COMPILER DESIGN LAB

CS304

**Project Phase1: Scanner for C- language**

****

**M Shree Harsha Bhat**

211CS137

[mshreeharshabhat.211cs137@nitk.edu.in](mailto:mshreeharshabhat.211cs137@nitk.edu.in)

**Shreekara Rajendra**

211CS151

[shreekararajendra.211cs151@nitk.edu.in](mailto:shreekararajendra.211cs151@nitk.edu.in)

**Shyam Balaji**

211CS154

[shyambalaji.211cs154@nitk.edu.in](mailto:shyambalaji.211cs154@nitk.edu.in)

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA**

**SURATHKAL, MANGALORE – 57025**

**Introduction**

Compiler design is a complex field of computer science that focuses on creating software tools called compilers. Compilers are essential for translating high-level programming languages into low-level machine code or other intermediate representations that can be executed by a computer's processor.

Here are the different stages in building a compiler:

**Lexical Analysis (Scanning):** The first phase of compilation, where the source code is analyzed to recognize basic building blocks called tokens, such as keywords, identifiers, constants, and operators. Lexical analyzers use regular expressions and finite automata to perform this task.

**Syntax Analysis (Parsing):** In this phase, the compiler ensures that the sequence of tokens from the lexical analysis forms valid syntactic structures according to the language's grammar rules. Parsing is typically done using techniques like LL(k) or LR(k) parsing, and it produces a parse tree or abstract syntax tree (AST).

**Semantic Analysis:** This phase checks the meaning and consistency of the code. It enforces type-checking, and scope rules and detects semantic errors. The output is usually an annotated AST.

**Intermediate Code Generation:** The compiler generates an intermediate representation of the source code that is closer to machine code but still platform-independent. This representation simplifies code optimization and target code generation.

**Code Optimization:** Compiler optimization techniques aim to improve the efficiency of the generated code. This includes techniques like constant folding, dead code elimination, loop optimization, and more.

**Phase 1: Lexical Analysis**

Lexical analysis, also known as scanning or tokenization, is the process of breaking down a stream of characters from the source code into a sequence of tokens.

Tokens are the basic building blocks of a programming language, such as keywords, identifiers, constants, operators, and special symbols. Tokenization involves reading characters from the source code and grouping them into tokens based on specific rules defined by the programming language's syntax.

**Regular Expressions and Finite Automata:**

Lexical analyzers use regular expressions to describe the patterns of valid tokens in the source code.

Finite automata (DFA or NFA) are commonly employed to recognize these regular expressions efficiently

**Error Handling:**

Lexical analyzers should provide meaningful error messages when encountering invalid tokens or syntax errors.

**Symbol Table:**

In addition to recognizing tokens, the lexer may maintain a symbol table to keep track of identifiers, their names, and other information relevant to the parser and later phases of compilation.

**Output:**

The output of the lexical analysis phase is a sequence of tokens, often represented as a stream or list of records, each containing a token type and (optionally) associated data.

**Lexical Code Format:**

**1.Definitions Section (%{ ... %}):**

The Definitions Section is enclosed within %{ and %} delimiters.

Purpose:

This section is used to include C code and user-defined macros that will be copied verbatim into the generated lexer code. It allows you to define custom C code that can be used within your lexer.

Content:

**C Code:** You can include C code for various purposes, such as importing standard C libraries, declaring global variables, or defining functions that are used within the lexer actions.

**Macros:** You can define macros that can simplify your lexer rules or actions. For example, you might define a macro for error handling or frequently used regular expressions.

**2. Rules Section (%%):**

The Rules Section is demarcated by a double-percent symbol (%%).

Purpose:

This is the core of your lexer specification. It defines the regular expressions and the corresponding actions that are executed when these regular expressions match input tokens.

Rules: Each rule consists of a regular expression pattern followed by an associated action. The pattern describes the token's pattern, and the action is the code executed when that pattern is matched.

**3.User Code Section (%{ ... %}):**

The User Code Section is also enclosed within %{ and %} delimiters.

Purpose:

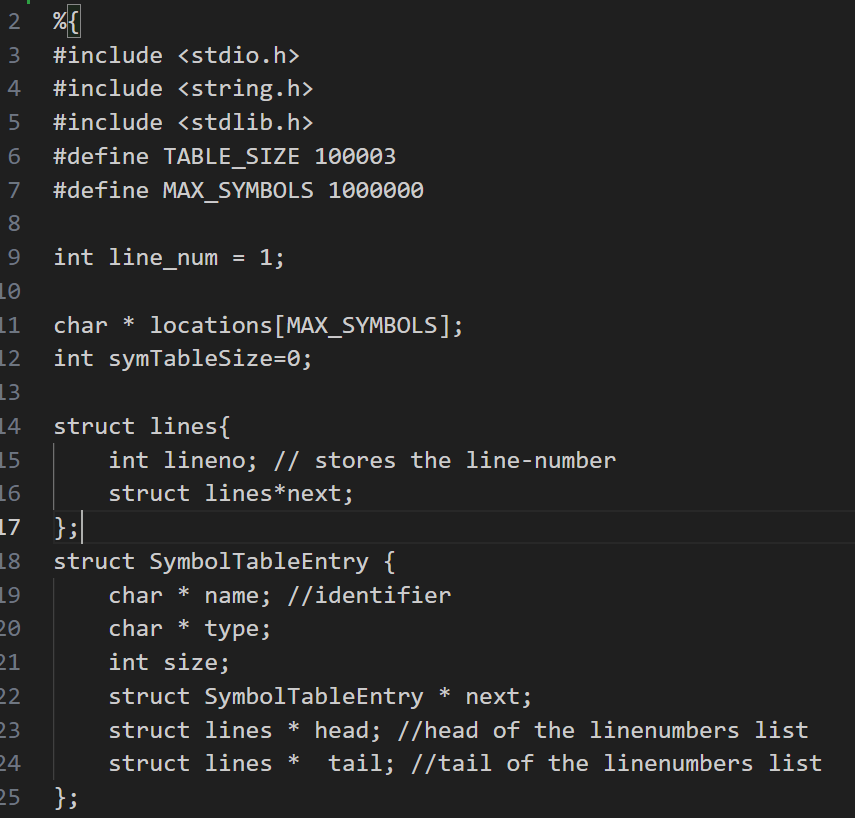
This section allows you to include additional C code that is copied directly into the generated lexer code. It's useful for adding custom functions, including external headers, or defining other global variables.

Content:

C Code: You can include any valid C code in this section. This code can be used to extend the functionality of your lexer or perform additional processing.

**Code Explanation**

1. **Declaration Section**



This code defines constants

and data structures to manage

a symbol table, including hash

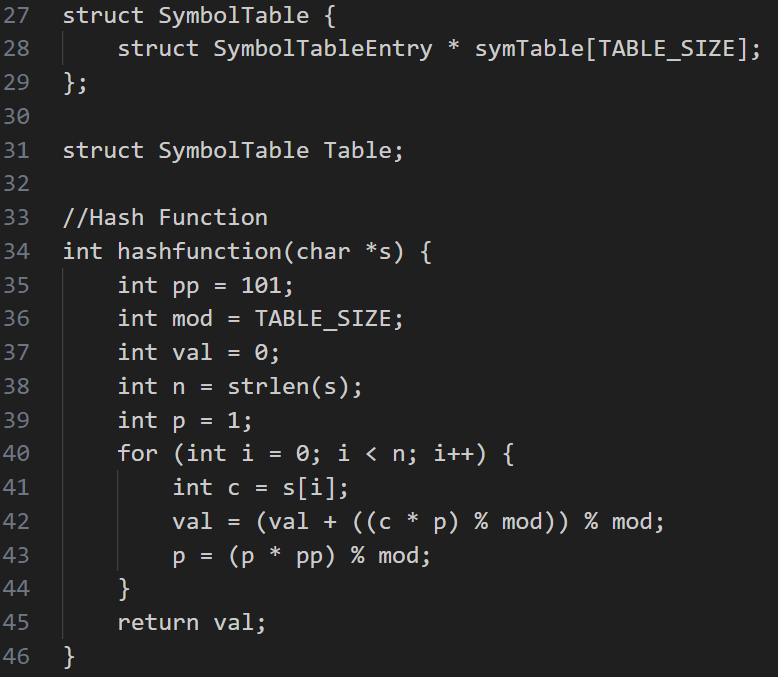
table sizing, tracking line

numbers associated with

symbols, and storing symbol

entries with their names, types,

and sizes.



