SCANNER FOR C LANGUAGE

COMPILER DESIGN PROJECT REPORT

By

ARYAN SRIVASTAVA (RA2011033010013)
SACHIN S (RA2011033010038)

Guided by:

Dr. Sheryl Oliver A

In partial fulfilment for the Course

of

18CSC304J – COMPILER DESIGN

in CSE SPLZ IN SOFTWARE ENGINEERING

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that, this lab report titled **SCANNER FOR C-LANGUAGE** is the bonafide work done by:

ARYAN SRIVASTAVA (RA2011033010013)

SACHIN S (RA2011033010038)

who carried out the lab exercises under my supervision.

Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

Signature
Ms. Sheryl Oliver A
Course Faculty
Assistant Professor

Signature
Head Of Department
CINTEL

Date:5/5/2023

SCANNER FOR C LANGUAGE

ABSTRACT

A compiler is a program that can read a program in one language - the source language – and translate it to an equivalent program in another language - the target language. An important role of the compiler is to detect any errors in the source program during the translation process.

The first phase of a compiler is called **lexical analysis or scanning.** The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token that it passes on to the subsequent phase, syntax analysis.

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.
The lexical analyzer maintains a data structure called as the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme into the symbol table. The lexical analyzer performs certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace. Another task is correlating error messages generated by the compiler with the source program.

Functionality / use of the scanner in compiler -

- Identifying and categorizing each character in the source code based on its function in the language, such as identifying keywords, identifiers, constants, operators, and punctuations.
- Building a symbol table that maintains information about each identifier, such as its name, type, and scope.
- Generating error messages if the source code contains any lexical errors, such as misspelled keywords or identifiers that are not defined.
- Optimizing the input stream by removing white spaces and comments that are not relevant to the program's functionality.

How a Scanner works-

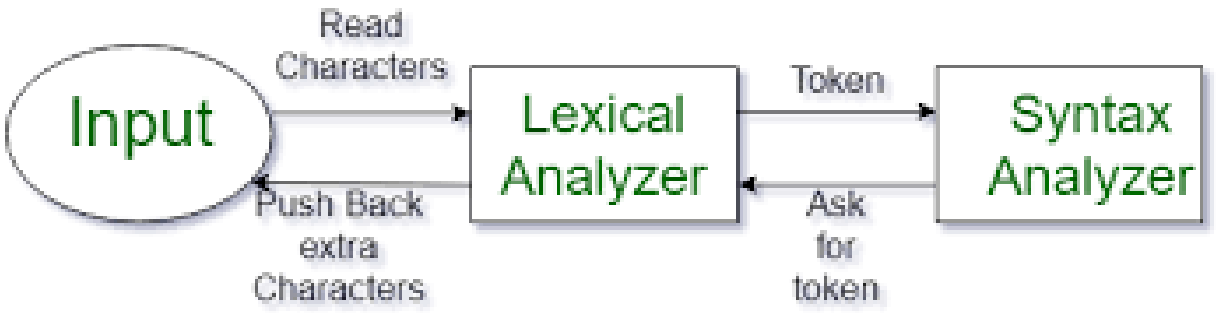
Input preprocessing involves cleaning up the input text by removing comments, whitespace, and other non-essential characters to prepare it for lexical analysis. This ensures that the input text is in a format that can be easily processed.

Tokenization involves breaking down the input text into a sequence of tokens, which are essentially the smallest units of meaning in the text. This is typically done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.

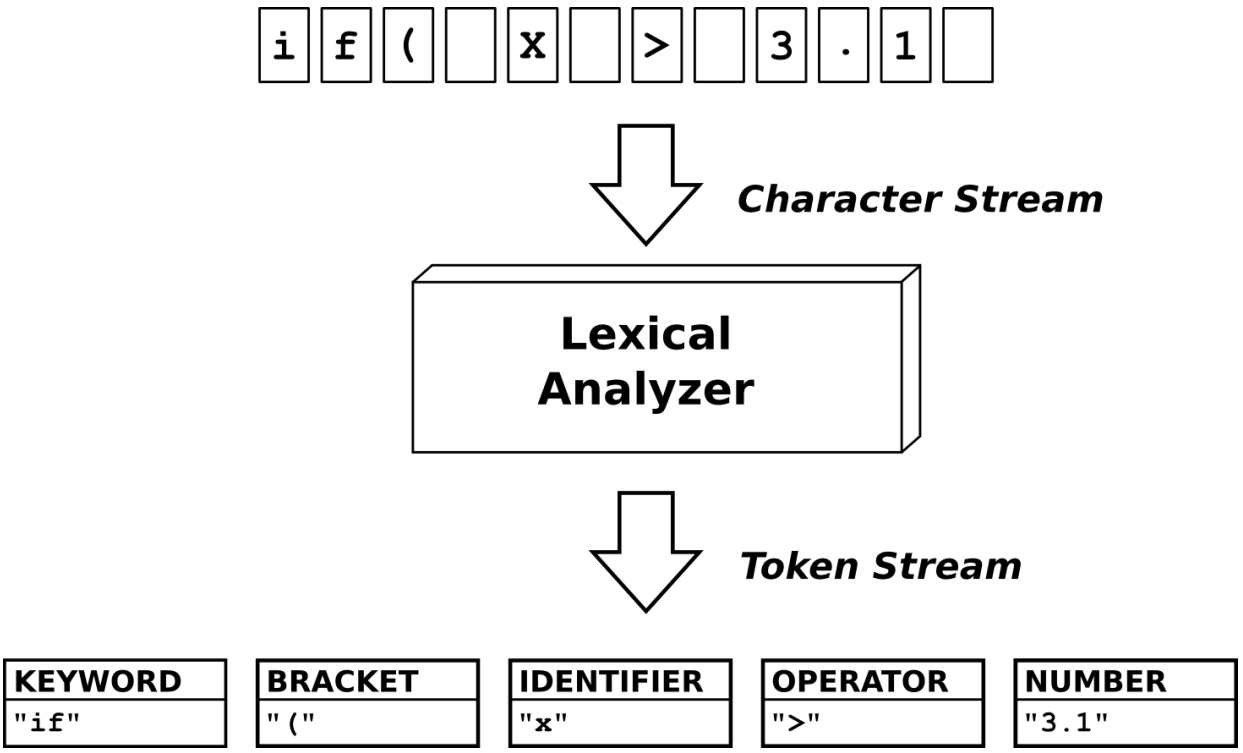
Token classification is the process of identifying the type of each token generated during tokenization. This can involve identifying keywords, identifiers, operators, punctuation symbols, and other token types as separate categories.

Token validation involves checking each token generated during tokenization to ensure that it is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.

Output generation is the final stage of the lexical analysis process, where the lexer generates the output, which is typically a list of tokens. This list of tokens can then be passed on to the next stage of compilation or interpretation.



