Lexical Analyzer for the C language

National Institute of Technology Karnataka Surathkal

Date: 25-08-2020

Submitted To: Dr P. Santhi Thilagam

Members:
Irfan Backer Iqbal Valappil - 181CO122
Sangeeth S V - 181CO246
Shreesha Bharadwaj - 181CO249
Vignesh Srinivasan - 181CO258

Abstract:

The objective of this project is to build a compiler for the C language. It will have the following features:

- Support the following cases
  - Keywords - int, char, float, main, const, extern, static
  - Identifiers - sum, result, product,
  - Constants - 0, 1, “hello”, 42738
  - Operators - +, -, *, /, ^
  - Special symbols - (,{,[,),},]
- Support various data types mentioned in the C language
- Support for single-line and multi-line comments
- Maintenance of symbol table for each token
- Error handling for some types of errors - invalid preprocessor statements, mismatched brackets, invalid strings

Contents:

List of Figures and Tables:

Introduction:

The lexical analyser is the first step of a compiler. The input to the lexical analyser is the preprocessed source code (without unnecessary whitespaces and comments) which is in one single line. The source code is parsed left to right from top to bottom. This stream of characters is parsed and is converted into lexemes.

Lexemes are a sequence of characters in a token. Lexemes can be broken down into tokens using the rules of a grammar. The rules dictate what can be a pattern which is in the form of regular expressions. The lexemes are passed through a deterministic finite automata (DFA) and are converted into their respective tokens.

At the end of this stage, we get tokens as the output in the form

<token-name, type-of-token>

Examples: <a, identifier> , <int, keyword> , <”hello”, constant>

The lexical analyzer returns an error message in case of an invalid token. THe generated tokens are passed on to the syntax analyzer.

Flex:

Fast lexical analyser generator (FLEX) is a tool for generating lexical analyzers. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. The yacc files are divided into 3 sections: definition section, rules section and code section. Two sections are separated from each other by %%

Definition section

%%

Rules section

%%

Code section

Definition section: This section contains the macros and the header file imports (basically, the preprocessor part of the C program).
Rule section: This section defines the rules (regular expressions) that the lexemes have to follow to be matched with a token. It matches the word against the pattern to see if it matches.
Code section: This contains the C statements and functions. This contains code called by the rules section.

Input to the lexical analyzer:

The input file to the lexical analyzer is a C program. After it is fed to the flex program, a flex file is generated following all the rules mentioned in the rules section. Finally, a file called lex.yy.c is created. When this file is run, it creates a list of tokens recognised in the input C program.

Deterministic finite automata for few regular expressions:

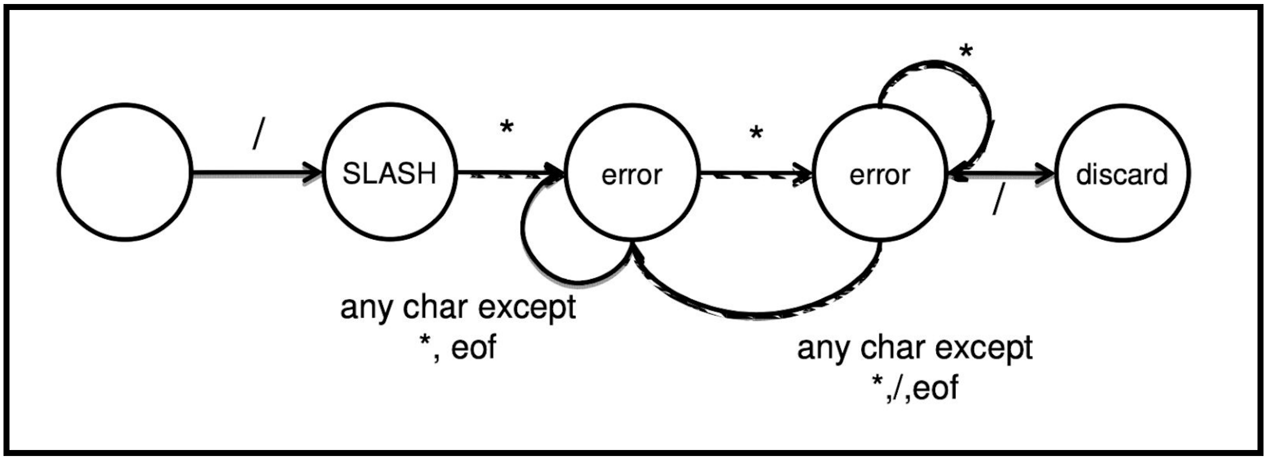


Figure1 : DFA for C-style comments

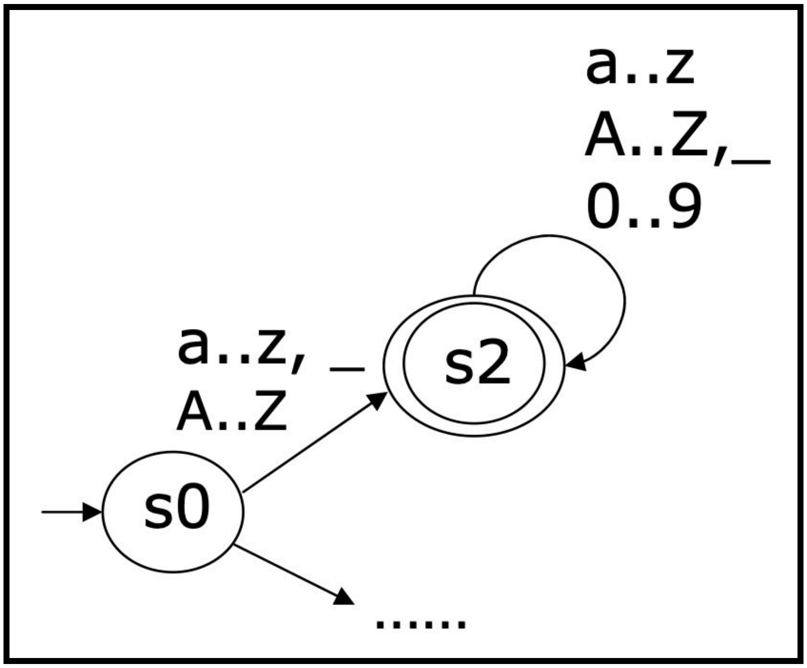
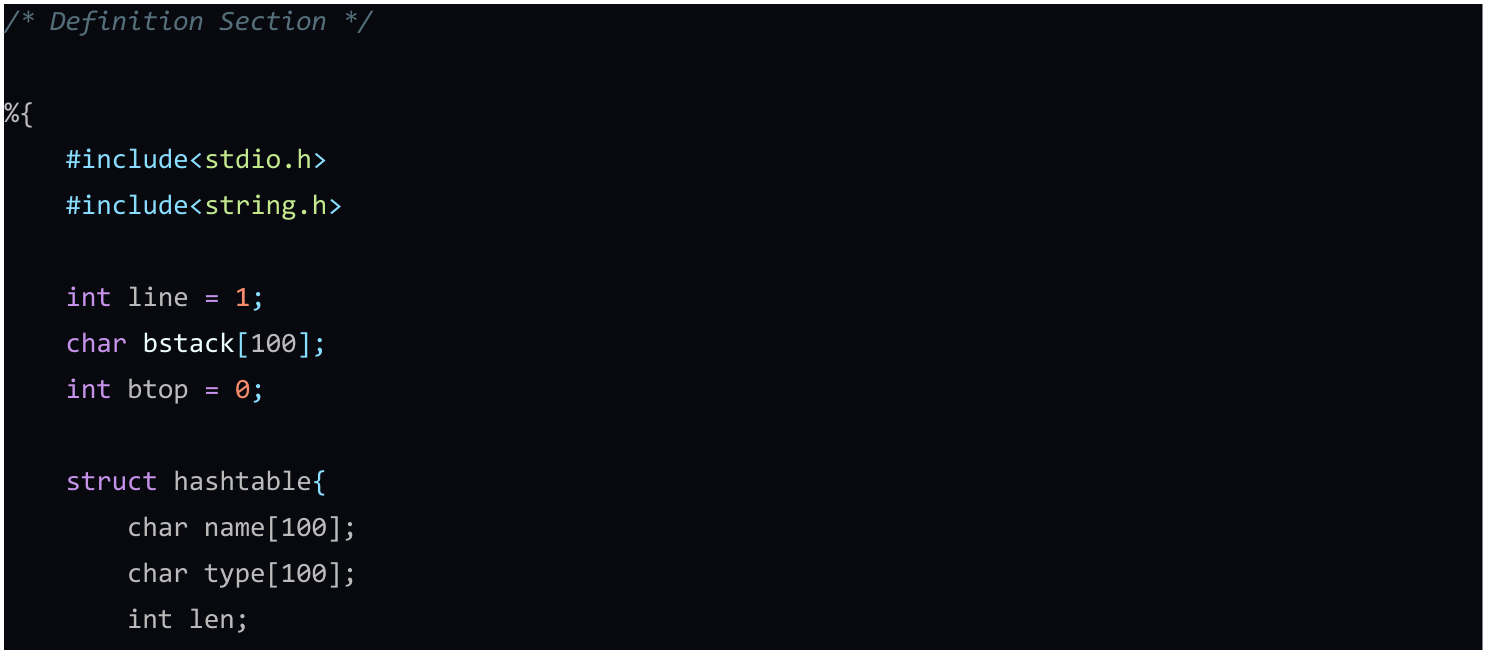
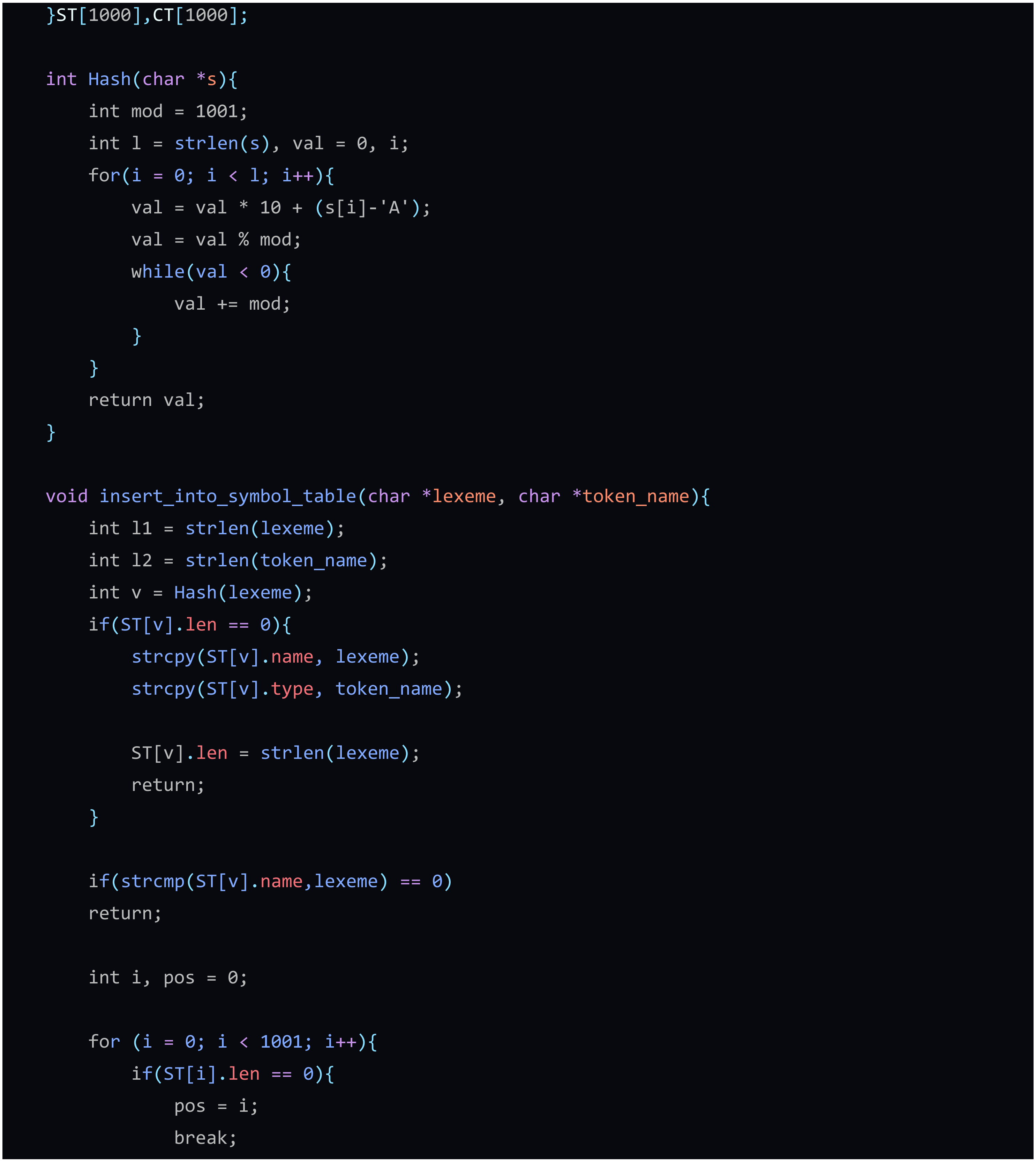