This is used to calculate a hash

value for a given string s using

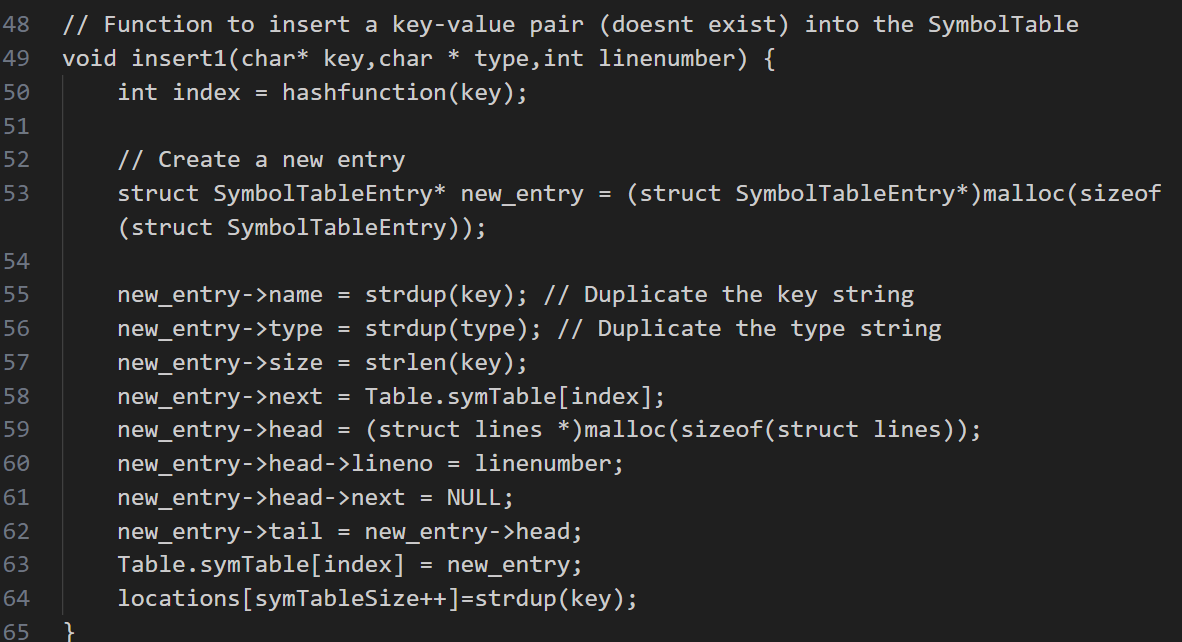
a custom hash function,

ensuring that the hash value

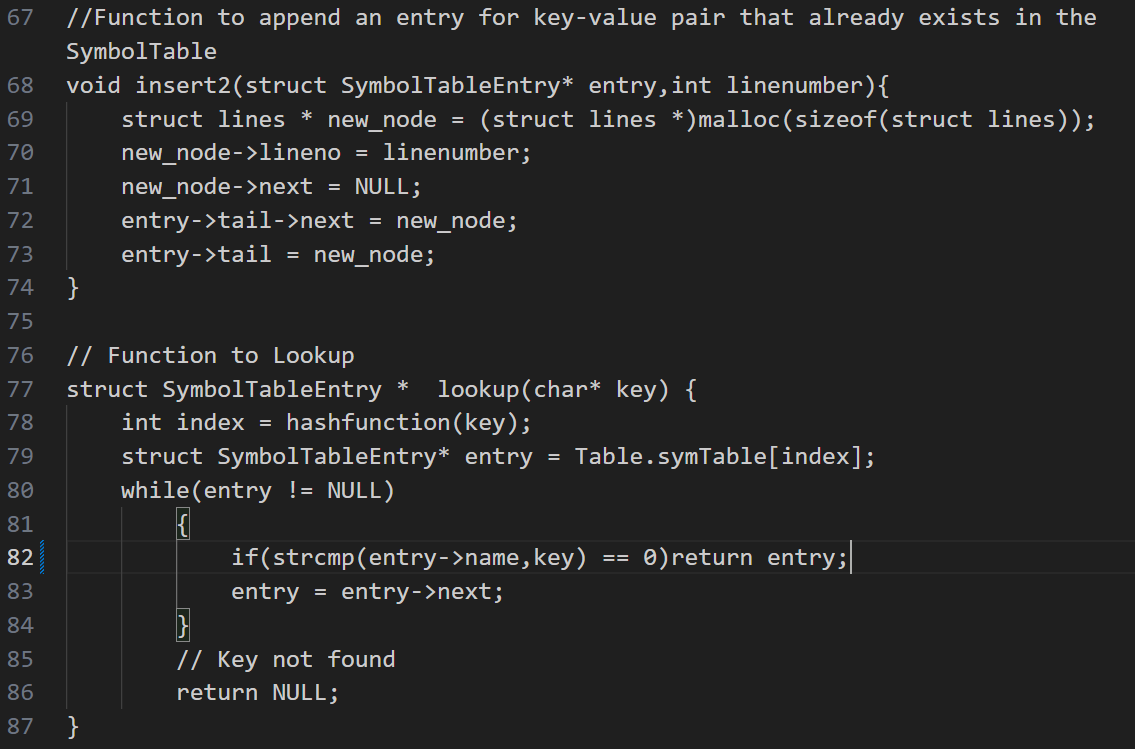
falls within the specified

TABLE\_SIZE for indexing into

a symbol table.

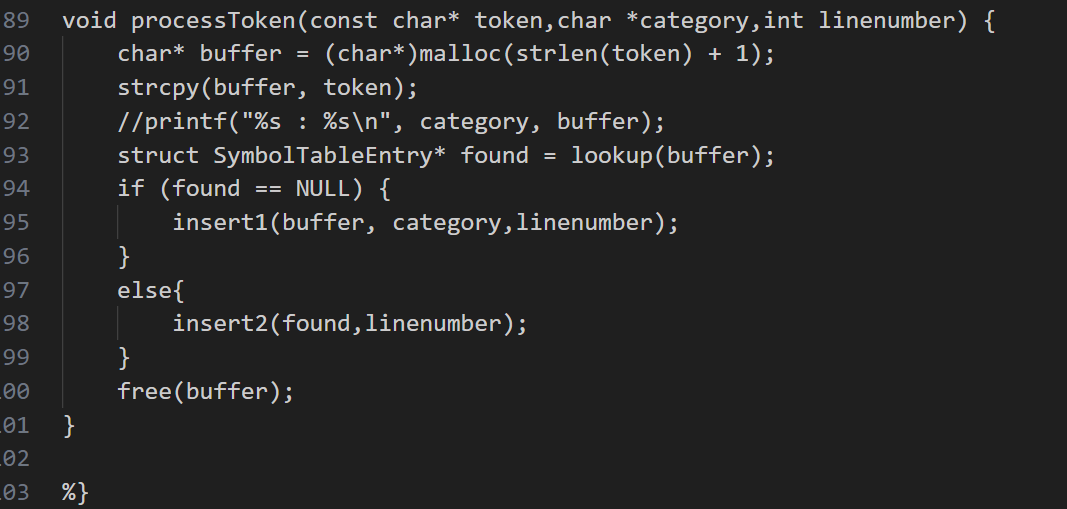


The insert () function inserts a new key-value pair (identifier and its associated type) into the symbol table, creating a new entry with memory allocation and linking it to the hash table, while also recording the line number of the first occurrence of the identifier and adding its name to the locations array for later reference.



In the insert2() function we append a new line number to the list of line numbers associated with an existing key-value pair (identifier) in the symbol table, creating a new node for the line number and updating the linked list accordingly.

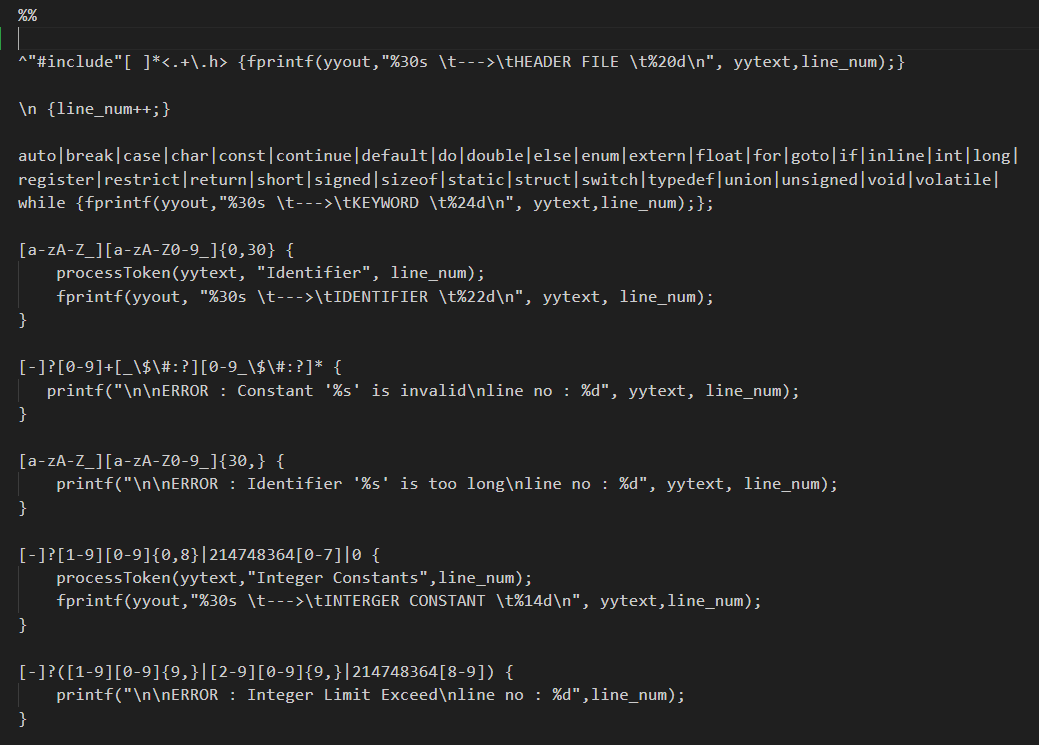
The lookup () function looks up an identifier (specified by the key) in the symbol table by calculating its hash index, traversing the linked list at that index, and returning the corresponding symbol table entry if found; otherwise, it returns NULL to indicate that the identifier is not in the symbol table.