Tokenization of a text:



HARDWARE AND SOFTWARE REQUIREMENTS

Hardware components used in this process are -

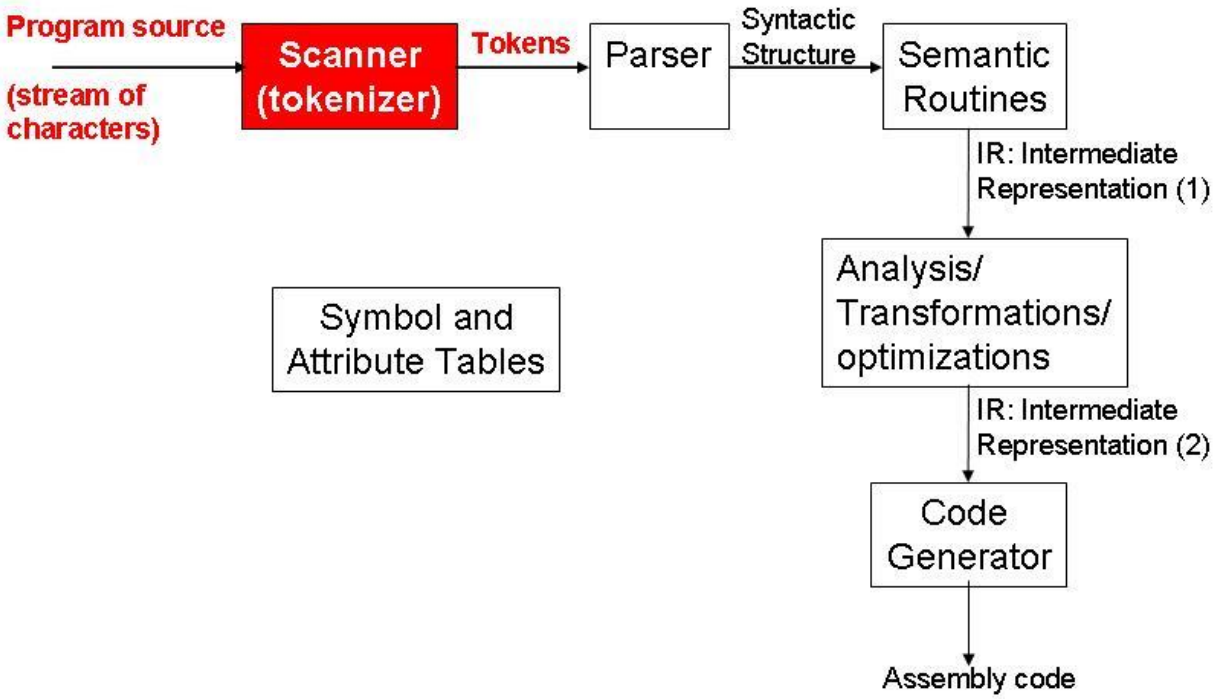
- Input device: This component is responsible for receiving the input source code written in the C language, such as a keyboard or a file storage device.
- Processor: The processor is responsible for executing the scanner program, which performs the lexical analysis of the input source code and generates a stream of tokens as output.
- Memory: The scanner program requires temporary memory to store the current lexeme being analyzed and the current state of the finite-state machine used to recognize the input source code's tokens.

Software components used in this process are -

- Scanner program: The scanner program is responsible for performing the lexical analysis of the input source code, which involves reading the input character by character, identifying and categorizing each character into a token, and generating a stream of tokens as output.
- Symbol table: The symbol table is a data structure used by the scanner to store information about each identifier found in the input source code, such as its name, type, and scope.
- Error handling module: The error handling module is responsible for detecting and reporting any lexical errors, such as misspelled keywords or undefined identifiers, in the input source code.
- Regular expressions: Regular expressions are used by the scanner to recognize patterns in the input source code and categorize them into tokens.

```
Input Stream -> Buffer -> Lexical Analyzer (Scanner) -> Token Stream
```

BLOCK DIAGRAM



METHODOLOGY USED

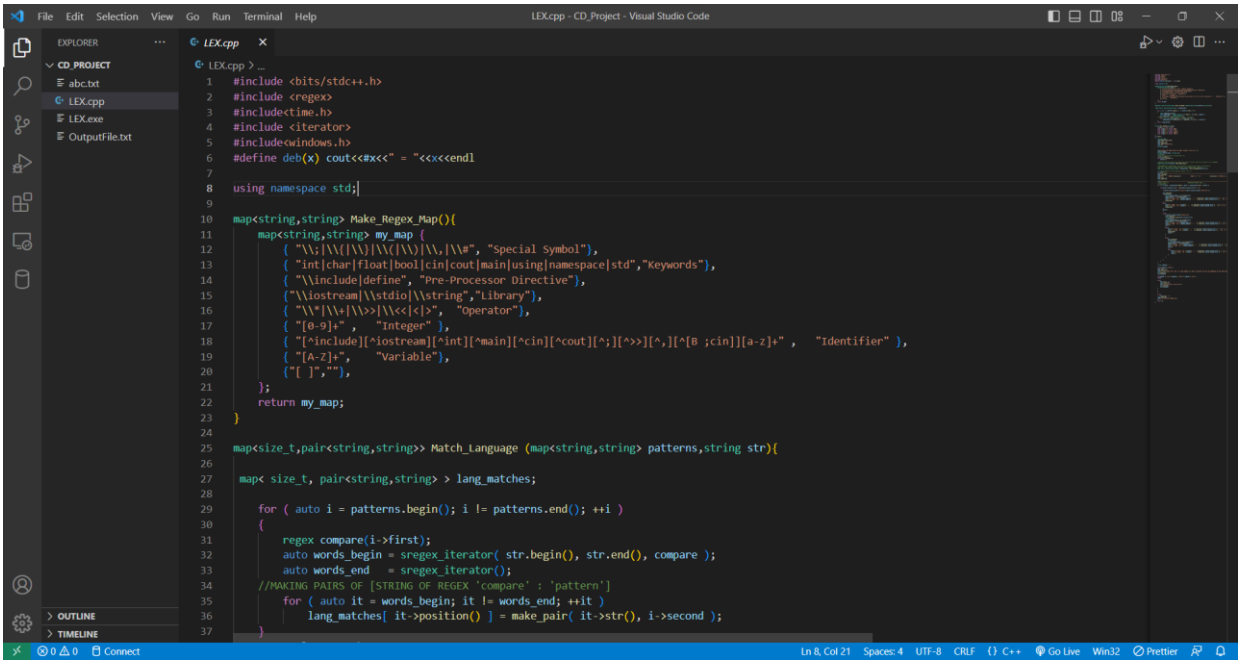
We have used Flex to perform lexical analysis on a subset of the C programming language.
Flex is a lexical analyzer generator that takes in a set of descriptions of possible tokens and produces a C file that performs lexical analysis and identifies the tokens.

