3 |
Position: 36    return     keyword            1 |  b10      identifier           4 |
Position: 37    --         unary operator     1 |
Position: 40    (          sp. character      4 |
Position: 41    )          sp. character      4 |
Position: 43    +          binary operator    1 |  char     keyword              1 |
Position: 44    a          identifier         1 |
Position: 47    /          binary operator    1 |
Position: 48    0          constant           2 |  if       keyword              1 |
Position: 50    main       keyword            1 |
```

```
--CONSTANTS TABLE--

Format : Constant, Datatype, Count(no. of times it has been repeated)


Position: 0     5        int            1 |
Position: 7     "NITK"   string         1 |
Position: 18    'c'      char           1 |
Position: 48    0        int            2 |
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$
```

```
RESULT

float b10
float : keyword
b10 : float identifier


float a_
float : keyword
a_ : float identifier


char 8a
char : keyword
Error: Invalid Identifier
char** var
char** : char pointer
var : char pointer identifier
```

3.
```c
#include<stdio.h>
#include<string.h>
int main()
{
/*Test Case 3
  NITK Surathkal*/
   float v1AA =strlen("Surathkal") ;
   char** name="NITK";
   FLOAT a3;
   float Aav=strlen(name);
   v1AA = v1AA + Aav;
   v1AA= sterlen(name);
   return 0;
   &;
   @
}
```

```
--SYMBOL TABLE--

Format : Lexeme, Class, Count(no. of times it has been repeated)


Position: 4     "Surathkal" string literal          1 |   float     keyword             2 |
Position: 6     ;           separator                7 |
Position: 7     "NITK"      string literal           1 |
Position: 8     =           binary operator          5 |
Position: 13    int         keyword                  1 |
Position: 15    Aav         identifier               2 |
Position: 17    {           sp. character            1 |
Position: 19    }           sp. character            1 |
Position: 23    sterlen     identifier               1 |
Position: 28    strlen      identifier               2 |
Position: 32    v1AA        identifier               4 |
Position: 36    return      keyword                  1 |
Position: 40    (           sp. character            4 |
Position: 41    )           sp. character            4 |
Position: 43    +           binary operator          1 |
Position: 46    name        identifier               3 |
Position: 48    0           constant                 1 |
Position: 50    main        keyword                  1 |
```

```
--CONSTANTS TABLE--

Format : Constant, Datatype, Count(no. of times it has been repeated)


Position: 4     "Surathkal" string           1 |
Position: 7     "NITK"      string           1 |
Position: 48    0           int              1 |
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$
```

4.
```
#include<stdio.h>
int main()
{
//Test Case 4
/*This is
test case 4 */
   float a=-3.14;
   float b=-8.0
   while(a<0)
   {
   Float c=4;
   while(c>0)
   {
        c=c+5.0*a+b;
   }
   a=a+0.5;
   }
int ao1=9.0;
   while(ao1--)
   {
        ao1=ao1+1;
        break;
   }
   return 0;
```

}

```
--SYMBOL TABLE--

Format : Lexeme, Class, Count(no. of times it has been repeated)

Position: 4     float     keyword           2 |
Position: 6     ;         separator         8 |
Position: 7     <         binary operator   1 |  while     keyword          3 |
Position: 8     =         binary operator   7 |
Position: 9     >         binary operator   1 |
Position: 13    int       keyword           2 |
Position: 17    {         sp. character     4 |
Position: 19    }         sp. character     4 |
Position: 31    -3.14     constant          1 |
Position: 36    return    keyword           1 |  -8.0      constant         1 |
Position: 37    --        unary operator    1 |
Position: 40    break     keyword           1 |  (         sp. character    4 |
Position: 41    0.5       constant          1 |  5.0       constant         1 |  )        sp. character    4 |
Position: 42    *         binary operator   1 |
Position: 43    +         binary operator   4 |
Position: 44    a         identifier        5 |
Position: 45    9.0       constant          1 |  ao1       identifier       4 |  b        identifier       2 |
Position: 46    c         identifier        3 |
Position: 48    0         constant          3 |
Position: 49    1         constant          1 |
Position: 50    main      keyword           1 |
Position: 52    4         constant          1 |
```

```
--CONSTANTS TABLE--

Format : Constant, Datatype, Count(no. of times it has been repeated)

Position: 31    -3.14     float             1 |
Position: 36    -8.0      float             1 |
Position: 41    0.5       float             1 |  5.0       float            1 |
Position: 45    9.0       float             1 |
Position: 48    0         int               3 |
Position: 49    1         int               1 |
Position: 52    4         int               1 |
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$
```

5.
```c
#include<stdio.h>
#include<math.h>

struct{
        int count;
        char a;
};
void main()
{
        int c;
        switch(c)
        {
                case 1: sizeof(c);
                                break;
                case 2: c++;
                                break;
        }
}
```

```
--SYMBOL TABLE--

Format : Lexeme, Class, Count(no. of times it has been repeated)


Position: 6      ;          separator             8 |
Position: 10     void       keyword               1 |
Position: 13     int        keyword               2 |
Position: 17     {          sp. character         3 |
Position: 19     }          sp. character         3 |
Position: 20     sizeof     keyword               1 |
Position: 22     switch     keyword               1 |
Position: 23     count      identifier            1 |
Position: 33     ++         unary operator        1 |
Position: 40     break      keyword               2 |  (          sp. character        3 |
Position: 41     case       keyword               2 |  )          sp. character        3 |  struct     keyword            1 |
Position: 43     char       keyword               1 |
Position: 44     a          identifier            1 |
Position: 46     c          identifier            4 |
Position: 49     1          constant              1 |
Position: 50     2          constant              1 |  main       keyword              1 |
```

```
--CONSTANTS TABLE--

Format : Constant, Datatype, Count(no. of times it has been repeated)


Position: 49    1        int                      1 |
Position: 50    2        int                      1 |
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$ 
```

6.
```c
#include<stdio.h>
int main()
{
//Test Case 6
/*This is NITK
float a=0;
test case 6
   float a=-3.14;
   float b=-8.0
   while(a<0)
   {
   Float c=4;
   while(c>0)
   {
        c=c+5.0*a+b;
   }
   a=a+0.5;
   }
}
```

```
RESULT


Error : Unterminated Comment

--SYMBOL TABLE--
Position: 13     int        keyword                    1 |
Position: 17     {          sp. character              1 |
Position: 40     (          sp. character              1 |
Position: 41     )          sp. character              1 |
Position: 50     main       keyword                    1 |
```