Figure 2 : DFA for identifiers

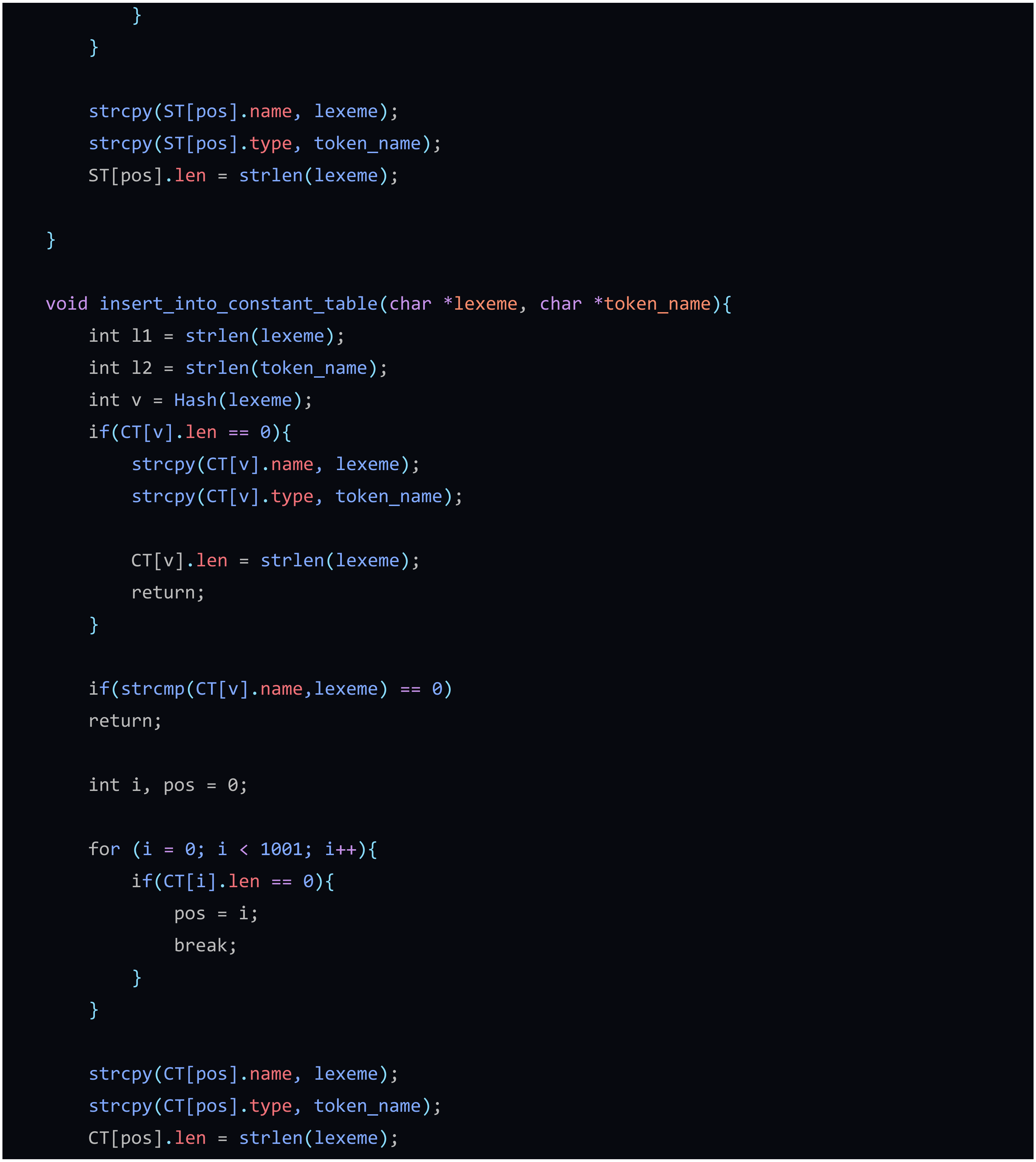
Lexical Analyzer - Code



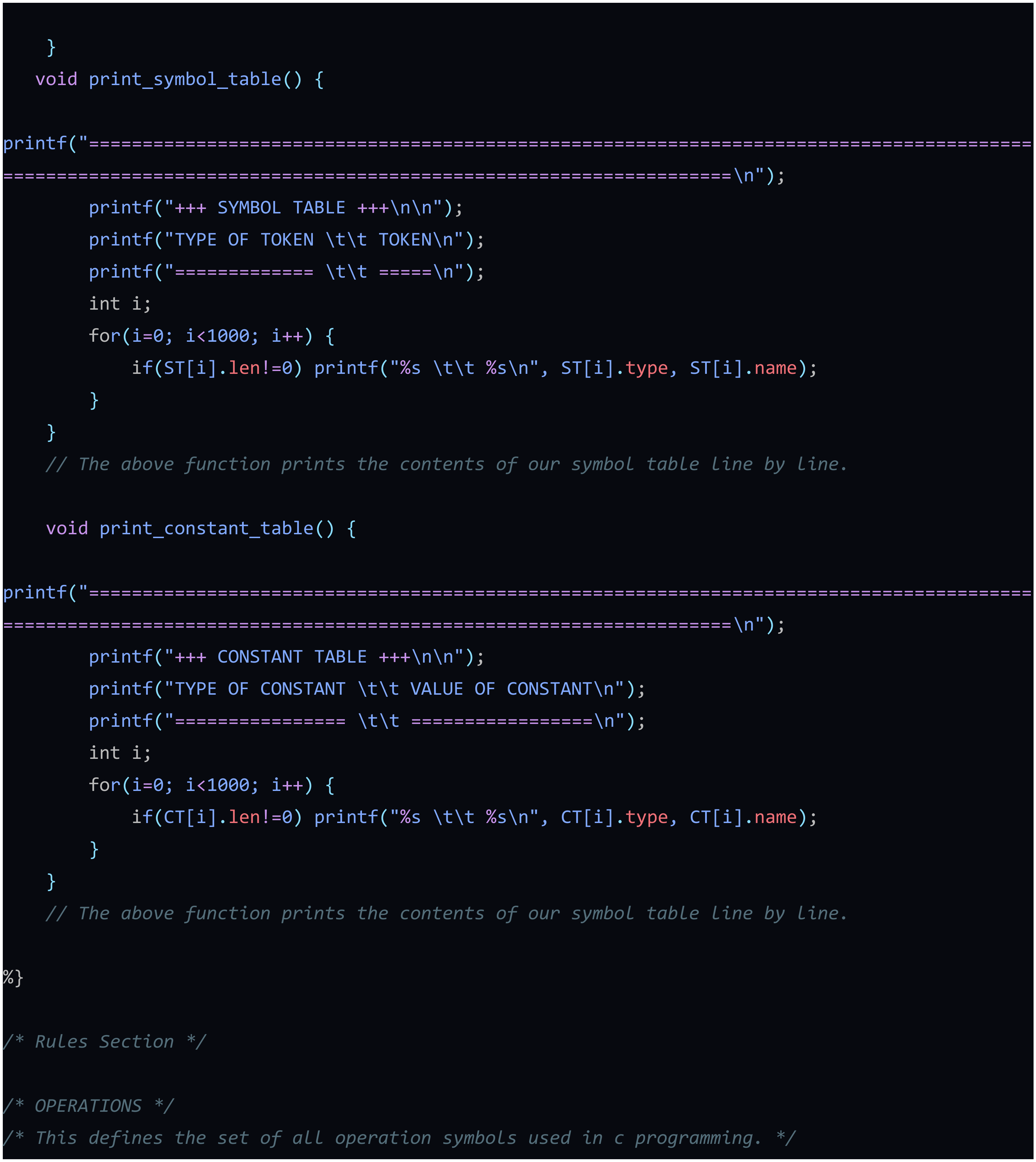
```
/* Definition Section */

%{
    #include<stdio.h>
    #include<string.h>

    int line = 1;
    char bstack[100];
    int btop = 0;

    struct hashtable{
        char name[100];
        char type[100];
        int len;
```



```
}ST[1000],CT[1000];

int Hash(char *s){
    int mod = 1001;
    int l = strlen(s), val = 0, i;
    for(i = 0; i < l; i++){
        val = val * 10 + (s[i]-'A');
        val = val % mod;
        while(val < 0){
            val += mod;
        }
    }
    return val;
}

void insert_into_symbol_table(char *lexeme, char *token_name){
    int l1 = strlen(lexeme);
    int l2 = strlen(token_name);
    int v = Hash(lexeme);
    if(ST[v].len == 0){
        strcpy(ST[v].name, lexeme);
        strcpy(ST[v].type, token_name);

        ST[v].len = strlen(lexeme);
        return;
    }

    if(strcmp(ST[v].name,lexeme) == 0)
    return;

    int i, pos = 0;

    for (i = 0; i < 1001; i++){
        if(ST[i].len == 0){
            pos = i;
            break;
```

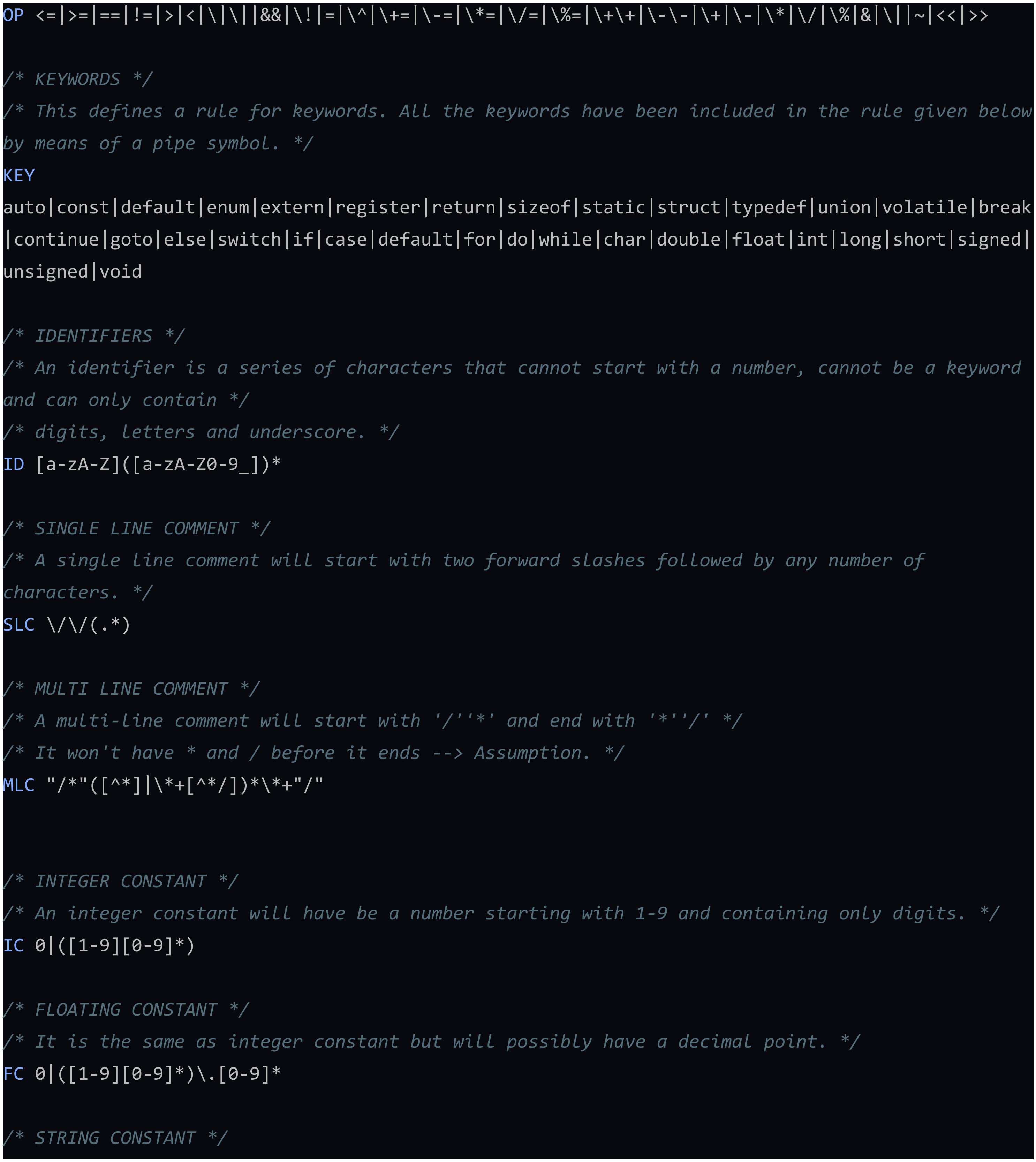


```
        }
    }

    strcpy(ST[pos].name, lexeme);
    strcpy(ST[pos].type, token_name);
    ST[pos].len = strlen(lexeme);

}

void insert_into_constant_table(char *lexeme, char *token_name){
    int l1 = strlen(lexeme);
    int l2 = strlen(token_name);
    int v = Hash(lexeme);
    if(CT[v].len == 0){
        strcpy(CT[v].name, lexeme);
        strcpy(CT[v].type, token_name);

        CT[v].len = strlen(lexeme);
        return;
    }

    if(strcmp(CT[v].name,lexeme) == 0)
    return;

    int i, pos = 0;

    for (i = 0; i < 1001; i++){
        if(CT[i].len == 0){
            pos = i;
            break;
        }
    }

    strcpy(CT[pos].name, lexeme);
    strcpy(CT[pos].type, token_name);
    CT[pos].len = strlen(lexeme);
```