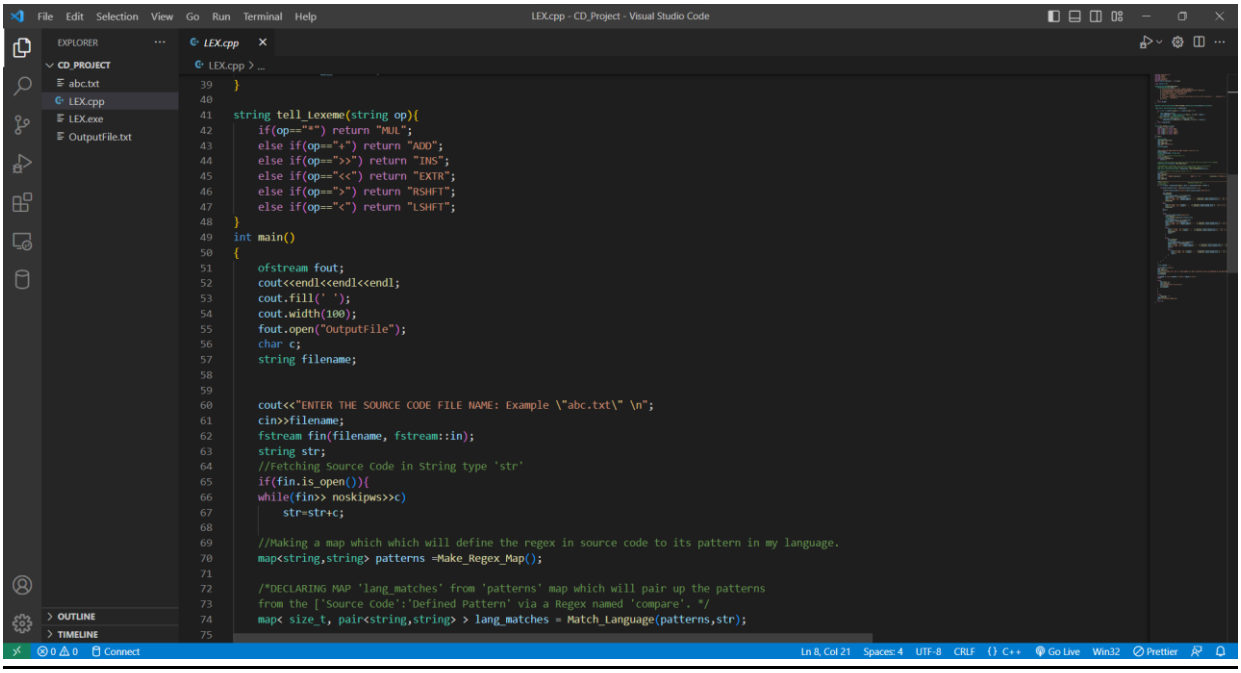
This document is divided into the following sections:
● **Functionality:** Contains a description of our Flex program and the variety of tokens that it can identify and the error handling strategies.
● **Symbol table and Constants table:** Contains an overview of the architecture of the symbol and constants table which contain descriptions of the lexemes identified during
lexical analysis.
● **Code Organisation:** Contains a description of the files used for lexical analysis
● **Source Code:** Contains the source code used for lexical analysis

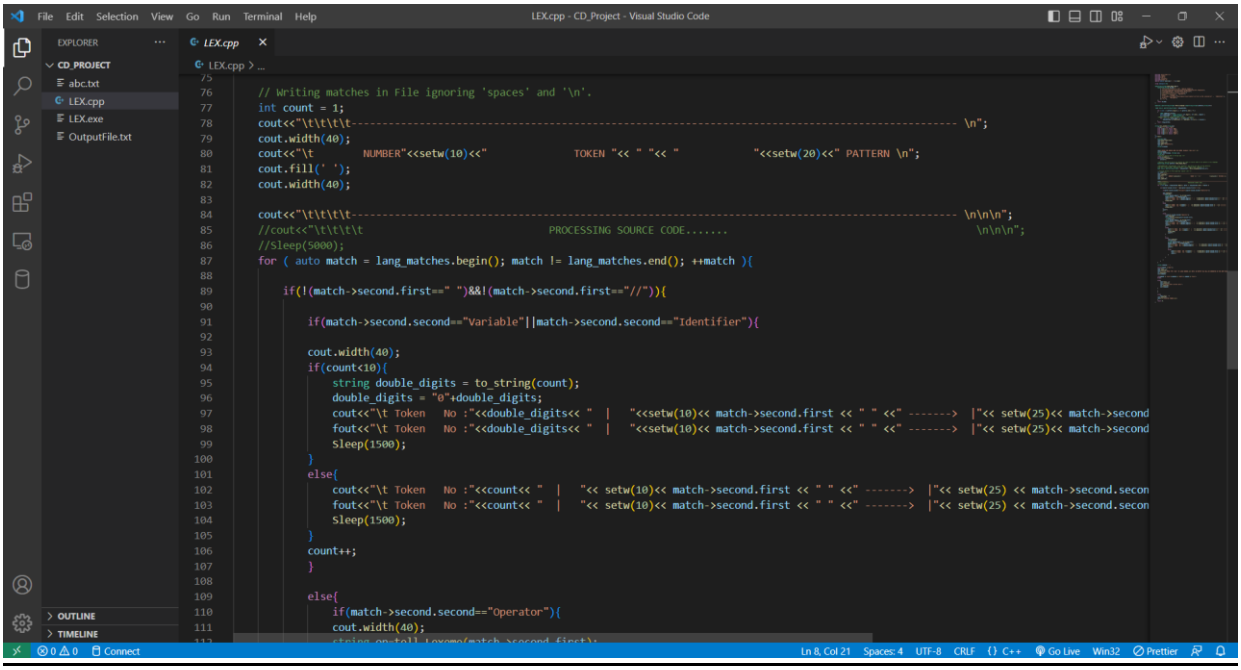
The entire code for lexical analysis is broken down into 3 files: lexer.l, tokens.h and symboltable.h

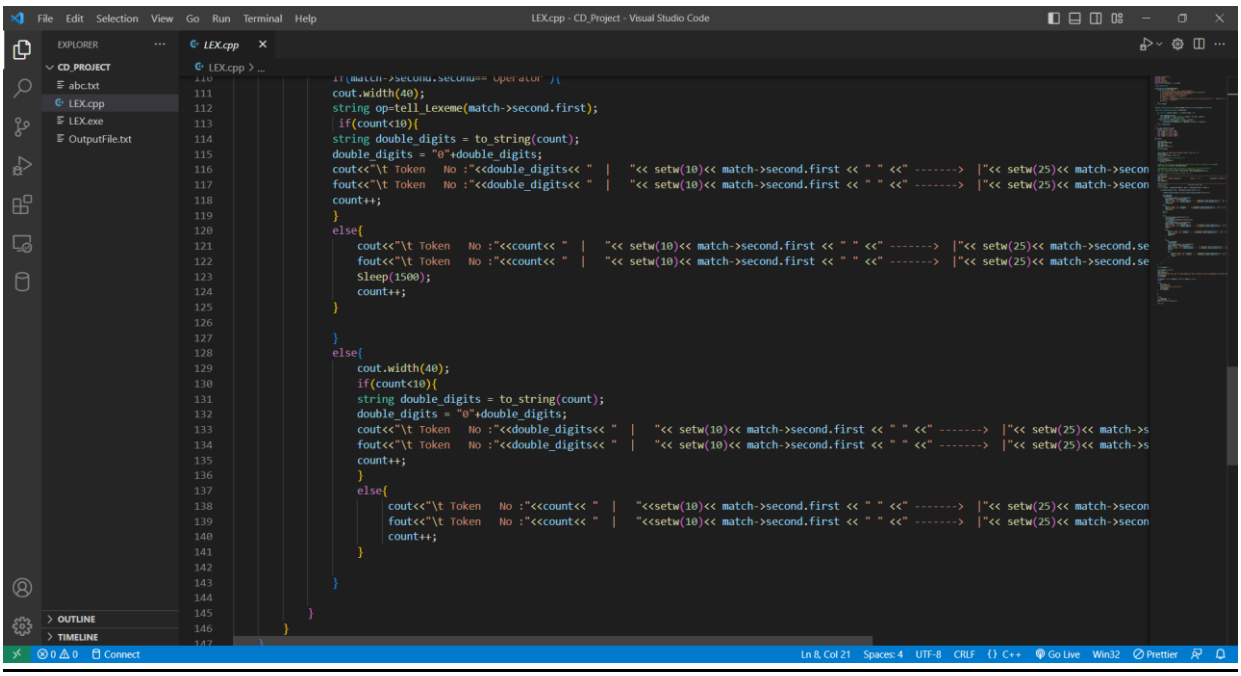
FILE	CONTENTS
lexer.l	A lex file containing the lex specification of regular expressions
tokens.h	Contains enumerated constants for keywords, operator, special symbols, constants and identifiers.
symboltable.h	Contains the definition of the symbol table and the constants table and also defines functions for inserting into the hash table and displaying its contents.