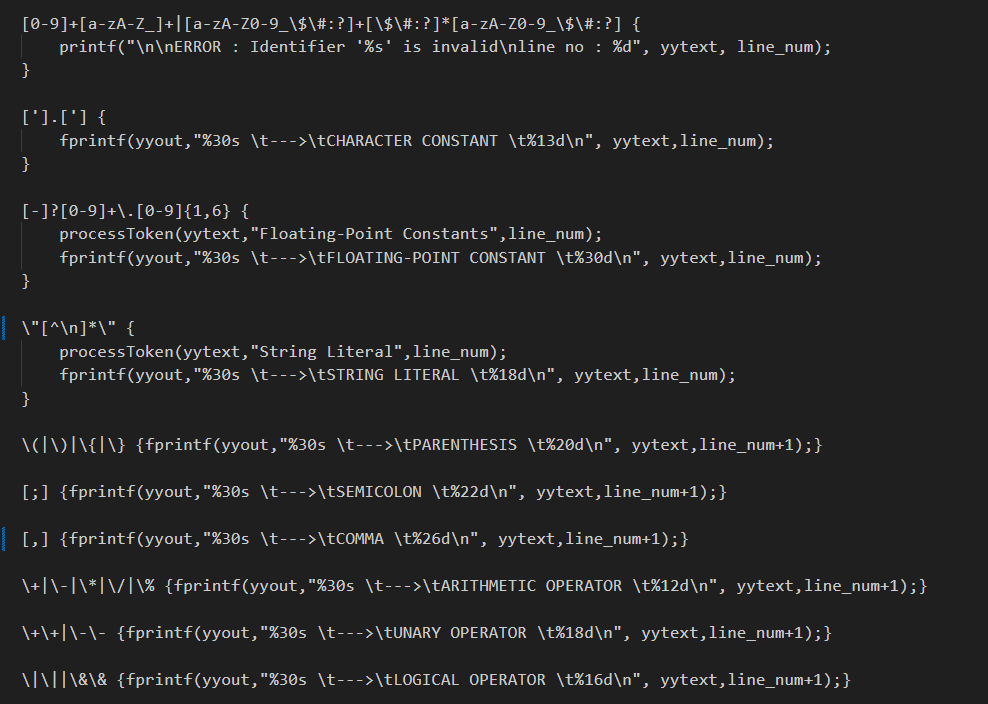


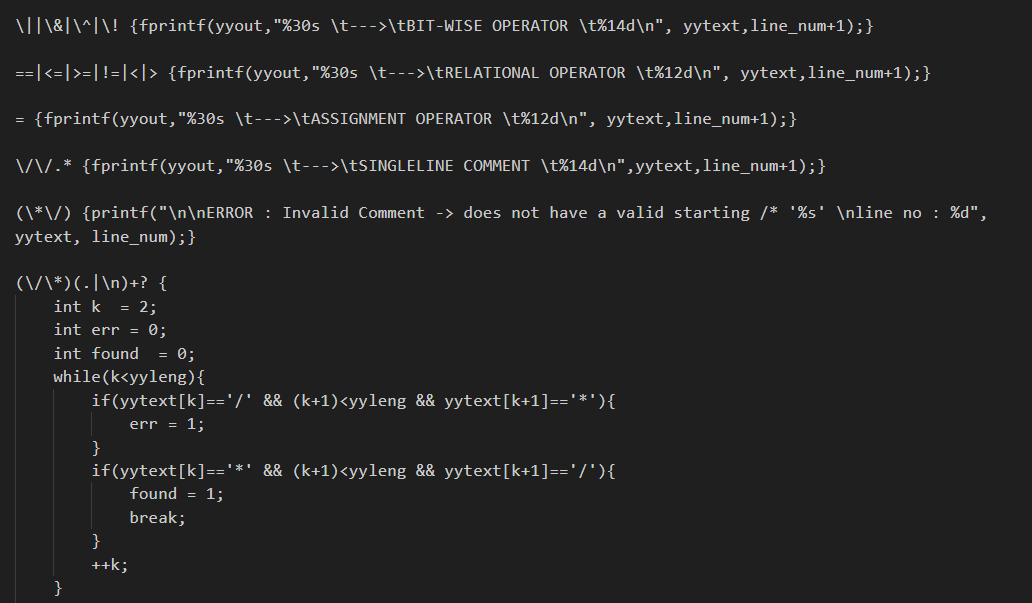
The processToken() function processes a token by making a copy of it, checks if the token (identifier) is already in the symbol table using the lookup function, and either inserts it as a new entry with its category and line number or appends the line number to the existing entry's line numbers list based on whether it's found or not in the symbol table, and then frees the allocated memory for the token copy.

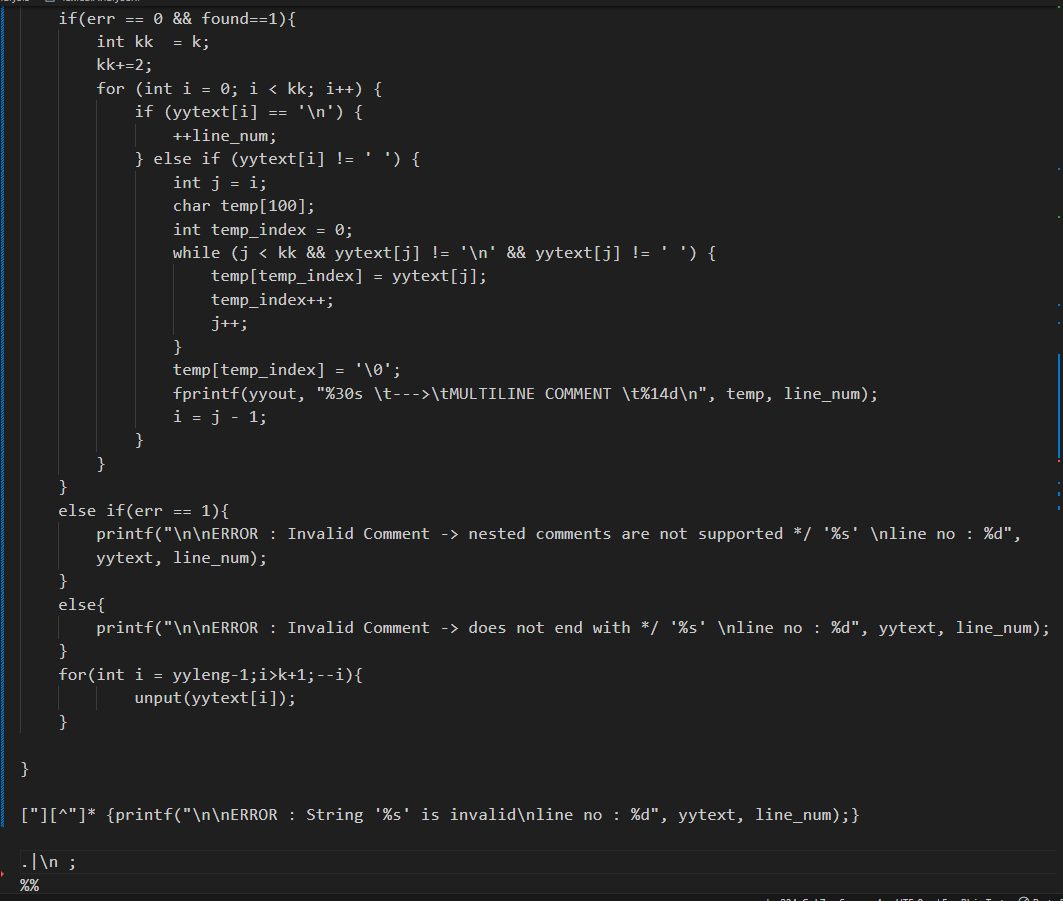
1. **Rules Section**

The rules for identifying identifiers, keywords, header files, numeric constants, string literal, single line comments, multiline comments, brackets, semicolons, commas, and operators (athematic, logical, relational, bit-wise, etc.) are written.  
  
The rules for lexical error identification such as invalid identifier, invalid string literal, invalid numeric literal, invalid multiline comment and nested comments are specified.

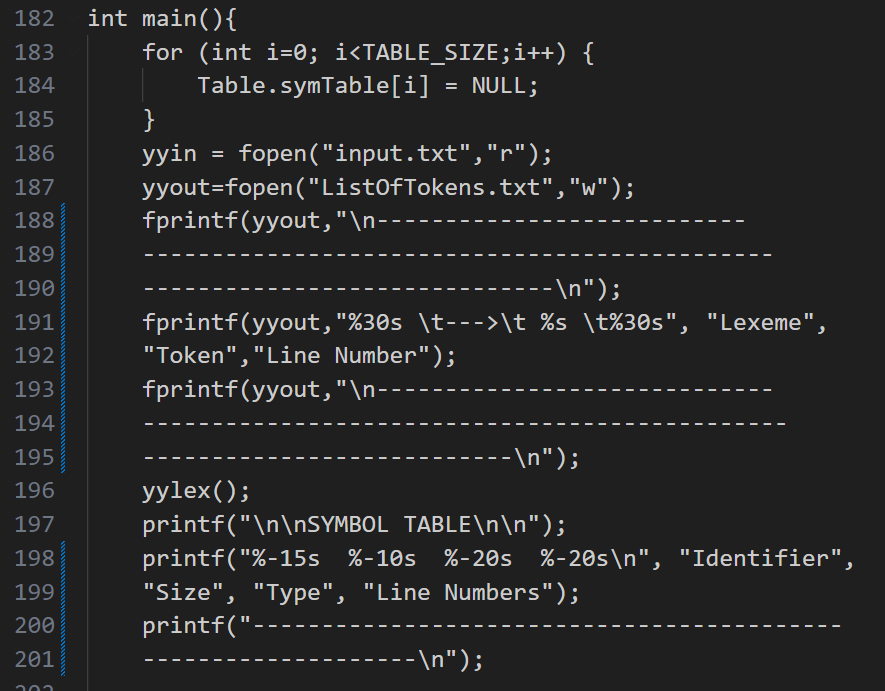
****



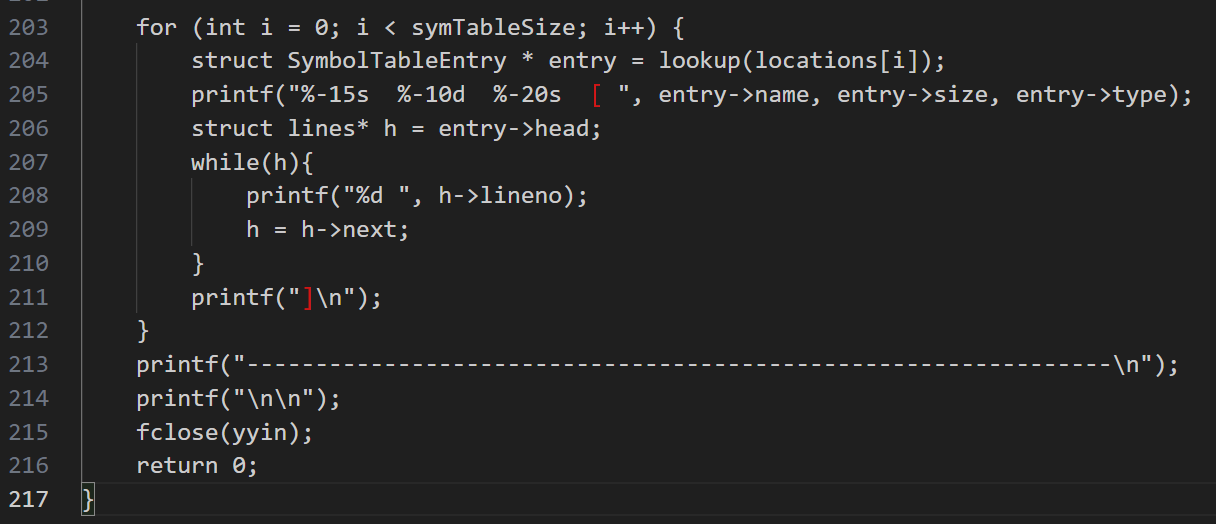




1. **Main Function**



We are initializing the table to NULL values. Here we are just outputting all our tokens into a “List of Tokens “file.