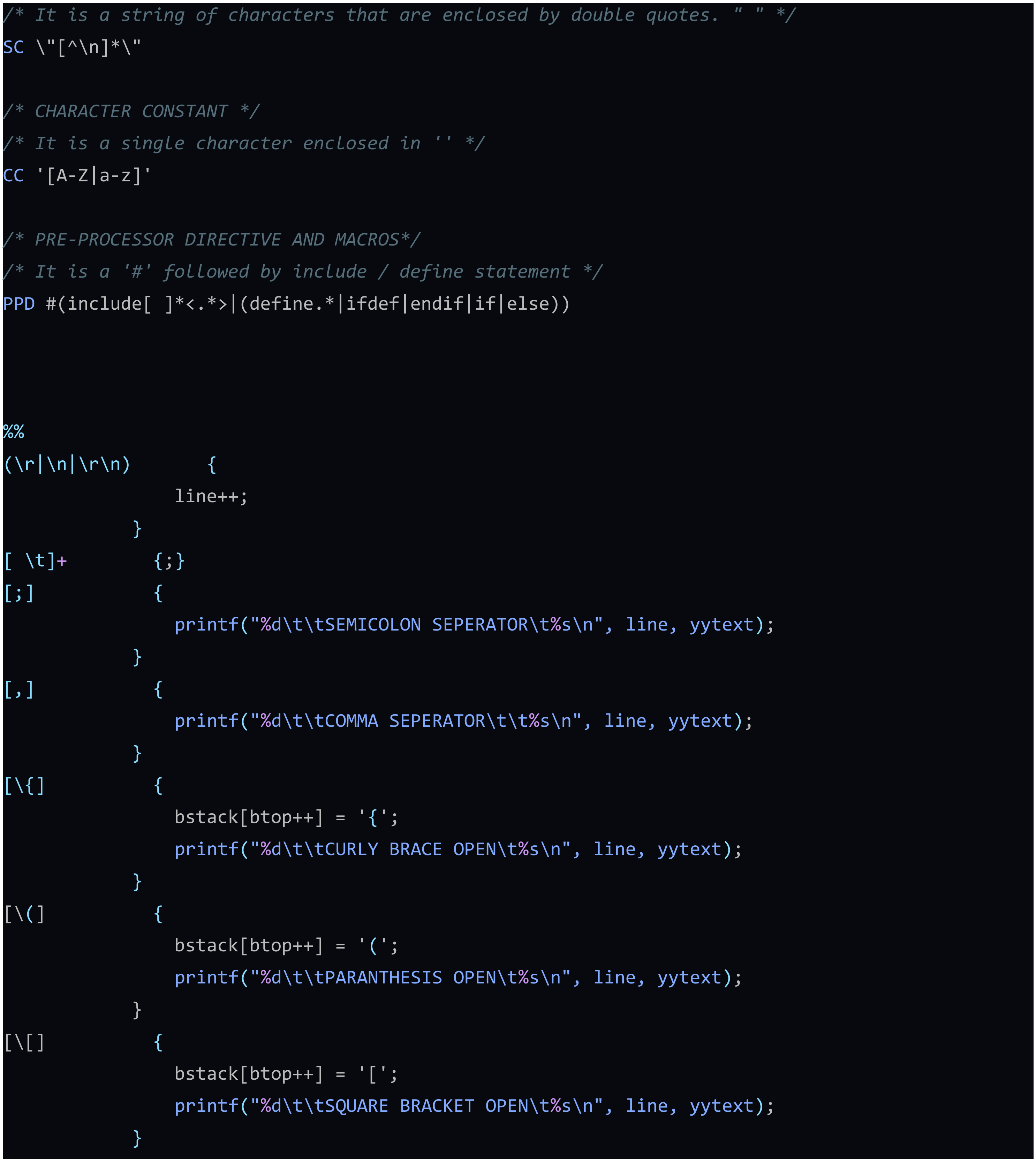


```
    }
   void print_symbol_table() {

printf("========================================================================================
====================================================================\n");
        printf("+++ SYMBOL TABLE +++\n\n");
        printf("TYPE OF TOKEN \t\t TOKEN\n");
        printf("============= \t\t =====\n");
        int i;
        for(i=0; i<1000; i++) {
            if(ST[i].len!=0) printf("%s \t\t %s\n", ST[i].type, ST[i].name);
        }
    }
    // The above function prints the contents of our symbol table line by line.

    void print_constant_table() {

printf("========================================================================================
====================================================================\n");
        printf("+++ CONSTANT TABLE +++\n\n");
        printf("TYPE OF CONSTANT \t\t VALUE OF CONSTANT\n");
        printf("================ \t\t =================\n");
        int i;
        for(i=0; i<1000; i++) {
            if(CT[i].len!=0) printf("%s \t\t %s\n", CT[i].type, CT[i].name);
        }
    }
    // The above function prints the contents of our symbol table line by line.

%}

/* Rules Section */

/* OPERATIONS */
/* This defines the set of all operation symbols used in c programming. */
```

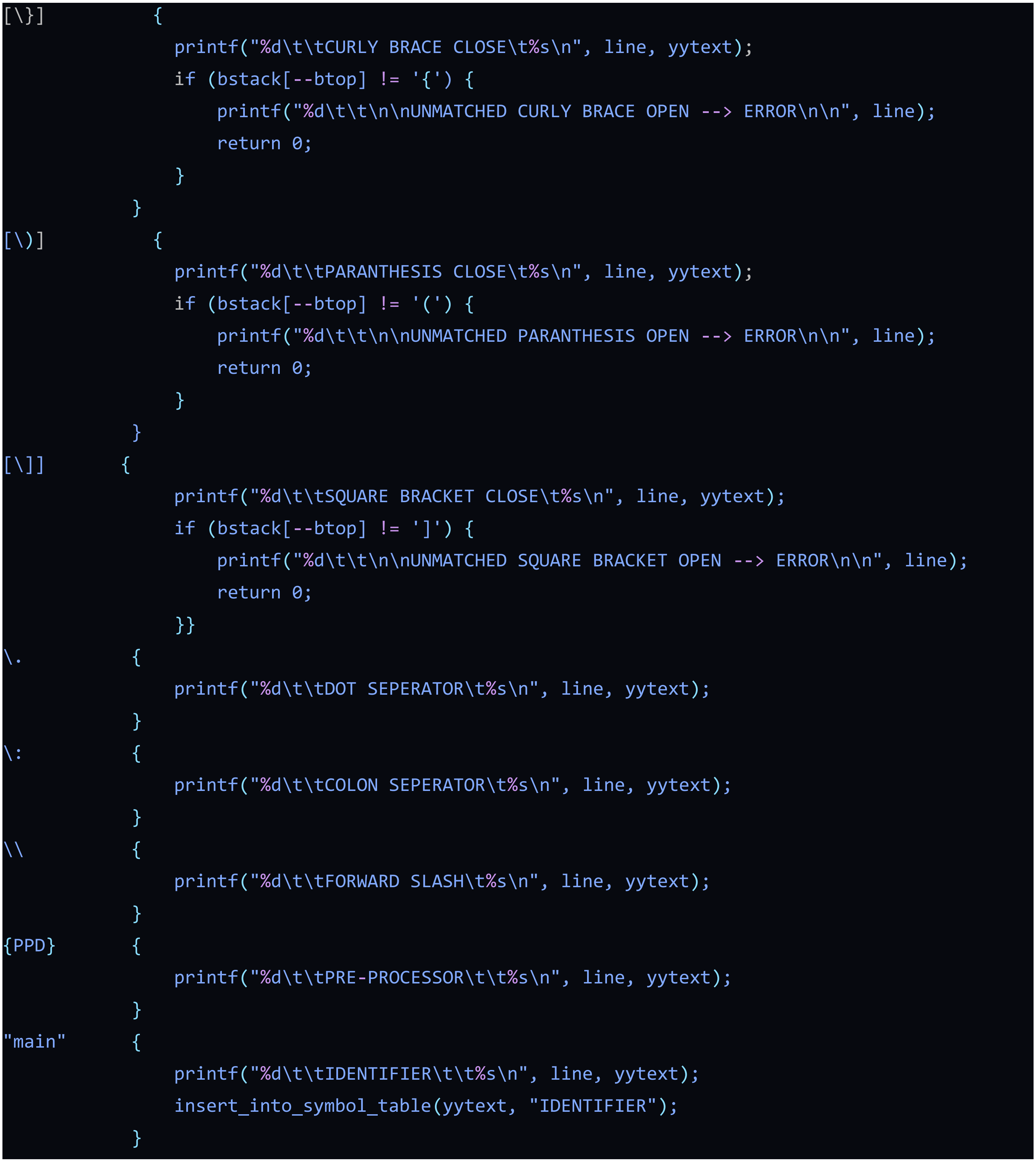