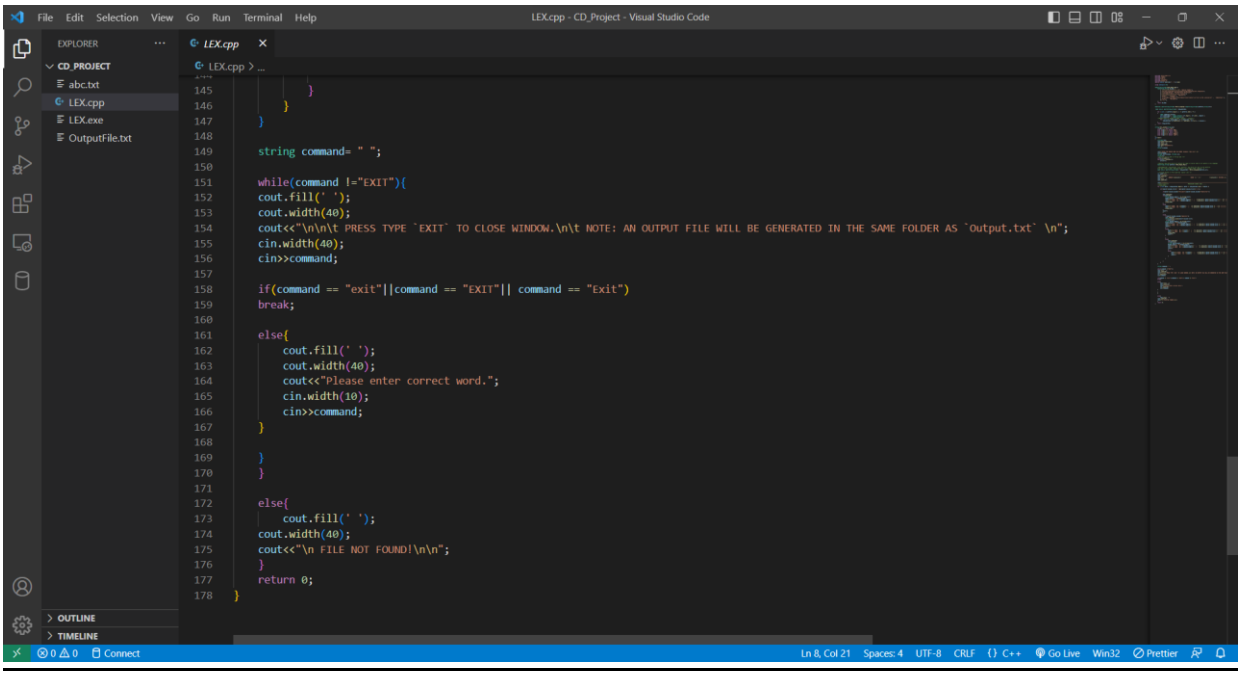
CODE


File Edit Selection View Go Run Terminal Help
LEX.cpp - CD_Project - Visual Studio Code
EXPLORER
CD_PROJECT
abc.txt
LEX.cpp
LEX.exe
OutputFile.txt
LEX.cpp
#include <bits/stdc++.h>
#include <regex>
#include<time.h>
#include <iterator>
#include<windows.h>
#define deb(x) cout<<#x<<" = "<<x<<endl
using namespace std;
map<string,string> Make_Regex_Map(){
map<string,string> my_map {
{ "\\;|\\{|\\}|\\(|\\)|\\,|\\#", "Special Symbol"},
{ "int|char|float|bool|cin|cout|main|using|namespace|std","Keywords"},
{ "\\include|define", "Pre-Processor Directive"},
{"\\iostream|\\stdio|\\string","Library"},
{ "*|\\+|\\>>|\\<<|<|>", "Operator"},
{ "[0-9]+" , "Integer" },
{ "[^include][^iostream][^int][^main][^cin][^cout][^;][^>>][^,][^[B ;cin]][a-z]+" , "Identifier" },
{ "[A-Z]+", "Variable"},
{"[ ]",""},
};
return my_map;
}
map<size_t,pair<string,string>> Match_Language (map<string,string> patterns,string str){
map< size_t, pair<string,string> > lang_matches;
for (auto i = patterns.begin(); i != patterns.end(); ++i)
{
regex compare(i->first);
auto words_begin = sregex_iterator(str.begin(), str.end(), compare);
auto words_end = sregex_iterator();
//MAKING PAIRS OF [STRING OF REGEX 'compare' : 'pattern']
for (auto it = words_begin; it != words_end; ++it)
lang_matches[it->position()] = make_pair(it->str(), i->second);
}
OUTLINE
TIMELINE
Connect
Ln 8, Col 21 Spaces: 4 UTF-8 CRLF C++ Go Live Win32 Prettier


File Edit Selection View Go Run Terminal Help
LEX.cpp - CD_Project - Visual Studio Code
EXPLORER
CD_PROJECT
abc.txt
LEX.cpp
LEX.exe
OutputFile.txt
}
string tell_Lexeme(string op){
if(op=="*") return "MUL";
else if(op=="+") return "ADD";
else if(op==">>") return "INS";
else if(op=="<<") return "EXTR";
else if(op==">") return "RSHFT";
else if(op=="<") return "LSHFT";
}
int main()
{
ofstream fout;
cout<<endl<<endl<<endl;
cout.fill(' ');
cout.width(100);
fout.open("OutputFile");
char c;
string filename;
cout<<"ENTER THE SOURCE CODE FILE NAME: Example \"abc.txt\" \n";
cin>>filename;
fstream fin(filename, fstream::in);
string str;
//Fetching Source Code in String type 'str'
if(fin.is_open()){
while(fin>> noskipws>>c)
str=str+c;
//Making a map which which will define the regex in source code to its pattern in my language.
map<string,string> patterns =Make_Regex_Map();
/*DECLARING MAP 'lang_matches' from 'patterns' map which will pair up the patterns
from the ['Source Code':'Defined Pattern' via a Regex named 'compare'. */
map< size_t, pair<string,string> > lang_matches = Match_Language(patterns,str);
OUTLINE
TIMELINE
Connect
Ln 8, Col 21 Spaces: 4 UTF-8 CRLF C++ Go Live Win32 Prettier