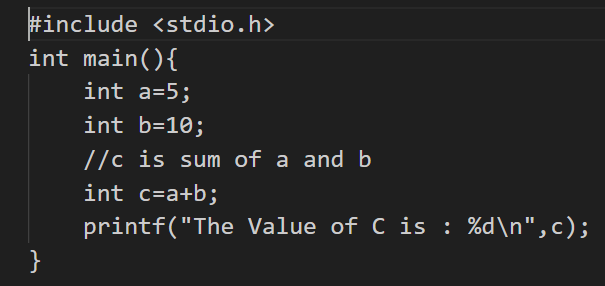


Here for each symbol in our Symbol table, we are printing its name, size, type, and line numbers.

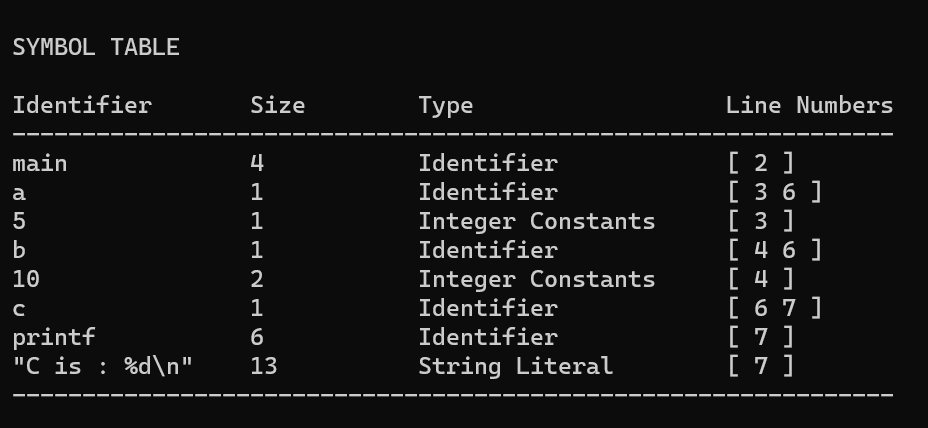
**TEST CASES And their OUTPUTS**

1. **input.txt - test input file 1**

* Input File code



* Symbol Table



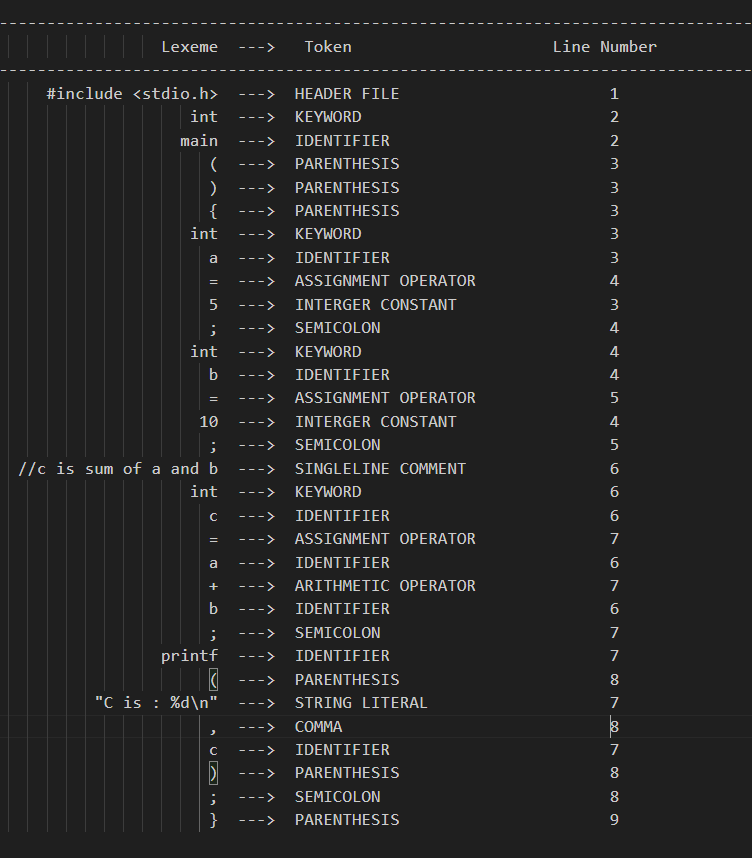
The Symbol Table for the above

input c file. The

code does not have any lexical errors

and the tokens are identified and are outputted into a file.

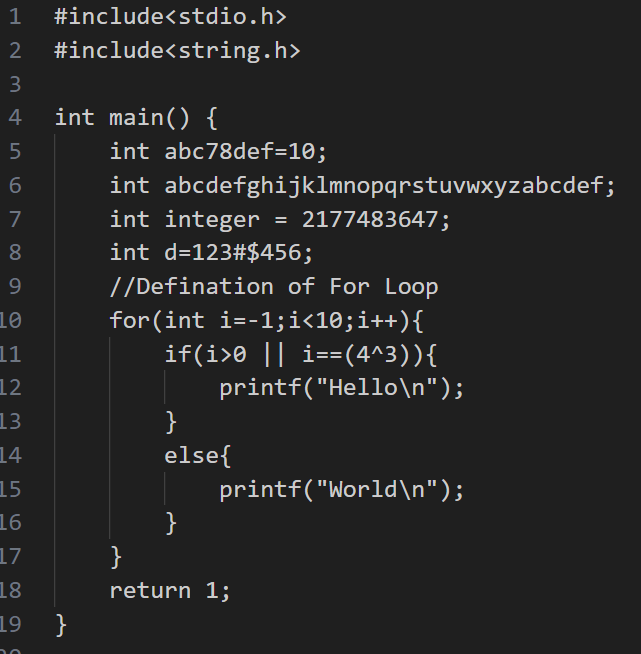
* ListOfToken.txt



This is the output file generated after the lexical analysis phase. All tokens identified are sent to this file and displayed. Every token has the lexeme, its type, and line number of appearance displayed in the output file.

1. **inputTwo.txt - test input file 2**

* Input File code



In this input test case, the variable **‘abcdefghijklmnopqrstuvwxyz**

**abcdef’** has exceeded the Identifier name size limit. This error is a lexical

error and has to be detected in

the lexical analysis phase. Thus

Our scanner identifies the error

and displays it along with the line

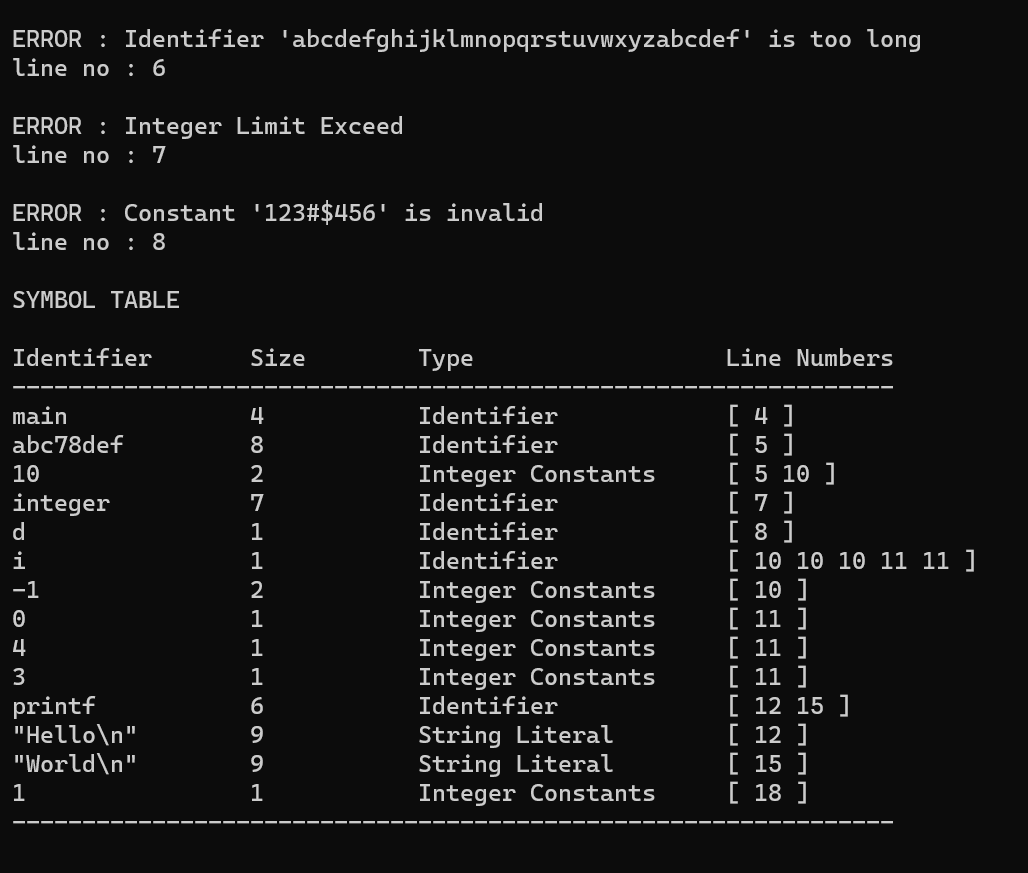
no. Similarly, the variable **‘integer’**