Lexical Analyzer for the C Language



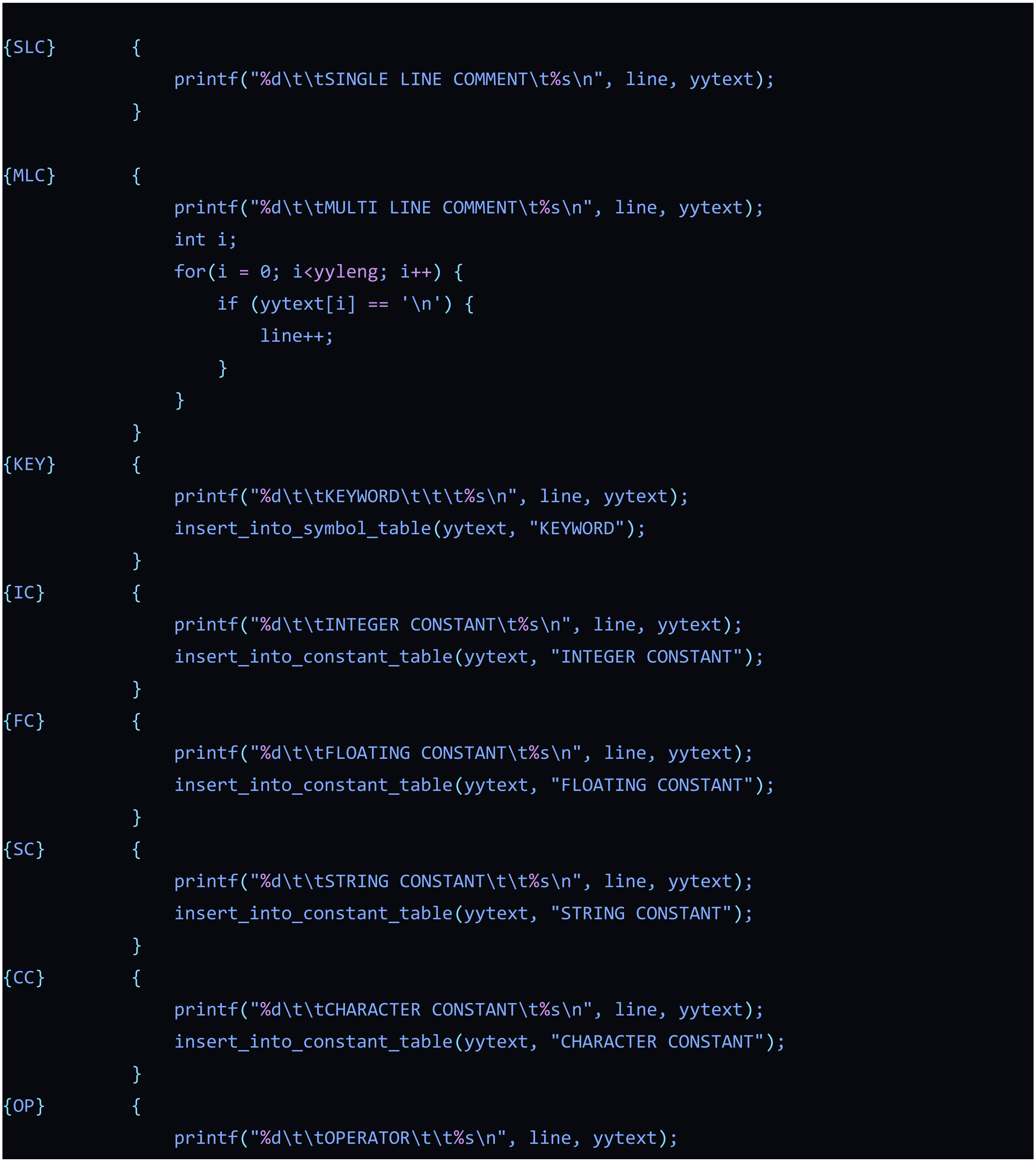
```
OP <=|>=|==|!=|>|<|\|\||&&|\!|=|\^|\+=|\-=|\*=|\/=|\%=|\+\+|\-\-|\+|\-|\*|\/|\%|&|\||~|<<|>>

/* KEYWORDS */
/* This defines a rule for keywords. All the keywords have been included in the rule given below
by means of a pipe symbol. */
KEY
auto|const|default|enum|extern|register|return|sizeof|static|struct|typedef|union|volatile|break
|continue|goto|else|switch|if|case|default|for|do|while|char|double|float|int|long|short|signed|
unsigned|void

/* IDENTIFIERS */
/* An identifier is a series of characters that cannot start with a number, cannot be a keyword
and can only contain */
/* digits, letters and underscore. */
ID [a-zA-Z]([a-zA-Z0-9_])*

/* SINGLE LINE COMMENT */
/* A single line comment will start with two forward slashes followed by any number of
characters. */
SLC \/\/(.*)

/* MULTI LINE COMMENT */
/* A multi-line comment will start with '/''*' and end with '*''/' */
/* It won't have * and / before it ends --> Assumption. */
MLC "/*"([^*]|\*+[^*/])*\*+"/"



/* INTEGER CONSTANT */
/* An integer constant will have be a number starting with 1-9 and containing only digits. */
IC 0|([1-9][0-9]*)

/* FLOATING CONSTANT */
/* It is the same as integer constant but will possibly have a decimal point. */
FC 0|([1-9][0-9]*)\.[0-9]*

/* STRING CONSTANT */
```