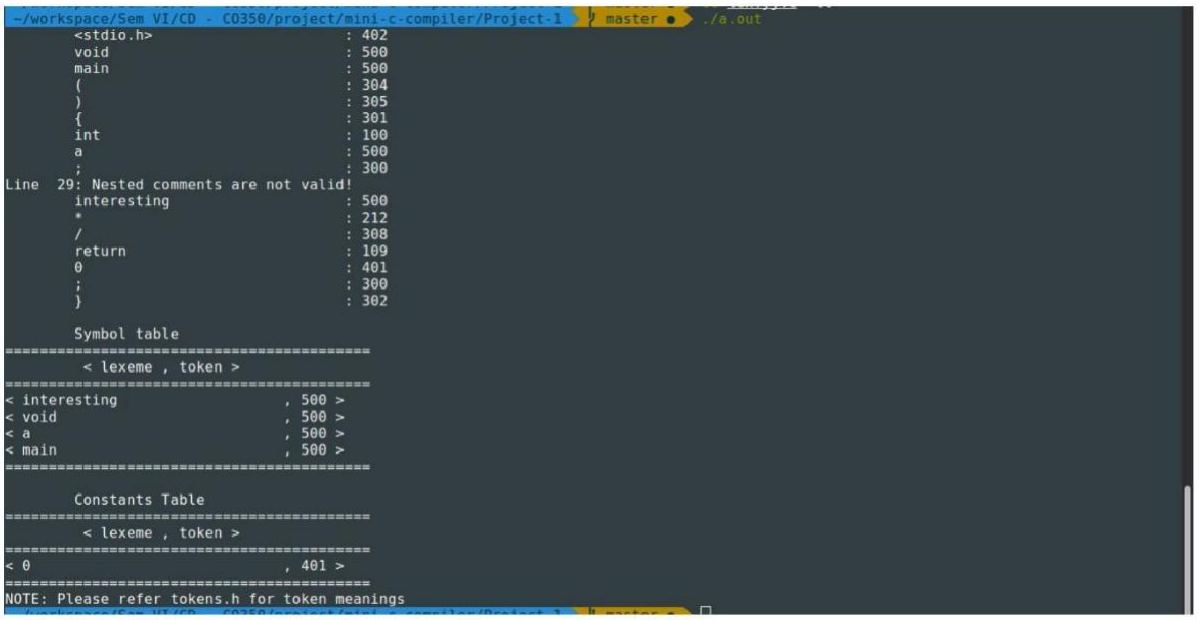

File Edit Selection View Go Run Terminal Help
LEX.cpp - CD_Project - Visual Studio Code
EXPLORER
CD_PROJECT
abc.txt
LEX.cpp
LEX.exe
OutputFile.txt
LEX.cpp
// Writing matches in File ignoring 'spaces' and '\n'.
int count = 1;
cout<<"\t\t\t\t\t----------\n";
cout.width(40);
cout<<"\t NUMBER"<<setw(10)<<" TOKEN "<< " "<< " "<<setw(20)<<" PATTERN \n";
cout.fill(' ');
cout.width(40);
cout<<"\t\t\t\t\t----------\n\n\n";
//cout<<"\t\t\t\t PROCESSING SOURCE CODE..... \n\n\n";
//Sleep(5000);
for (auto match = lang_matches.begin(); match != lang_matches.end(); ++match){
if(!(match->second.first==" ")&&!(match->second.first=="//")){
if(match->second.second=="Variable"||match->second.second=="Identifier"){
cout.width(40);
if(count<10){
string double_digits = to_string(count);
double_digits = "0"+double_digits;
cout<<"\t Token No :"<<double_digits<< " | "<<setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->second
fout<<"\t Token No :"<<double_digits<< " | "<<setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->second
Sleep(1500);
}
else{
cout<<"\t Token No :"<<count<< " | "<< setw(10)<< match->second.first << " " <<" -------> |"<< setw(25) << match->second.secon
fout<<"\t Token No :"<<count<< " | "<< setw(10)<< match->second.first << " " <<" -------> |"<< setw(25) << match->second.secon
Sleep(1500);
}
count++;
}
else{
if(match->second.second=="Operator"){
cout.width(40);
OUTLINE
TIMELINE
Ln 8, Col 21 Spaces: 4 UTF-8 CRLF {} C++ Go Live Win32 Prettier


File Edit Selection View Go Run Terminal Help
LEX.cpp - CD_Project - Visual Studio Code
EXPLORER
CD_PROJECT
abc.txt
LEX.cpp
LEX.exe
OutputFile.txt
LEX.cpp
cout.width(40);
string op=tell_Lexeme(match->second.first);
if(count<10){
string double_digits = to_string(count);
double_digits = "0"+double_digits;
cout<<"\t Token No :"<<double_digits<< " | "<< setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->secon
fout<<"\t Token No :"<<double_digits<< " | "<< setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->secon
count++;
}
else{
cout<<"\t Token No :"<<count<< " | "<< setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->second.se
fout<<"\t Token No :"<<count<< " | "<< setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->second.se
Sleep(1500);
count++;
}
}
else{
cout.width(40);
if(count<10){
string double_digits = to_string(count);
double_digits = "0"+double_digits;
cout<<"\t Token No :"<<double_digits<< " | "<< setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->s
fout<<"\t Token No :"<<double_digits<< " | "<< setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->s
count++;
}
else{
cout<<"\t Token No :"<<count<< " | "<<setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->secon
fout<<"\t Token No :"<<count<< " | "<<setw(10)<< match->second.first << " " <<" -------> |"<< setw(25)<< match->secon
count++;
}
}
}
}
OUTLINE
TIMELINE
Ln 8, Col 21 Spaces: 4 UTF-8 CRLF {} C++ Go Live Win32 Prettier


File Edit Selection View Go Run Terminal Help
LEX.cpp - CD_Project - Visual Studio Code
EXPLORER
CD_PROJECT
abc.txt
LEX.cpp
LEX.exe
OutputFile.txt
LEX.cpp
}
}
}
string command= " ";
while(command !="EXIT"){
cout.fill(' ');
cout.width(40);
cout<<"\n\n\t PRESS TYPE `EXIT` TO CLOSE WINDOW.\n\t NOTE: AN OUTPUT FILE WILL BE GENERATED IN THE SAME FOLDER AS `Output.txt` \n";
cin.width(40);
cin>>command;
if(command == "exit"||command == "EXIT"|| command == "Exit")
break;
else{
cout.fill(' ');
cout.width(40);
cout<<"Please enter correct word.";
cin.width(10);
cin>>command;
}
}
}
else{
cout.fill(' ');
cout.width(40);
cout<<"\n FILE NOT FOUND!\n\n";
}
return 0;
}
OUTLINE
TIMELINE
Ln 8, Col 21 Spaces: 4 UTF-8 CRLF {} C++ Go Live Win32 Prettier