has exceeded the integer limit so

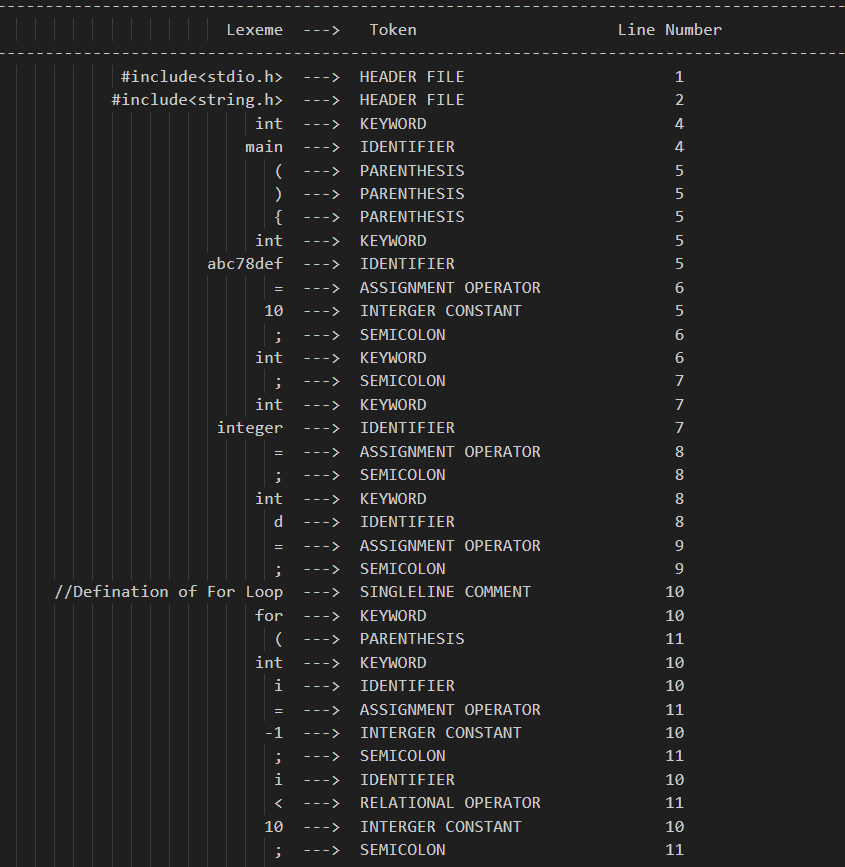
it is also detected by a scanner.

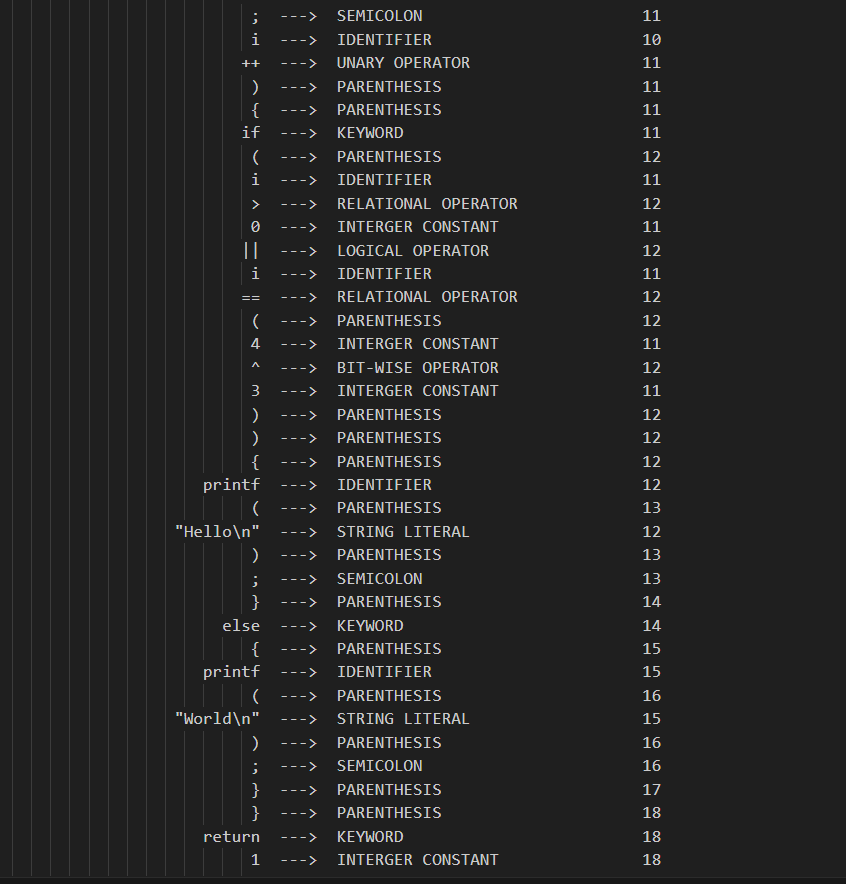
The variable **‘d’** is assigned with a constant value that is an invalid numeric constant. So, our scanner identifies these lexical errors and displays them.

* Symbol Table



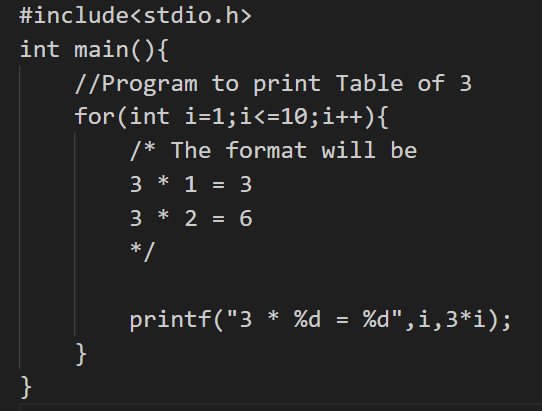
* ListOfSymbol.txt





1. **inputThree.txt - test input file 3**

* Input File code

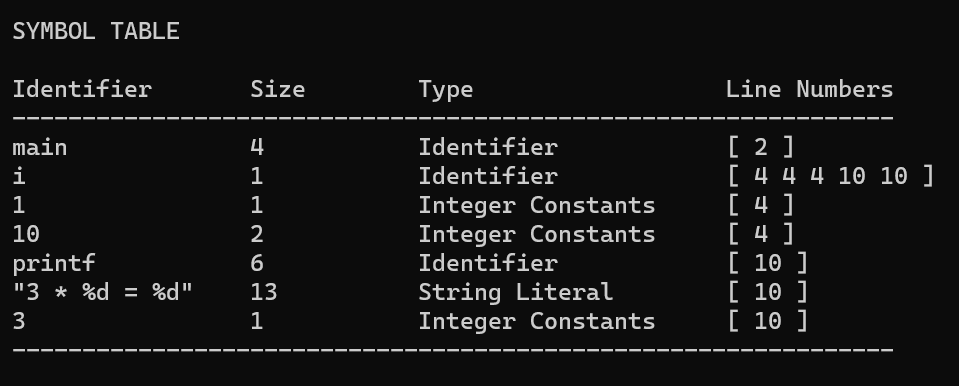


This test case is to show that our scanner is capable of identifying

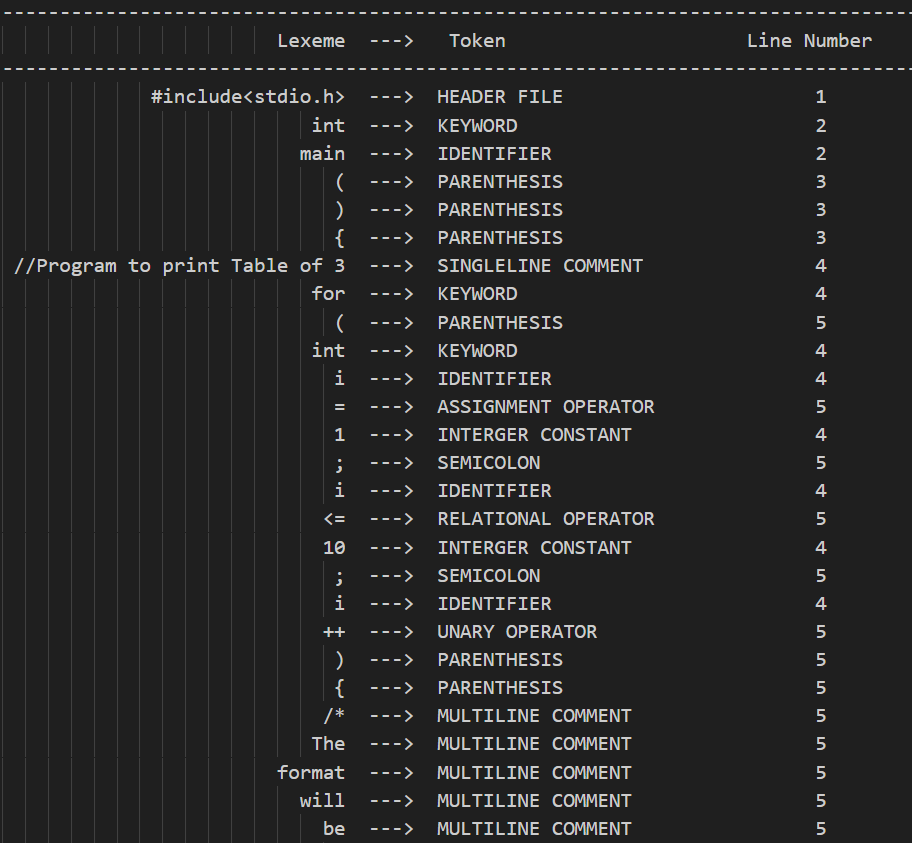
the single line and, multiline comments in C. After the identification of comments, they