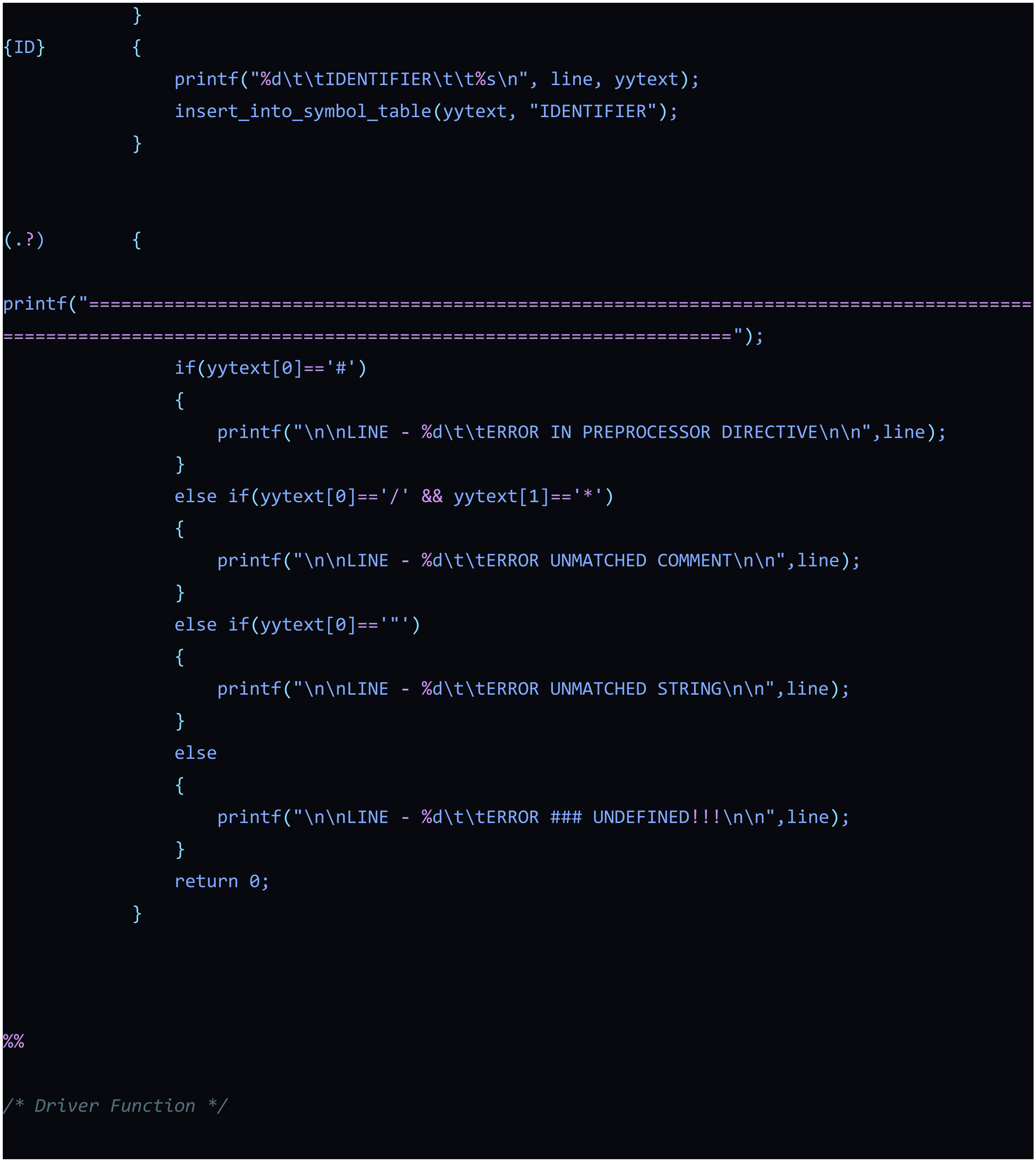
```
/* It is a string of characters that are enclosed by double quotes. " " */
SC \"[^\n]*\"

/* CHARACTER CONSTANT */
/* It is a single character enclosed in '' */
CC '[A-Z|a-z]'

/* PRE-PROCESSOR DIRECTIVE AND MACROS*/
/* It is a '#' followed by include / define statement */
PPD #(include[ ]*<.*>|(define.*|ifdef|endif|if|else))



%%
(\r|\n|\r\n)       {
                line++;
            }
[ \t]+        {;}
[;]           {
                printf("%d\t\tSEMICOLON SEPERATOR\t%s\n", line, yytext);
            }
[,]           {
                printf("%d\t\tCOMMA SEPERATOR\t\t%s\n", line, yytext);
            }
[\{]          {
                bstack[btop++] = '{';
                printf("%d\t\tCURLY BRACE OPEN\t%s\n", line, yytext);
            }
[\(]          {
                bstack[btop++] = '(';
                printf("%d\t\tPARANTHESIS OPEN\t%s\n", line, yytext);
            }
[\[]          {
                bstack[btop++] = '[';
                printf("%d\t\tSQUARE BRACKET OPEN\t%s\n", line, yytext);
            }
```