RESULTS

```
Test Case 1
 - Test for single line comments
 - Test for multi-line comments
 - Test for single line nested comments
 - Test for multiline nested comments

The output in lex should remove all the comments including this one
*/

#include<stdio.h>

void main(){
        // Single line comment

        /* Multi-line comment
        Like this */

        /* here */ int a; /* "int a" should be untouched */

    // This nested comment // This comment should be removed should be removed



    /* To make things /*  nested multi-line comment */ interesting */

    return 0;
}
```



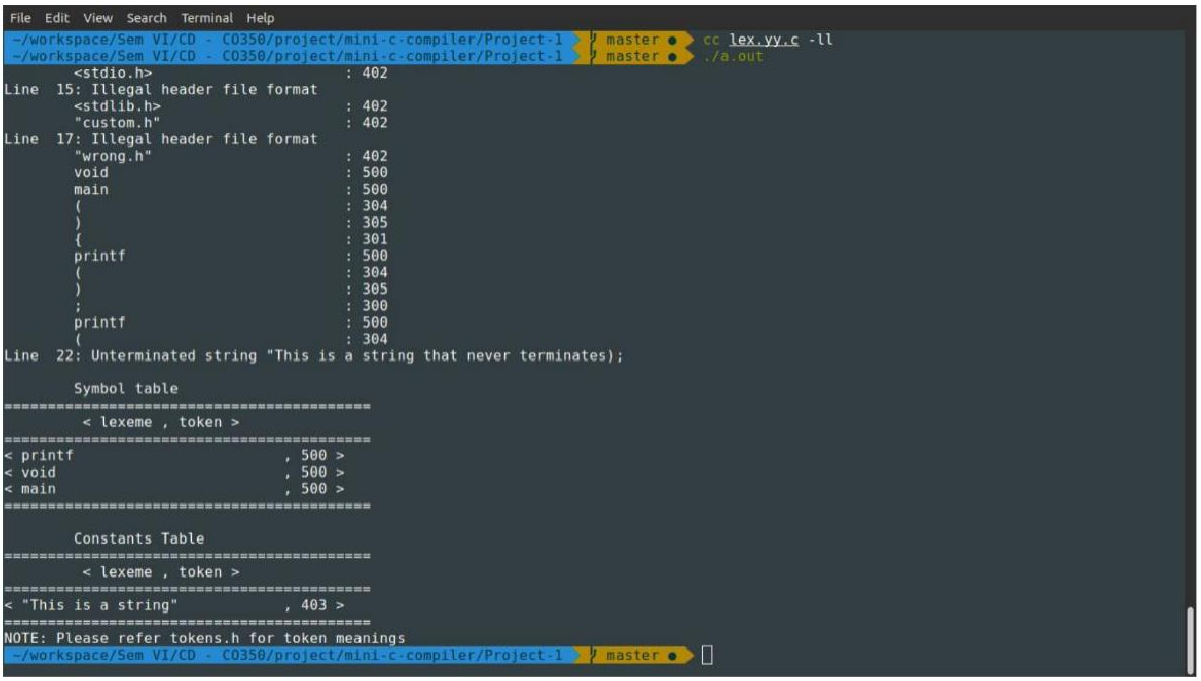
test-case-1

```
Test Case 2
 - Test for multi-line comment that doesn't end till EOF

The output in lex should print as error message when the comment does not terminate
It should remove the comments that terminate
*/

#include<stdio.h>

void main(){

    // This is fine
    /* This as well
    like we know */

    /* This is not fine since
    this comment has to end somewhere

    return 0;
}
```

```
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  ./a.out
        <stdio.h>                       : 402
        void                            : 500
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
Line  21: Unterminated comment

        Symbol table
=========================================
         < lexeme , token >
=========================================
< void                          , 500 >
< main                          , 500 >
=========================================

        Constants Table
=========================================
         < lexeme , token >
=========================================
=========================================
NOTE: Please refer tokens.h for token meanings
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●
```

test-case-2

```
Test Case 3
 - Test for string
 - Test for string that doesn't end till EOF
 - Test for invalid header name

The output in lex should identify the first string correct and display error message that
the second one does not terminate
*/

#include<stdio.h>
#include <<stdlib.h>
#include "custom.h"
#include ""wrong.h"

void main(){

    printf("This is a string");
    printf("This is a string that never terminates);
}
```

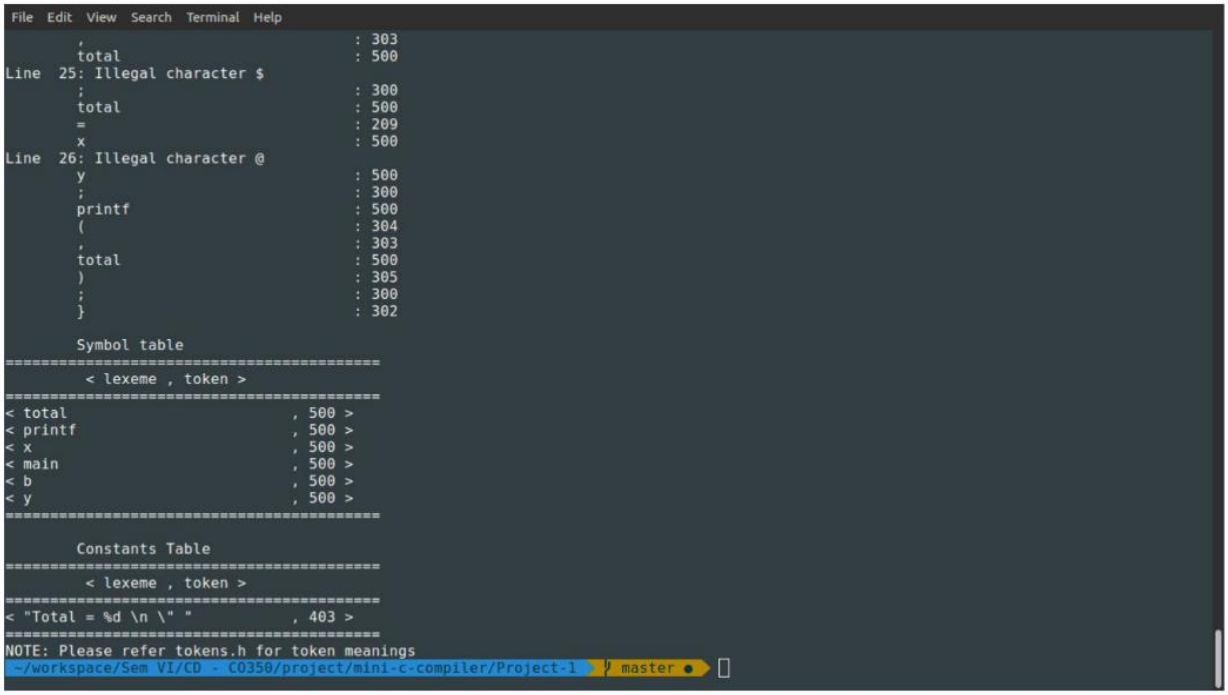


test-case-3

```
Test Case 4

Following errors must be detected
 - Invalid identifiers: 9y, total$
 - Invalid operator: @
 - Escaped quoted should be part of the string that is identified
 - Stray characters: `, @, -

The output should display appropriate errors
*/

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

int main()
{
  `
  @ -
  short int b;
  int x, 9y, total$;
  total = x @ y;
  printf ("Total = %d \n \" ", total);
}
```