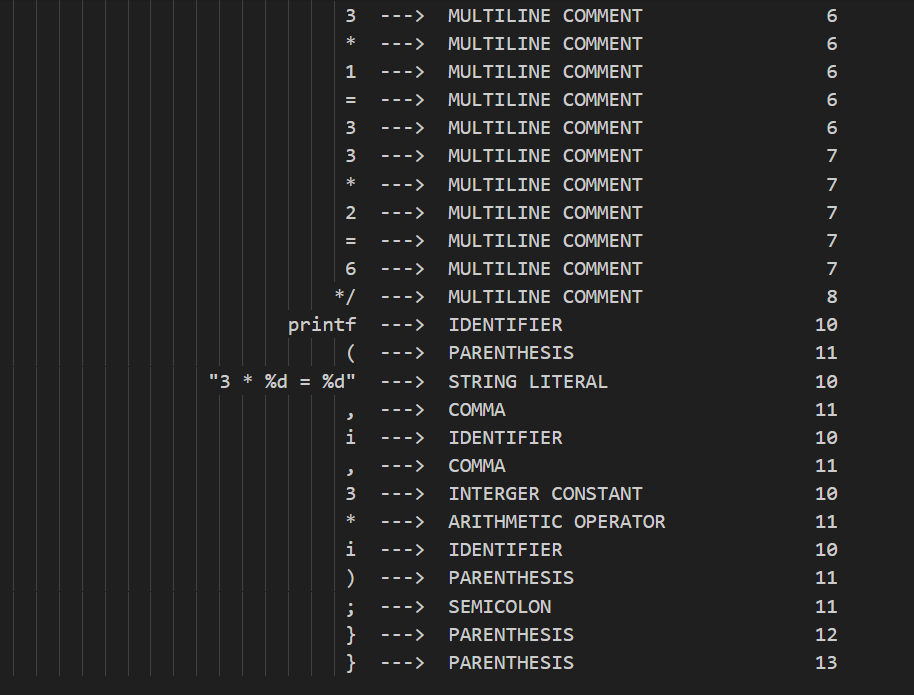
are sent to the ListOfTokens.txt   
file which maintains a list of all the tokens.

* Symbol Table



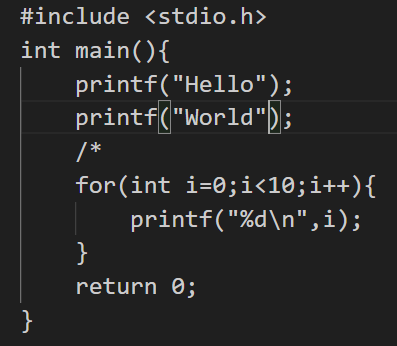
* ListOfTokens.txt





1. **inputFour.txt - test input file 4**

* Input File code



This test case has an error related to multiline comments. The ‘/\*’ indicates the beginning of a multiline comment but

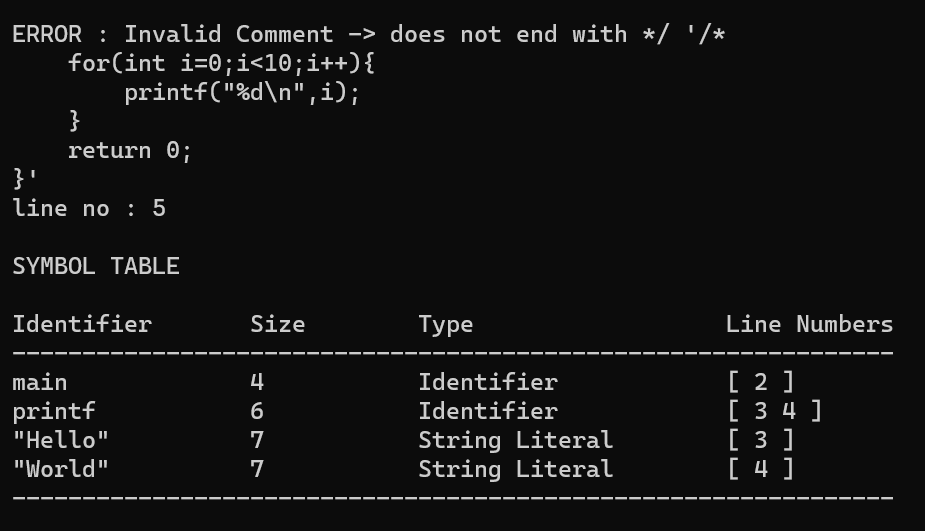
there is no ‘\*/’ to end it. This Lexical error

is detected by our scanner and the Error

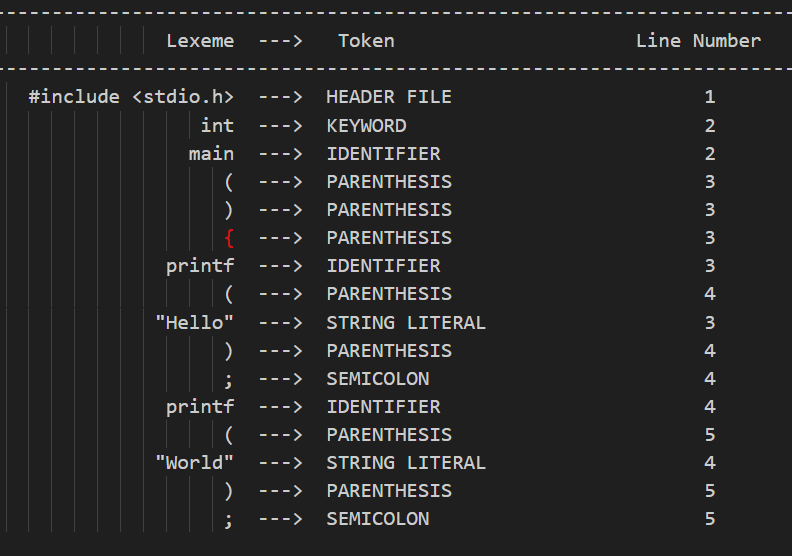
is displayed along with the line number.

All the other tokens that are inside the multiline comments are not added to the ListOftoken.txt file.

* Symbol Table

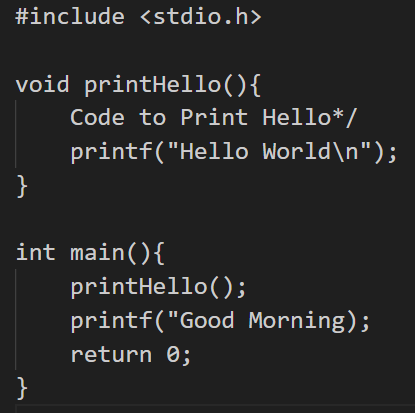


* ListOfToken.txt



1. **inputFive.txt - test input file 5**

* Input file code



This test case has 2 lexical errors - an invalid string as it doesn’t have a closing quote and an invalid comment as there

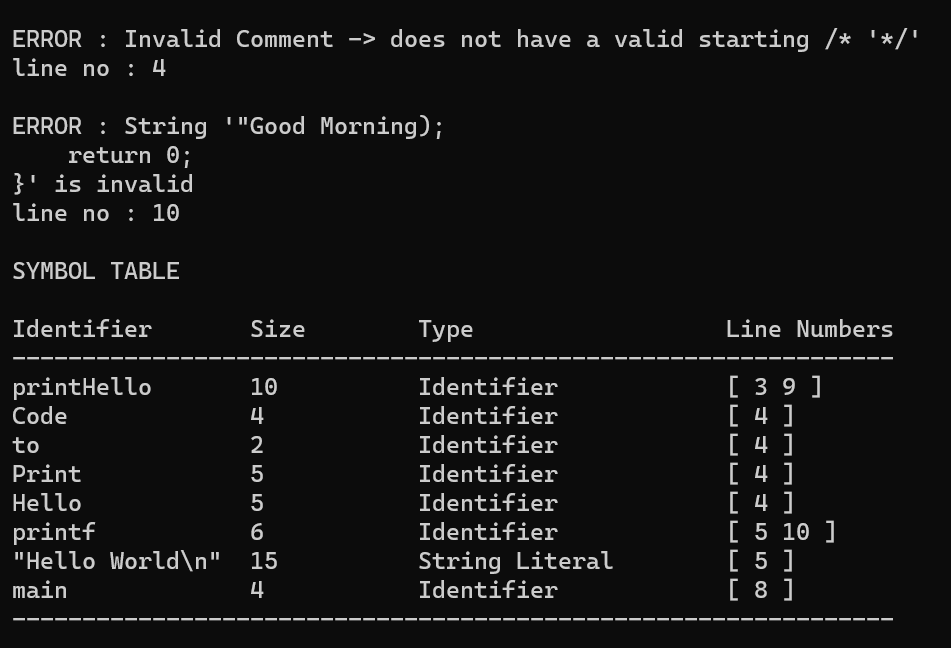
is no ‘/\*’ to indicate the beginning of

the multiline comment. This is identified

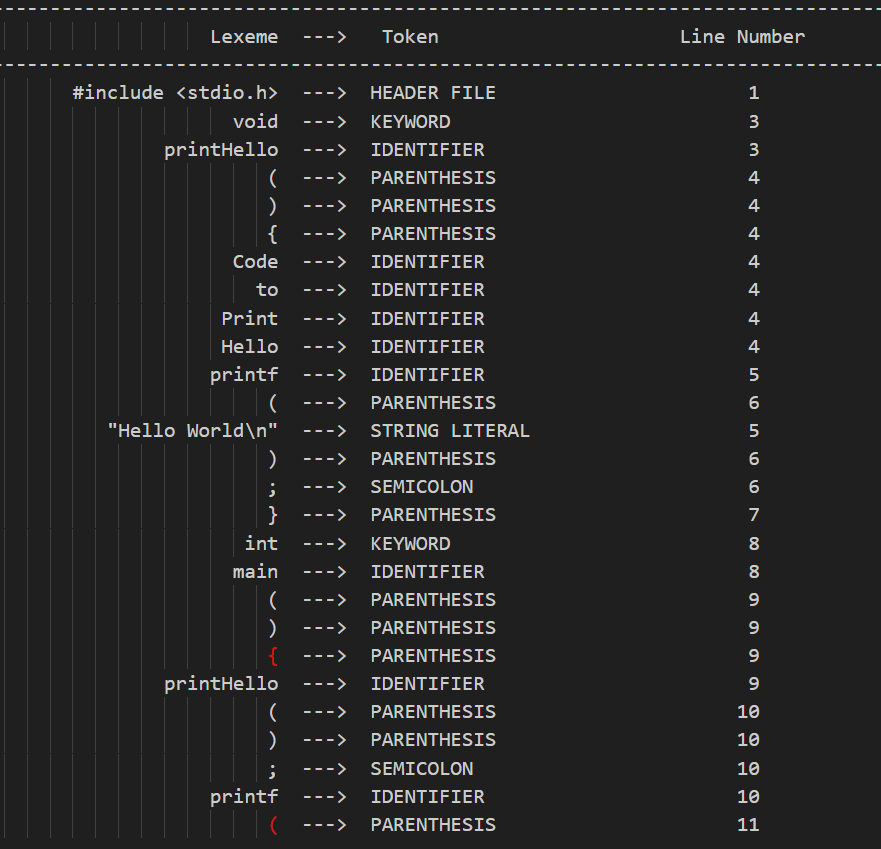
by our scanner. And displayed as an

error along with the line no of error.