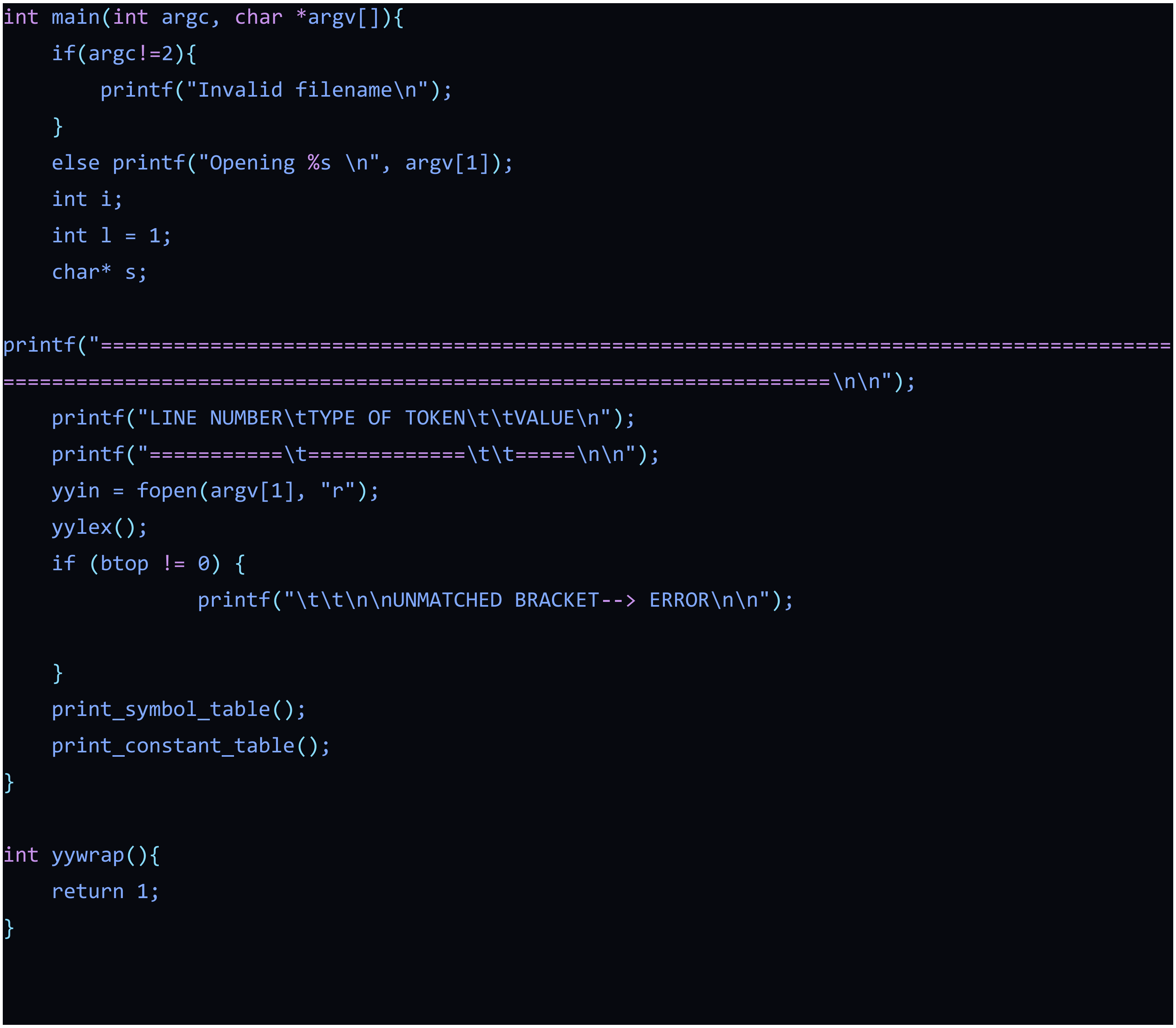
```
[\}]          {
                printf("%d\t\tCURLY BRACE CLOSE\t%s\n", line, yytext);
                if (bstack[--btop] != '{') {
                    printf("%d\t\t\n\nUNMATCHED CURLY BRACE OPEN --> ERROR\n\n", line);
                    return 0;
                }
            }
[\)]          {
                printf("%d\t\tPARANTHESIS CLOSE\t%s\n", line, yytext);
                if (bstack[--btop] != '(') {
                    printf("%d\t\t\n\nUNMATCHED PARANTHESIS OPEN --> ERROR\n\n", line);
                    return 0;
                }
            }
[\]]       {
                printf("%d\t\tSQUARE BRACKET CLOSE\t%s\n", line, yytext);
                if (bstack[--btop] != ']') {
                    printf("%d\t\t\n\nUNMATCHED SQUARE BRACKET OPEN --> ERROR\n\n", line);
                    return 0;
                }}
\.          {
                printf("%d\t\tDOT SEPERATOR\t%s\n", line, yytext);
            }
\:          {
                printf("%d\t\tCOLON SEPERATOR\t%s\n", line, yytext);
            }
\\          {
                printf("%d\t\tFORWARD SLASH\t%s\n", line, yytext);
            }
{PPD}       {
                printf("%d\t\tPRE-PROCESSOR\t\t%s\n", line, yytext);
            }
"main"      {
                printf("%d\t\tIDENTIFIER\t\t%s\n", line, yytext);
                insert_into_symbol_table(yytext, "IDENTIFIER");
            }
```



```
{SLC}       {
                printf("%d\t\tSINGLE LINE COMMENT\t%s\n", line, yytext);
            }

{MLC}       {
                printf("%d\t\tMULTI LINE COMMENT\t%s\n", line, yytext);
                int i;
                for(i = 0; i<yyleng; i++) {
                    if (yytext[i] == '\n') {
                        line++;
                    }
                }
            }
{KEY}       {
                printf("%d\t\tKEYWORD\t\t\t%s\n", line, yytext);
                insert_into_symbol_table(yytext, "KEYWORD");
            }
{IC}        {
                printf("%d\t\tINTEGER CONSTANT\t%s\n", line, yytext);
                insert_into_constant_table(yytext, "INTEGER CONSTANT");
            }
{FC}        {
                printf("%d\t\tFLOATING CONSTANT\t%s\n", line, yytext);
                insert_into_constant_table(yytext, "FLOATING CONSTANT");
            }
{SC}        {
                printf("%d\t\tSTRING CONSTANT\t\t%s\n", line, yytext);
                insert_into_constant_table(yytext, "STRING CONSTANT");
            }
{CC}        {
                printf("%d\t\tCHARACTER CONSTANT\t%s\n", line, yytext);
                insert_into_constant_table(yytext, "CHARACTER CONSTANT");
            }
{OP}        {
                printf("%d\t\tOPERATOR\t\t%s\n", line, yytext);
```