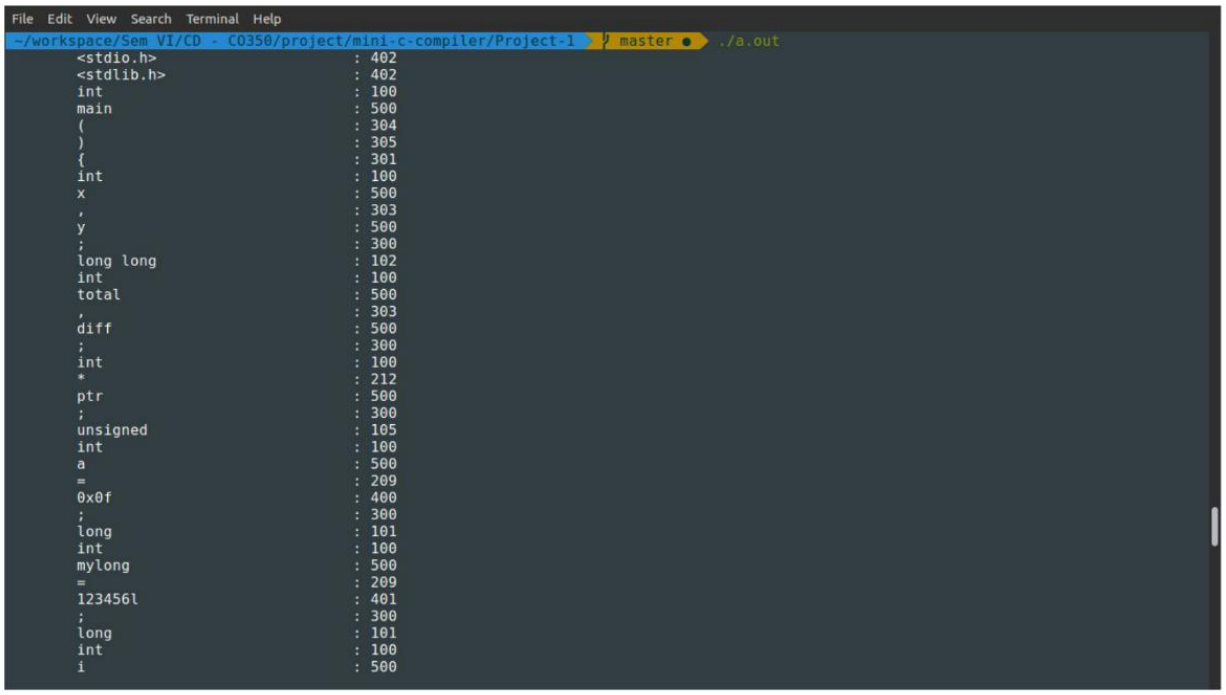
```
File  Edit  View  Search  Terminal  Help
~/workspace/Sem VI/CD - CO350/project/mini-c-compiler/Project-1  master ●  ./a.out
        <stdio.h>                       : 402
        <stdlib.h>                      : 402
        int                             : 100
        main                            : 500
        (                               : 304
        )                               : 305
        {                               : 301
Line  22: Illegal character `
Line  23: Illegal character @
        -                               : 210
        short                           : 103
        int                             : 100
        b                               : 500
        ;                               : 300
        int                             : 100
        x                               : 500
        ,                               : 303
Line  25: Illegal identifier name 9y
        ,                               : 303
        total                           : 500
Line  25: Illegal character $
        ;                               : 300
        total                           : 500
        =                               : 209
        x                               : 500
Line  26: Illegal character @
        y                               : 500
        ;                               : 300
        printf                          : 500
        (                               : 304
        ,                               : 303
        total                           : 500
        )                               : 305
        ;                               : 300
        }                               : 302

        Symbol table
```