* Symbol Table

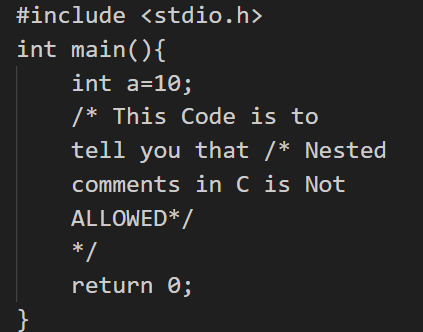


* ListOfTokens.txt



1. **inputSix.txt - test input file 6**

* input file code

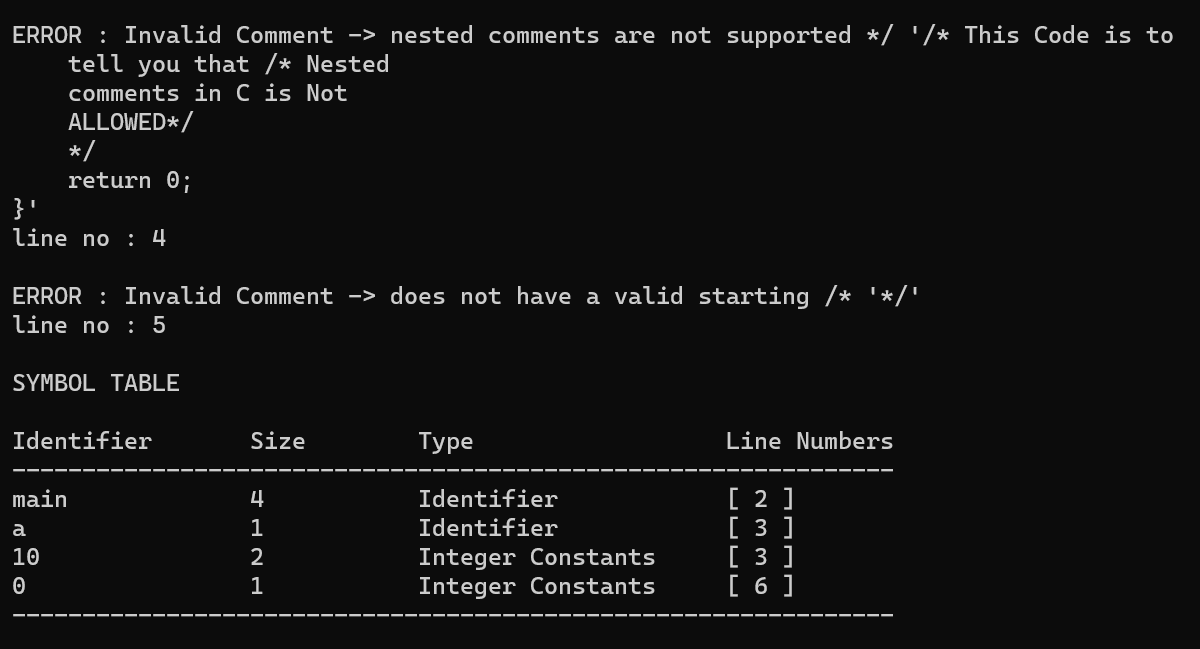


This test case has a nested multiline comment. In C multiline comment can’t

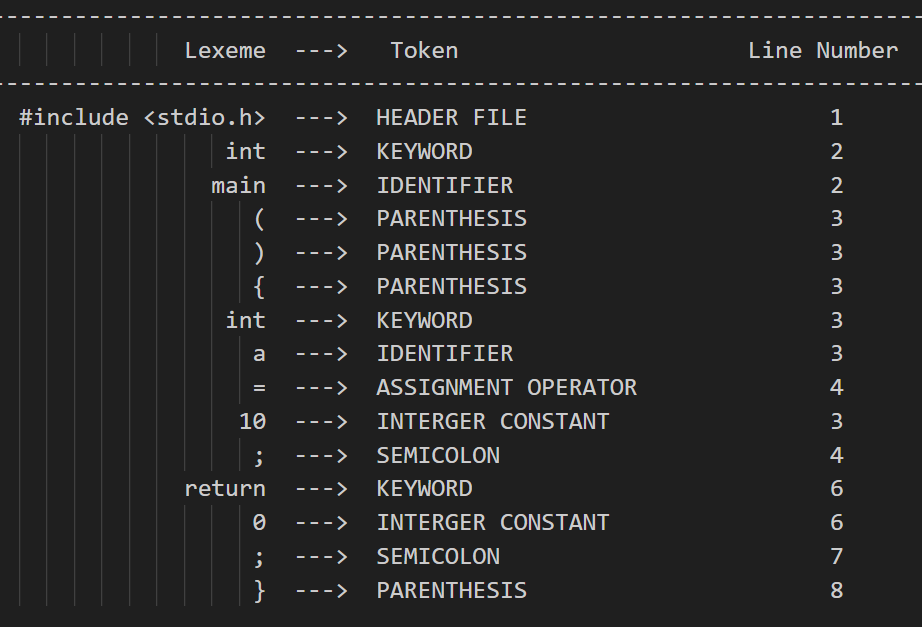
be nested and it comes under the

category of Lexical errors. This error is identified and displayed by scanner.

* Symbol Table



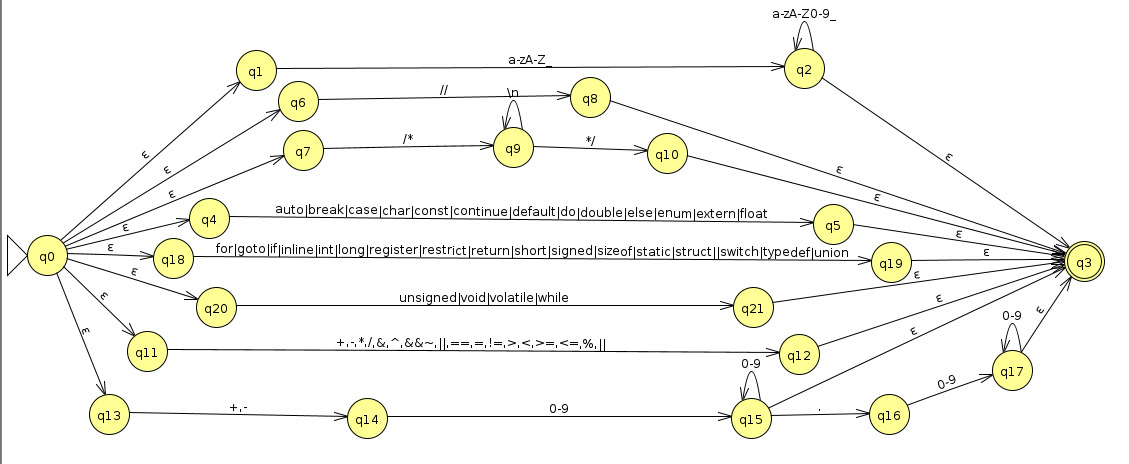
* ListOfTokens.txt

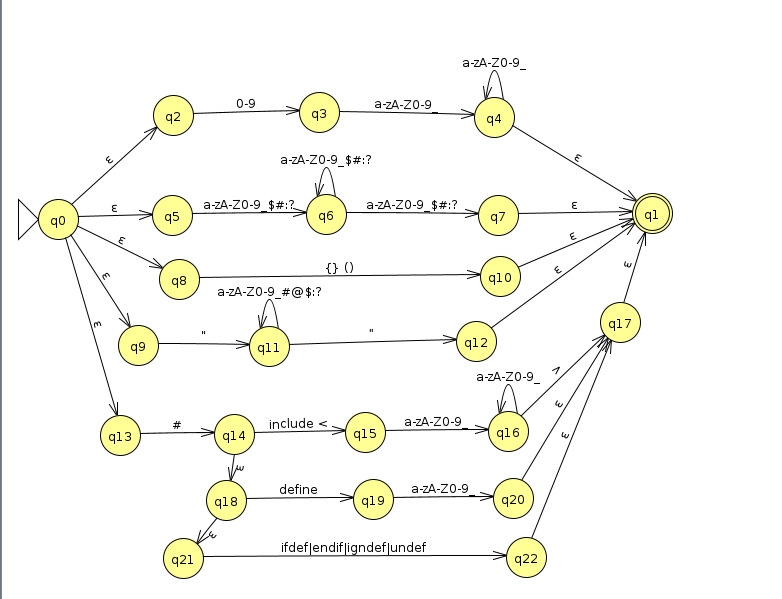


**List of rules not implemented**

* Lexical error regarding the number and correctness of opening and closing flower braces.
* Error handling for floating-point numbers and characters.