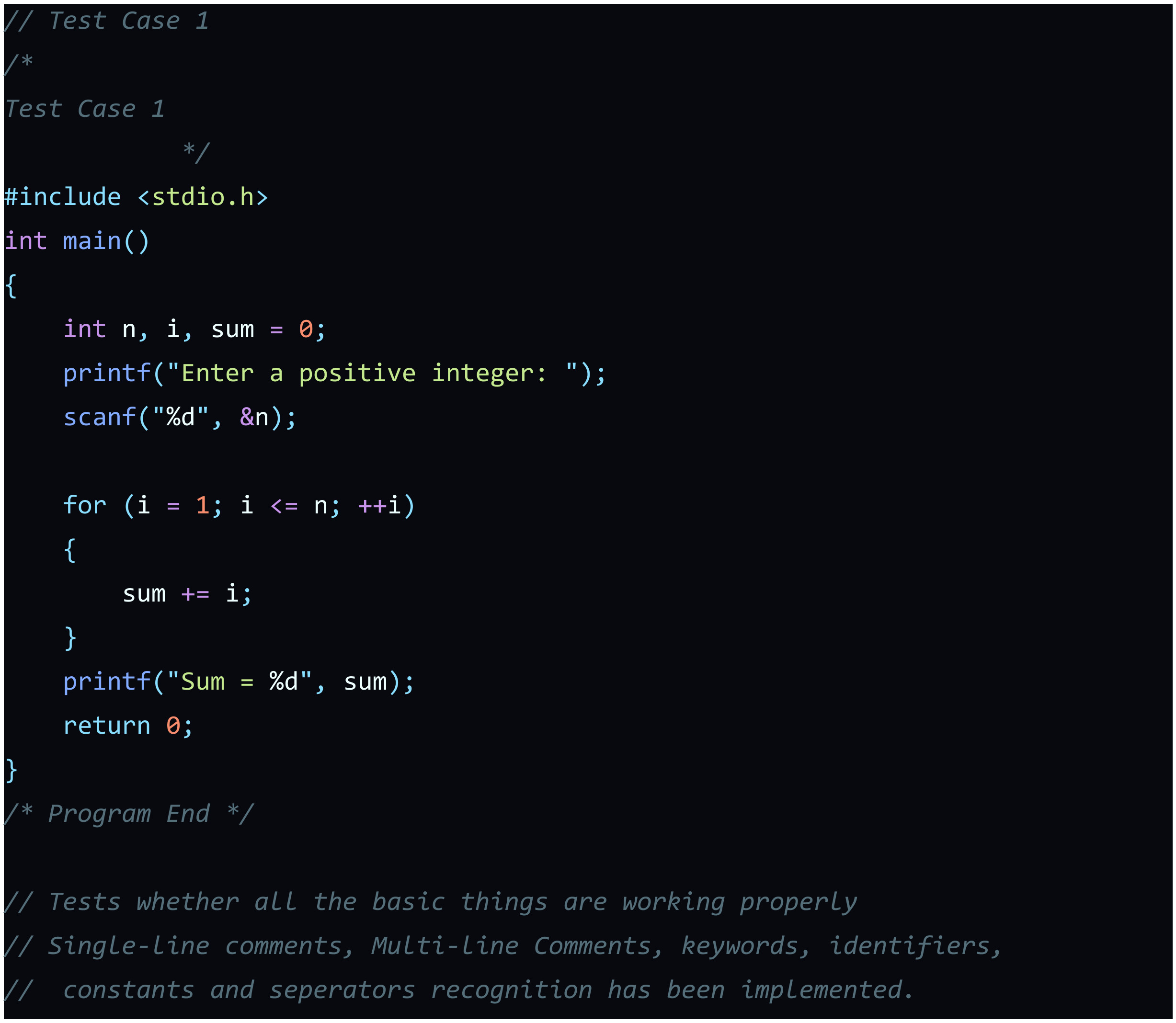
```
            }
{ID}        {
                printf("%d\t\tIDENTIFIER\t\t%s\n", line, yytext);
                insert_into_symbol_table(yytext, "IDENTIFIER");
            }


(.?)        {
printf("=======================================================================================
===================================================================");
                if(yytext[0]=='#')
                {
                    printf("\n\nLINE - %d\t\tERROR IN PREPROCESSOR DIRECTIVE\n\n",line);
                }
                else if(yytext[0]=='/' && yytext[1]=='*')
                {
                    printf("\n\nLINE - %d\t\tERROR UNMATCHED COMMENT\n\n",line);
                }
                else if(yytext[0]=='"')
                {
                    printf("\n\nLINE - %d\t\tERROR UNMATCHED STRING\n\n",line);
                }
                else
                {
                    printf("\n\nLINE - %d\t\tERROR ### UNDEFINED!!!\n\n",line);
                }
                return 0;
            }



%%

/* Driver Function */
```



```
int main(int argc, char *argv[]){
    if(argc!=2){
        printf("Invalid filename\n");
    }
    else printf("Opening %s \n", argv[1]);
    int i;
    int l = 1;
    char* s;

printf("========================================================================================
====================================================================\n\n");
    printf("LINE NUMBER\tTYPE OF TOKEN\t\tVALUE\n");
    printf("===========\t=============\t\t=====\n\n");
    yyin = fopen(argv[1], "r");
    yylex();
    if (btop != 0) {
                printf("\t\t\n\nUNMATCHED BRACKET--> ERROR\n\n");

    }
    print_symbol_table();
    print_constant_table();
}

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

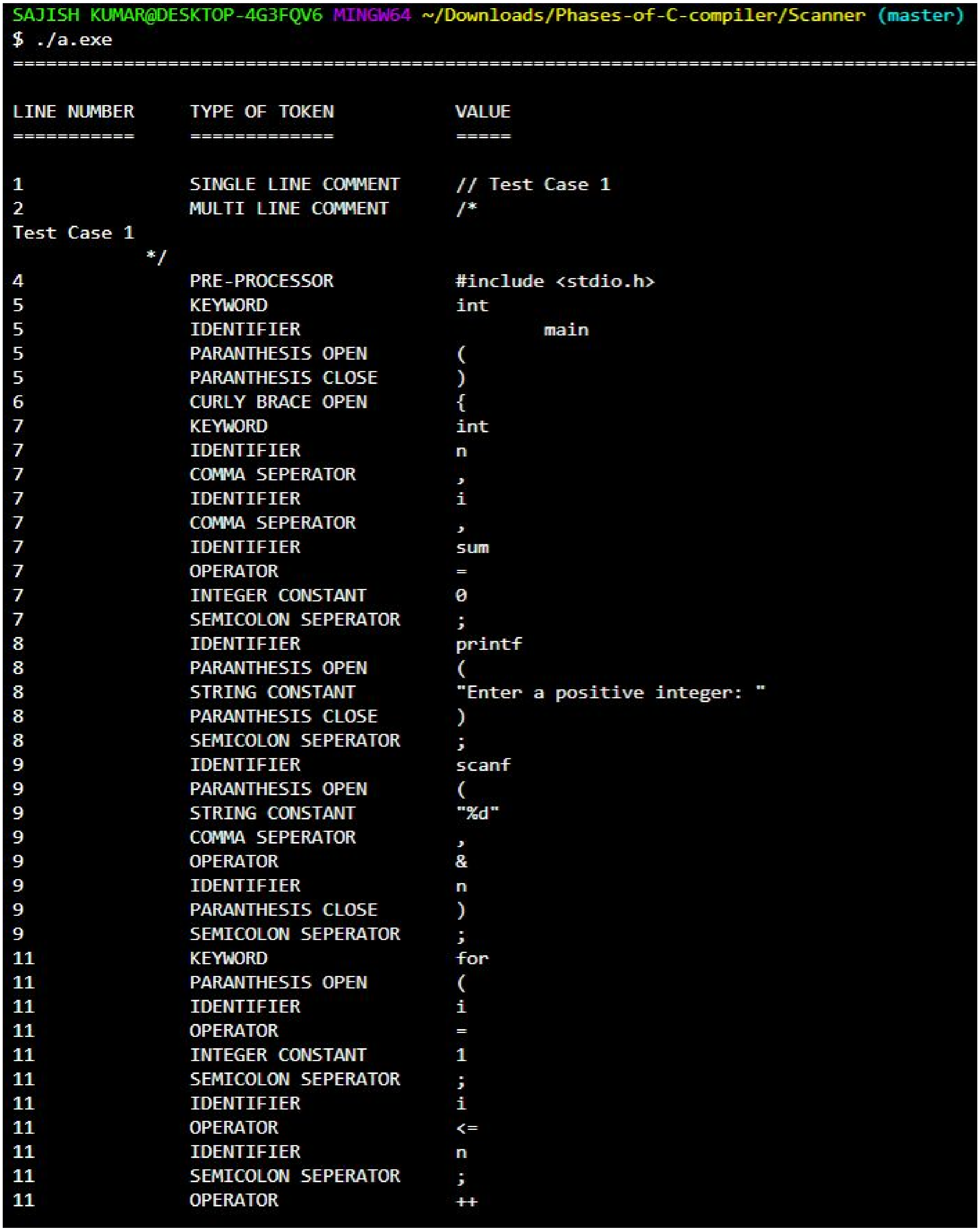
Test Case - 1

This test case tests all the basic parts of a C program. It recognizes keywords, identifiers, operators and separators and comments.

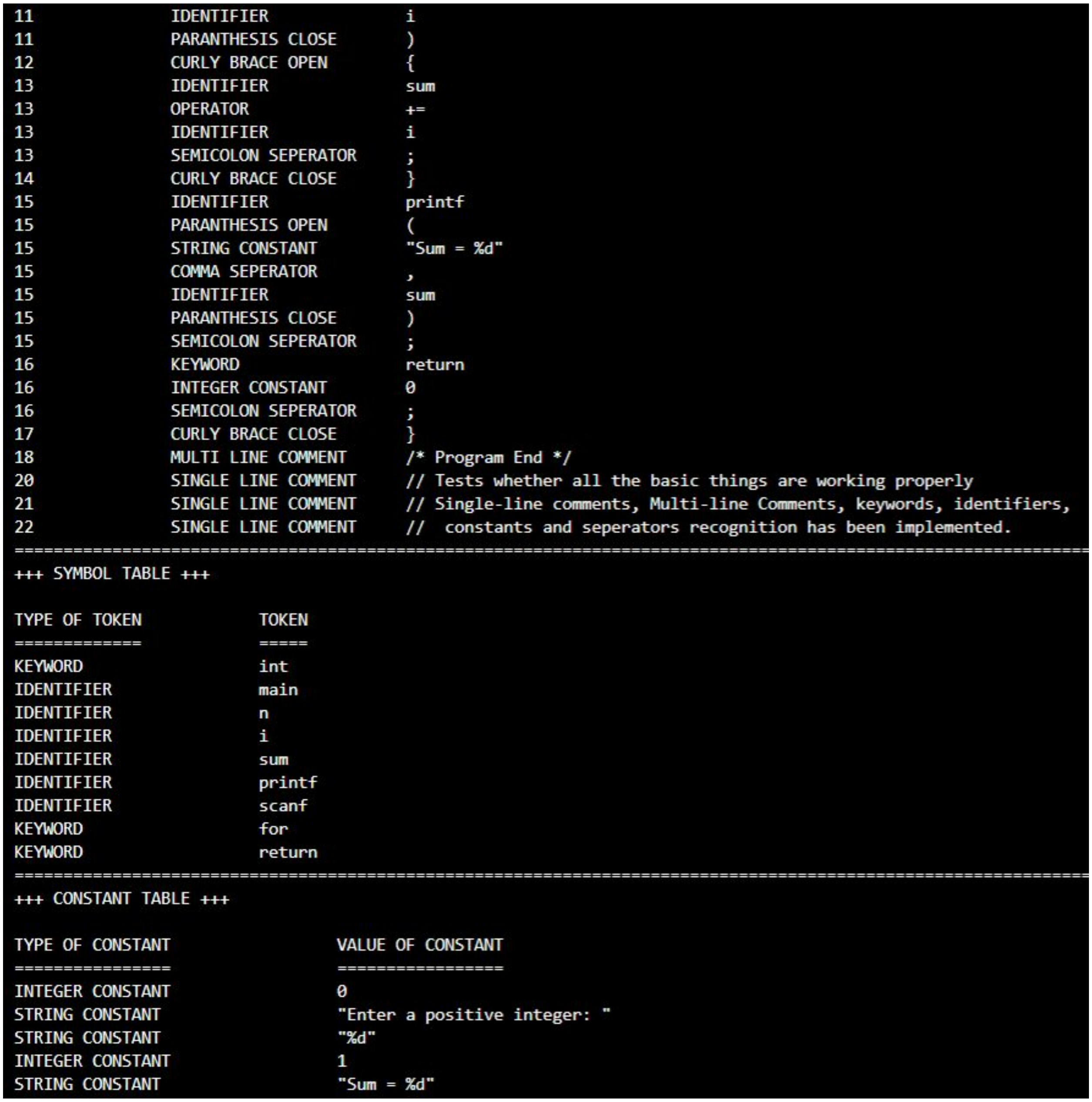


```c
// Test Case 1
/*
Test Case 1
            */
#include <stdio.h>
int main()
{
    int n, i, sum = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);

    for (i = 1; i <= n; ++i)
    {
        sum += i;
    }
    printf("Sum = %d", sum);
    return 0;
}
/* Program End */