Test-case-4a

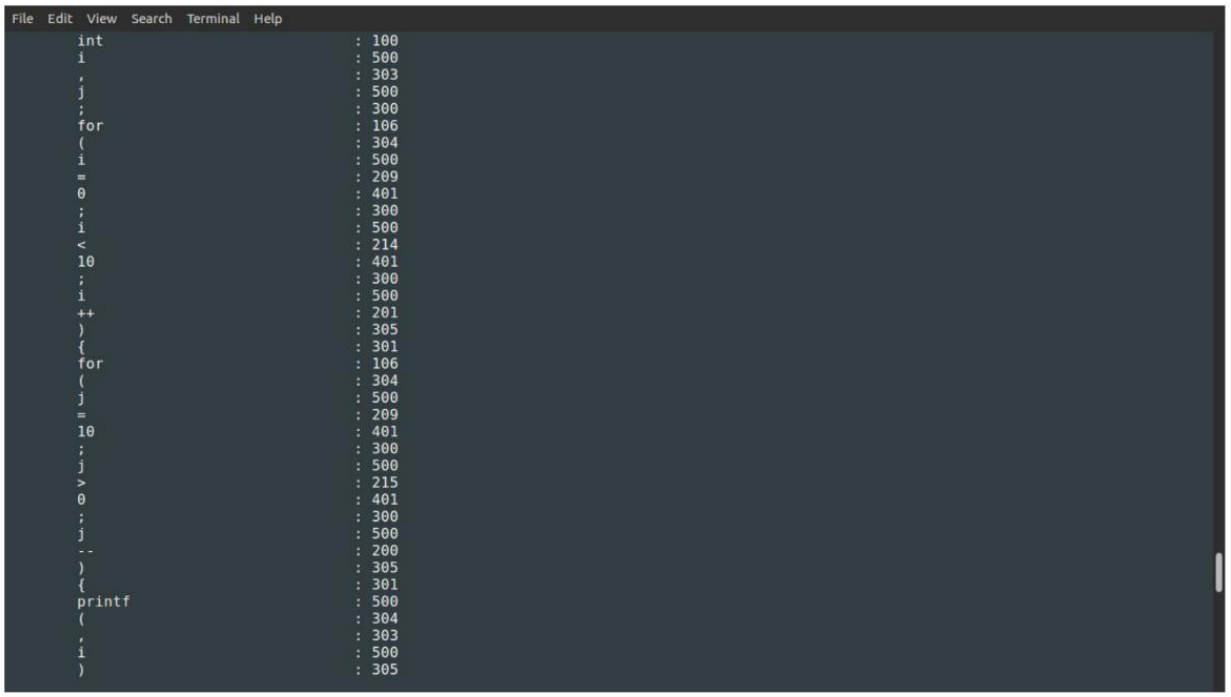


test-case-4b

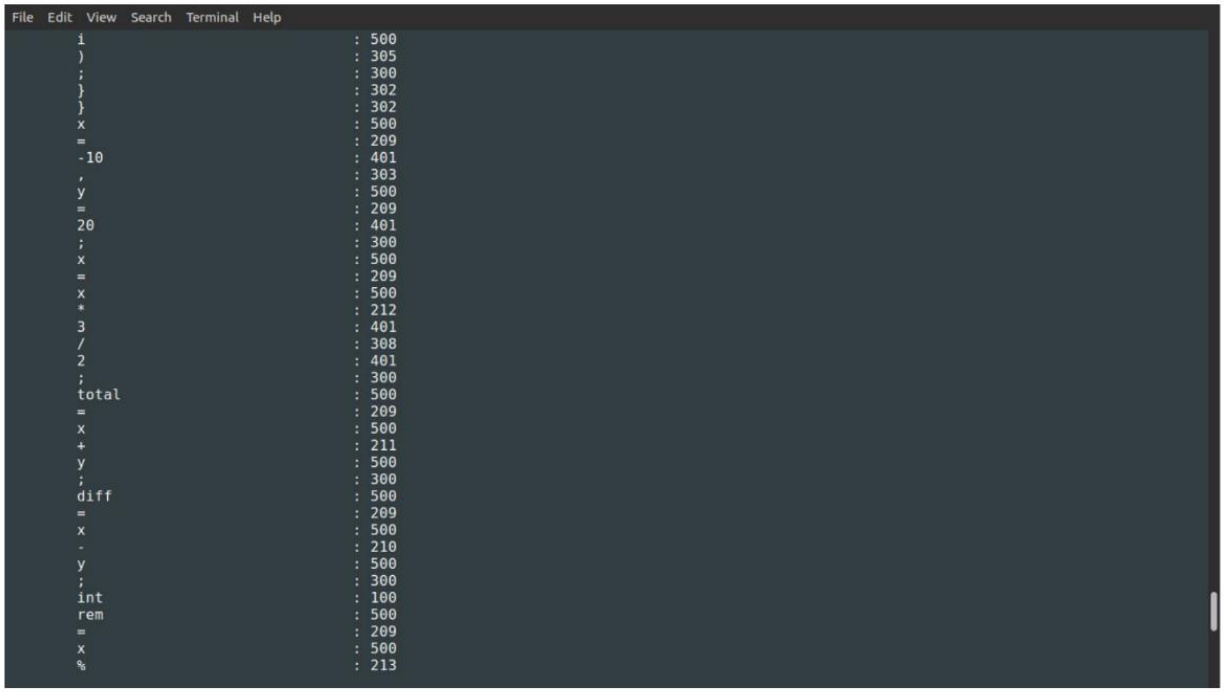
```
/*
Test Case 5

Identifying tokens and displaying symbol and constants table

Following tokens must be detected
 - Keywords    (int, long int, long long int, main include)


 - Identifiers (main,total,x,y,printf),
 - Constants  (-10, 20, 0x0f, 123456l)
 - Strings ("Total = %d \n")
 - Special symbols and Brackets ( (), {}, ;, ,)
 - Operators (+,-,=,*,/,%,--,++)

The output should display appropriate tokens with their type and also the symbol and
constants table
*/
#include<stdio.h>
#include<stdlib.h>

int main()
{
  int x, y;
  long long int total, diff;
  int *ptr;
  unsigned int a = 0x0f;
  long int mylong = 123456l;
  long int i, j;
  for(i=0; i < 10; i++){
        for(j=10; j > 0; j--){
        printf("%d",i);
        }
  }
  x = -10, y = 20;
  x=x*3/2;
  total = x + y;
  diff = x - y;
  int rem = x % y;
  printf ("Total = %d \n", total);
}
```



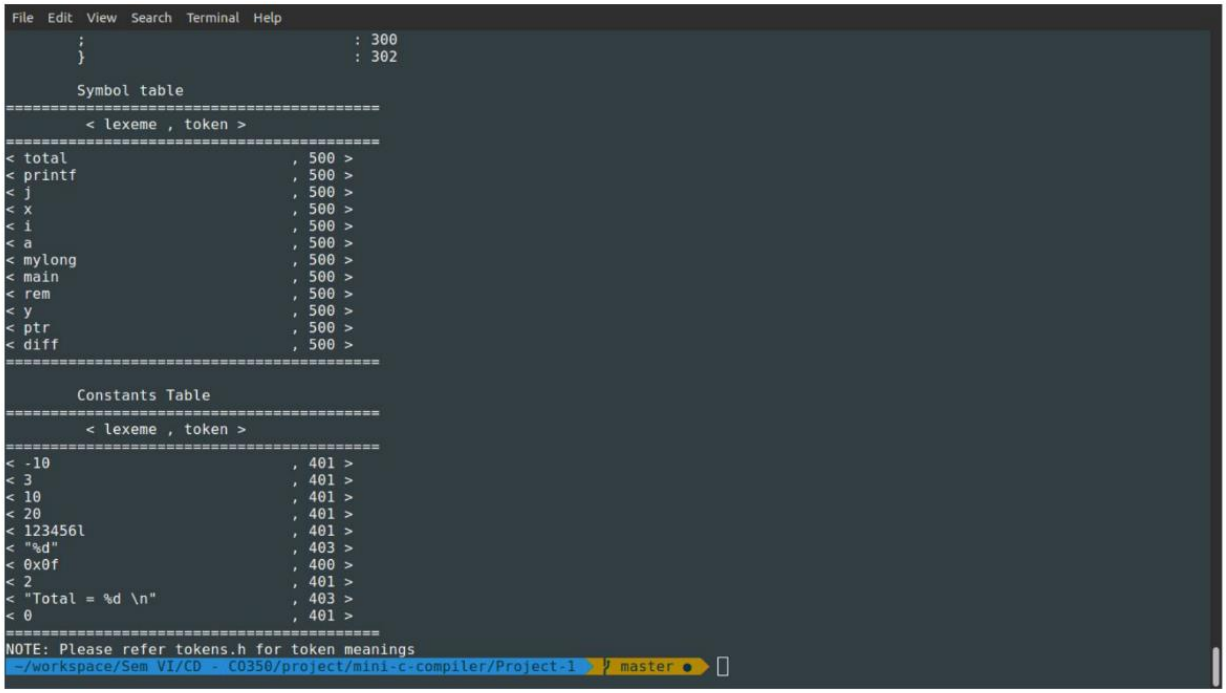
test-case-5a



test-case-5b



test-case-5c



test-case-5d

CONCLUSION

The lexical analyzer that was created in this project helps us to break down a C source file into tokens as per the C language specifications. Each token (such as identifiers, keywords, special symbols, operators, etc.) has an integer value associated with it.

When we design the parser in the next phase, the parser will call upon the Flex program to give it tokens and the lexical analyzer will return to the parser the integer value associated with the tokens as and when required by the parser. Together with the symbol, the parser will prepare a syntax tree with the help of a grammar that we provide it with. The parser can then logically group the tokens to form meaningful statements and can detect C programming constructs such as arrays, loops, and functions. The parser will also help us identify errors that could not be detected in the lexical analysis phase such as unbalanced parentheses, unterminated statements, missing operators, two operators in a row, etc.