**Pictures of DFA underlying the scanner**





**CODE**

%{

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define TABLE\_SIZE 100003

#define MAX\_SYMBOLS 1000000

int line\_num = 1;

char \* locations[MAX\_SYMBOLS];

int symTableSize=0;

struct lines{

int lineno; // stores the line-number

struct lines\*next;

};

struct SymbolTableEntry {

char \* name; //identifier

char \* type;

int size;

struct SymbolTableEntry \* next;

struct lines \* head; //head of the linenumbers list

struct lines \* tail; //tail of the linenumbers list

};

struct SymbolTable {

struct SymbolTableEntry \* symTable[TABLE\_SIZE];

};

struct SymbolTable Table;

//Hash Function

int hashfunction(char \*s) {

int pp = 101;

int mod = TABLE\_SIZE;

int val = 0;

int n = strlen(s);

int p = 1;

for (int i = 0; i < n; i++) {

int c = s[i];

val = (val + ((c \* p) % mod)) % mod;

p = (p \* pp) % mod;

}

return val;

}

// Function to insert a key-value pair (doesnt exist) into the SymbolTable

void insert1(char\* key,char \* type,int linenumber) {

int index = hashfunction(key);

// Create a new entry

struct SymbolTableEntry\* new\_entry = (struct SymbolTableEntry\*)malloc(sizeof(struct SymbolTableEntry));

new\_entry->name = strdup(key); // Duplicate the key string

new\_entry->type = strdup(type); // Duplicate the type string

new\_entry->size = strlen(key);

new\_entry->next = Table.symTable[index];

new\_entry->head = (struct lines \*)malloc(sizeof(struct lines));

new\_entry->head->lineno = linenumber;

new\_entry->head->next = NULL;

new\_entry->tail = new\_entry->head;

Table.symTable[index] = new\_entry;

locations[symTableSize++]=strdup(key);

}

//Function to append an entry for key-value pair that already exists in the SymbolTable

void insert2(struct SymbolTableEntry\* entry,int linenumber){

struct lines \* new\_node = (struct lines \*)malloc(sizeof(struct lines));

new\_node->lineno = linenumber;

new\_node->next = NULL;

entry->tail->next = new\_node;

entry->tail = new\_node;

}

// Function to Lookup

struct SymbolTableEntry \* lookup(char\* key) {

int index = hashfunction(key);

struct SymbolTableEntry\* entry = Table.symTable[index];

while(entry != NULL)

{

if(strcmp(entry->name,key) == 0)return entry;

entry = entry->next;

}

// Key not found

return NULL;

}

void processToken(const char\* token,char \*category,int linenumber) {

char\* buffer = (char\*)malloc(strlen(token) + 1);

strcpy(buffer, token);

//printf("%s : %s\n", category, buffer);

struct SymbolTableEntry\* found = lookup(buffer);

if (found == NULL) {

insert1(buffer, category,linenumber);

}

else{

insert2(found,linenumber);

}

free(buffer);

}

%}

%%

^"#include"[ ]\*<.+\.h> {fprintf(yyout,"%30s \t--->\tHEADER FILE \t%20d\n", yytext,line\_num);}

\n {line\_num++;}

auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|inline|int|long|register|restrict|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while {fprintf(yyout,"%30s \t--->\tKEYWORD \t%24d\n", yytext,line\_num);};

[a-zA-Z\_][a-zA-Z0-9\_]{0,30} {

processToken(yytext, "Identifier", line\_num);

fprintf(yyout, "%30s \t--->\tIDENTIFIER \t%22d\n", yytext, line\_num);

}

[-]?[0-9]+[\_\$\#:?][0-9\_\$\#:?]\* {

printf("\n\nERROR : Constant '%s' is invalid\nline no : %d", yytext, line\_num);

}

[a-zA-Z\_][a-zA-Z0-9\_]{30,} {

printf("\n\nERROR : Identifier '%s' is too long\nline no : %d", yytext, line\_num);

}

[-]?[1-9][0-9]{0,8}|214748364[0-7]|0 {

processToken(yytext,"Integer Constants",line\_num);

fprintf(yyout,"%30s \t--->\tINTERGER CONSTANT \t%14d\n", yytext,line\_num);

}

[-]?([1-9][0-9]{9,}|[2-9][0-9]{9,}|214748364[8-9]) {

printf("\n\nERROR : Integer Limit Exceed\nline no : %d",line\_num);

}

[0-9]+[a-zA-Z\_]+|[a-zA-Z0-9\_\$\#:?]+[\$\#:?]\*[a-zA-Z0-9\_\$\#:?] {

printf("\n\nERROR : Identifier '%s' is invalid\nline no : %d", yytext, line\_num);

}

['].['] {

fprintf(yyout,"%30s \t--->\tCHARACTER CONSTANT \t%13d\n", yytext,line\_num);

}

[-]?[0-9]+\.[0-9]{1,6} {

processToken(yytext,"Floating-Point Constants",line\_num);

fprintf(yyout,"%30s \t--->\tFLOATING-POINT CONSTANT \t%30d\n", yytext,line\_num);

}

\"[^\n]\*\" {

processToken(yytext,"String Literal",line\_num);

fprintf(yyout,"%30s \t--->\tSTRING LITERAL \t%18d\n", yytext,line\_num);

}

\(|\)|\{|\} {fprintf(yyout,"%30s \t--->\tPARENTHESIS \t%20d\n", yytext,line\_num+1);}

[;] {fprintf(yyout,"%30s \t--->\tSEMICOLON \t%22d\n", yytext,line\_num+1);}

[,] {fprintf(yyout,"%30s \t--->\tCOMMA \t%26d\n", yytext,line\_num+1);}

\+|\-|\\*|\/|\% {fprintf(yyout,"%30s \t--->\tARITHMETIC OPERATOR \t%12d\n", yytext,line\_num+1);}

\+\+|\-\- {fprintf(yyout,"%30s \t--->\tUNARY OPERATOR \t%18d\n", yytext,line\_num+1);}

\|\||\&\& {fprintf(yyout,"%30s \t--->\tLOGICAL OPERATOR \t%16d\n", yytext,line\_num+1);}

\||\&|\^|\! {fprintf(yyout,"%30s \t--->\tBIT-WISE OPERATOR \t%14d\n", yytext,line\_num+1);}

==|<=|>=|!=|<|> {fprintf(yyout,"%30s \t--->\tRELATIONAL OPERATOR \t%12d\n", yytext,line\_num+1);}

= {fprintf(yyout,"%30s \t--->\tASSIGNMENT OPERATOR \t%12d\n", yytext,line\_num+1);}

\/\/.\* {fprintf(yyout,"%30s \t--->\tSINGLELINE COMMENT \t%14d\n",yytext,line\_num+1);}

(\\*\/) {printf("\n\nERROR : Invalid Comment -> does not have a valid starting /\* '%s' \nline no : %d", yytext, line\_num);}

(\/\\*)(.|\n)+? {

int k = 2;

int err = 0;

int found = 0;

while(k<yyleng){

if(yytext[k]=='/' && (k+1)<yyleng && yytext[k+1]=='\*'){

err = 1;

}

if(yytext[k]=='\*' && (k+1)<yyleng && yytext[k+1]=='/'){

found = 1;

break;

}

++k;

}

if(err == 0 && found==1){

int kk = k;

kk+=2;

for (int i = 0; i < kk; i++) {

if (yytext[i] == '\n') {

++line\_num;

} else if (yytext[i] != ' ') {

int j = i;

char temp[100];

int temp\_index = 0;

while (j < kk && yytext[j] != '\n' && yytext[j] != ' ') {

temp[temp\_index] = yytext[j];

temp\_index++;

j++;

}

temp[temp\_index] = '\0';

fprintf(yyout, "%30s \t--->\tMULTILINE COMMENT \t%14d\n", temp, line\_num);

i = j - 1;

}

}

}

else if(err == 1){

printf("\n\nERROR : Invalid Comment -> nested comments are not supported \*/ '%s' \nline no : %d", yytext, line\_num);

}

else{

printf("\n\nERROR : Invalid Comment -> does not end with \*/ '%s' \nline no : %d", yytext, line\_num);

}

for(int i = yyleng-1;i>k+1;--i){

unput(yytext[i]);

}

}

["][^"]\* {printf("\n\nERROR : String '%s' is invalid\nline no : %d", yytext, line\_num);}

.|\n ;

%%

int yywrap(){

return 1;

}

int main(){

for (int i=0; i<TABLE\_SIZE;i++) {

Table.symTable[i] = NULL;

}

yyin = fopen("inputSix.txt","r");

yyout=fopen("ListOfTokens.txt","w");

fprintf(yyout,"\n-------------------------------------------------------------------------------------------------------\n");

fprintf(yyout,"%30s \t--->\t %s \t%30s", "Lexeme",

"Token","Line Number");

fprintf(yyout,"\n-------------------------------------------------------------------------------------------------------\n");

yylex();

printf("\n\nSYMBOL TABLE\n\n");

printf("%-15s %-10s %-20s %-20s\n", "Identifier",

"Size", "Type", "Line Numbers");

printf("---------------------------------------------------------------\n");

for (int i = 0; i < symTableSize; i++) {

struct SymbolTableEntry \* entry = lookup(locations[i]);

printf("%-15s %-10d %-20s [ ", entry->name, entry->size, entry->type);

struct lines\* h = entry->head;

while(h){

printf("%d ", h->lineno);

h = h->next;

}

printf("]\n");

}

printf("---------------------------------------------------------------\n");

printf("\n\n");

fclose(yyin);

return 0;

}

**Conclusion**

In conclusion, the development of a comprehensive lexical analyzer for the C language using LEX/Flex has been a substantial undertaking that has yielded a powerful tool for parsing and understanding C code. Through meticulous study of the language's grammar and specifications, we successfully identified and categorized the essential lexical components, including keywords, identifiers, constants, string literals, and comments. Our implementation goes beyond mere token recognition by addressing more complex language features, such as nested comments, and ensuring that the generated error messages are as informative as possible, aiding developers in pinpointing and rectifying issues within their code. Additionally, the incorporation of a hash-based symbol table enhances the analyzer's potential for broader use in compiler construction. This project underscores the significance of precise language analysis, effective tool usage, and robust error handling in the development of a sophisticated C compiler or similar software systems, marking a significant milestone in the realm of programming language processing.

**References**

* <https://www.geeksforgeeks.org/working-of-lexical-analyzer-in-compiler/>
* <https://www.geeksforgeeks.org/lexical-analysis-and-syntax-analysis/>
* <https://www.jflap.org/>
* Compiler Principles, Techniques and Tools by Alfred V Aho, Monica S Lam, Ravi Sethi, Jefferey D. Ullman