// Tests whether all the basic things are working properly
// Single-line comments, Multi-line Comments, keywords, identifiers,
//  constants and seperators recognition has been implemented.
```

Output of Test Case-1



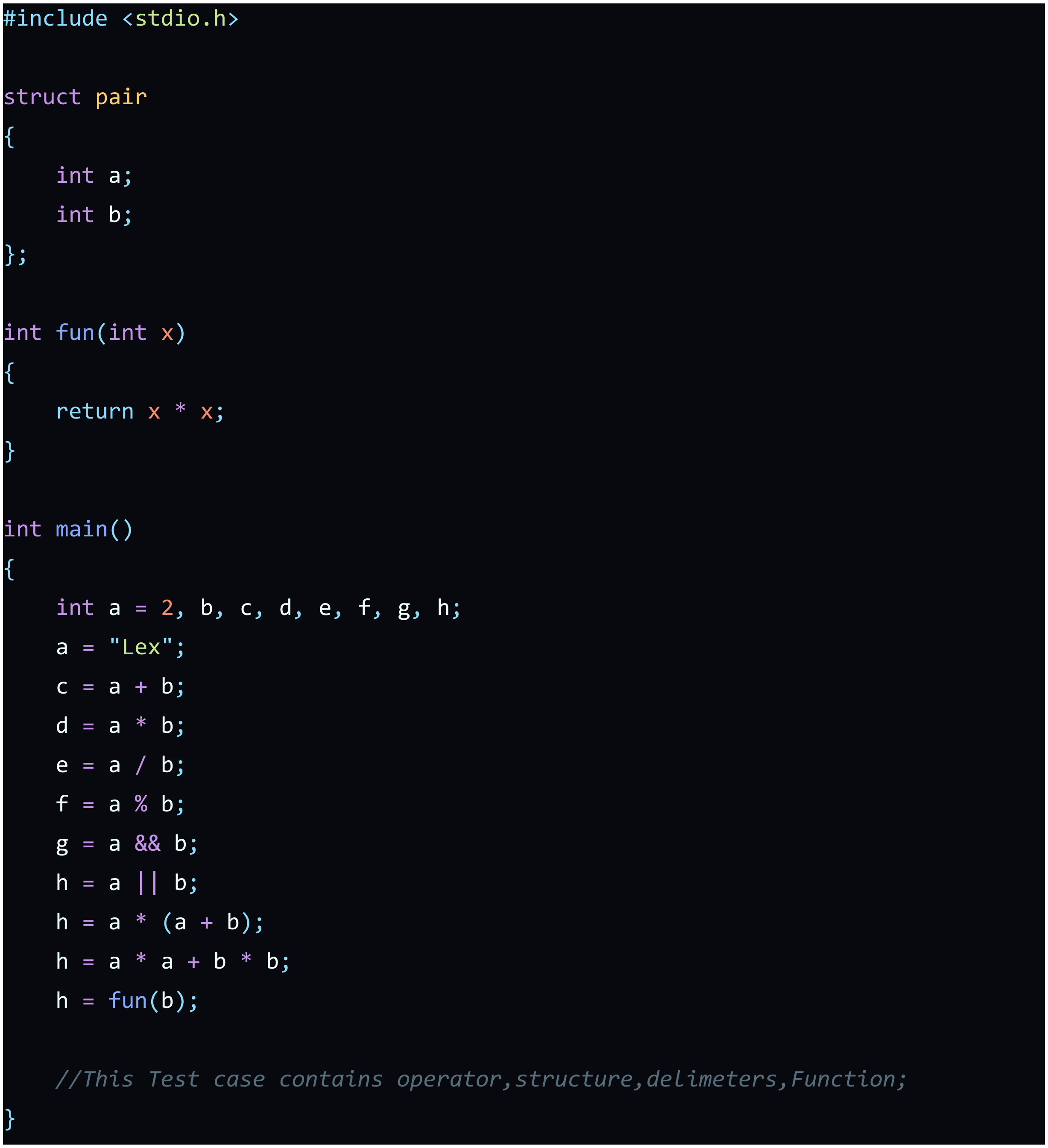
```
SAJISH KUMAR@DESKTOP-4G3FQV6 MINGW64 ~/Downloads/Phases-of-C-compiler/Scanner (master)
$ ./a.exe
==================================================================================

LINE NUMBER     TYPE OF TOKEN           VALUE
===========     =============           =====

1               SINGLE LINE COMMENT     // Test Case 1
2               MULTI LINE COMMENT      /*
Test Case 1
            */
4               PRE-PROCESSOR           #include <stdio.h>
5               KEYWORD                 int
5               IDENTIFIER                      main
5               PARANTHESIS OPEN        (
5               PARANTHESIS CLOSE       )
6               CURLY BRACE OPEN        {
7               KEYWORD                 int
7               IDENTIFIER              n
7               COMMA SEPERATOR         ,
7               IDENTIFIER              i
7               COMMA SEPERATOR         ,
7               IDENTIFIER              sum
7               OPERATOR                =
7               INTEGER CONSTANT        0
7               SEMICOLON SEPERATOR     ;
8               IDENTIFIER              printf
8               PARANTHESIS OPEN        (
8               STRING CONSTANT         "Enter a positive integer: "
8               PARANTHESIS CLOSE       )
8               SEMICOLON SEPERATOR     ;
9               IDENTIFIER              scanf
9               PARANTHESIS OPEN        (
9               STRING CONSTANT         "%d"
9               COMMA SEPERATOR         ,
9               OPERATOR                &
9               IDENTIFIER              n
9               PARANTHESIS CLOSE       )
9               SEMICOLON SEPERATOR     ;
11              KEYWORD                 for
11              PARANTHESIS OPEN        (
11              IDENTIFIER              i
11              OPERATOR                =
11              INTEGER CONSTANT        1
11              SEMICOLON SEPERATOR     ;
11              IDENTIFIER              i
11              OPERATOR                <=
11              IDENTIFIER              n
11              SEMICOLON SEPERATOR     ;
11              OPERATOR                ++
```



```
11              IDENTIFIER              i
11              PARANTHESIS CLOSE       )
12              CURLY BRACE OPEN        {
13              IDENTIFIER              sum
13              OPERATOR                +=
13              IDENTIFIER              i
13              SEMICOLON SEPERATOR     ;
14              CURLY BRACE CLOSE       }
15              IDENTIFIER              printf
15              PARANTHESIS OPEN        (
15              STRING CONSTANT         "Sum = %d"
15              COMMA SEPERATOR         ,
15              IDENTIFIER              sum
15              PARANTHESIS CLOSE       )
15              SEMICOLON SEPERATOR     ;
16              KEYWORD                 return
16              INTEGER CONSTANT        0
16              SEMICOLON SEPERATOR     ;
17              CURLY BRACE CLOSE       }
18              MULTI LINE COMMENT      /* Program End */
20              SINGLE LINE COMMENT     // Tests whether all the basic things are working properly
21              SINGLE LINE COMMENT     // Single-line comments, Multi-line Comments, keywords, identifiers,
22              SINGLE LINE COMMENT     //  constants and seperators recognition has been implemented.
==============================================================================================================
+++ SYMBOL TABLE +++

TYPE OF TOKEN            TOKEN
=============            =====
KEYWORD                  int
IDENTIFIER               main
IDENTIFIER               n
IDENTIFIER               i
IDENTIFIER               sum
IDENTIFIER               printf
IDENTIFIER               scanf
KEYWORD                  for
KEYWORD                  return
==============================================================================================================
+++ CONSTANT TABLE +++

TYPE OF CONSTANT                 VALUE OF CONSTANT
================                 =================
INTEGER CONSTANT                 0
STRING CONSTANT                  "Enter a positive integer: "
STRING CONSTANT                  "%d"
INTEGER CONSTANT                 1
STRING CONSTANT                  "Sum = %d"
```

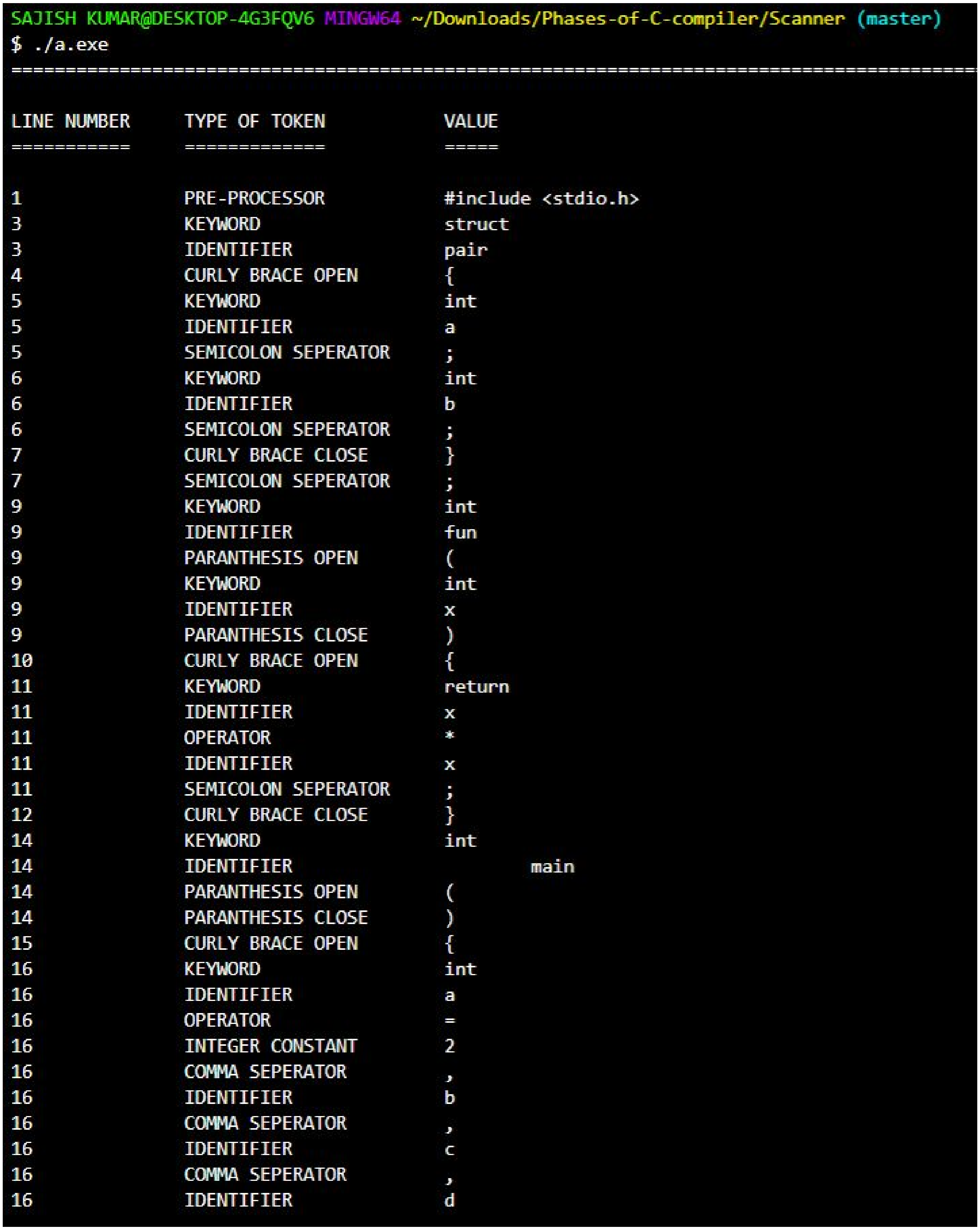
Test Case - 2

This program is also a basic C program to show that all the identifiers and operations are working properly.

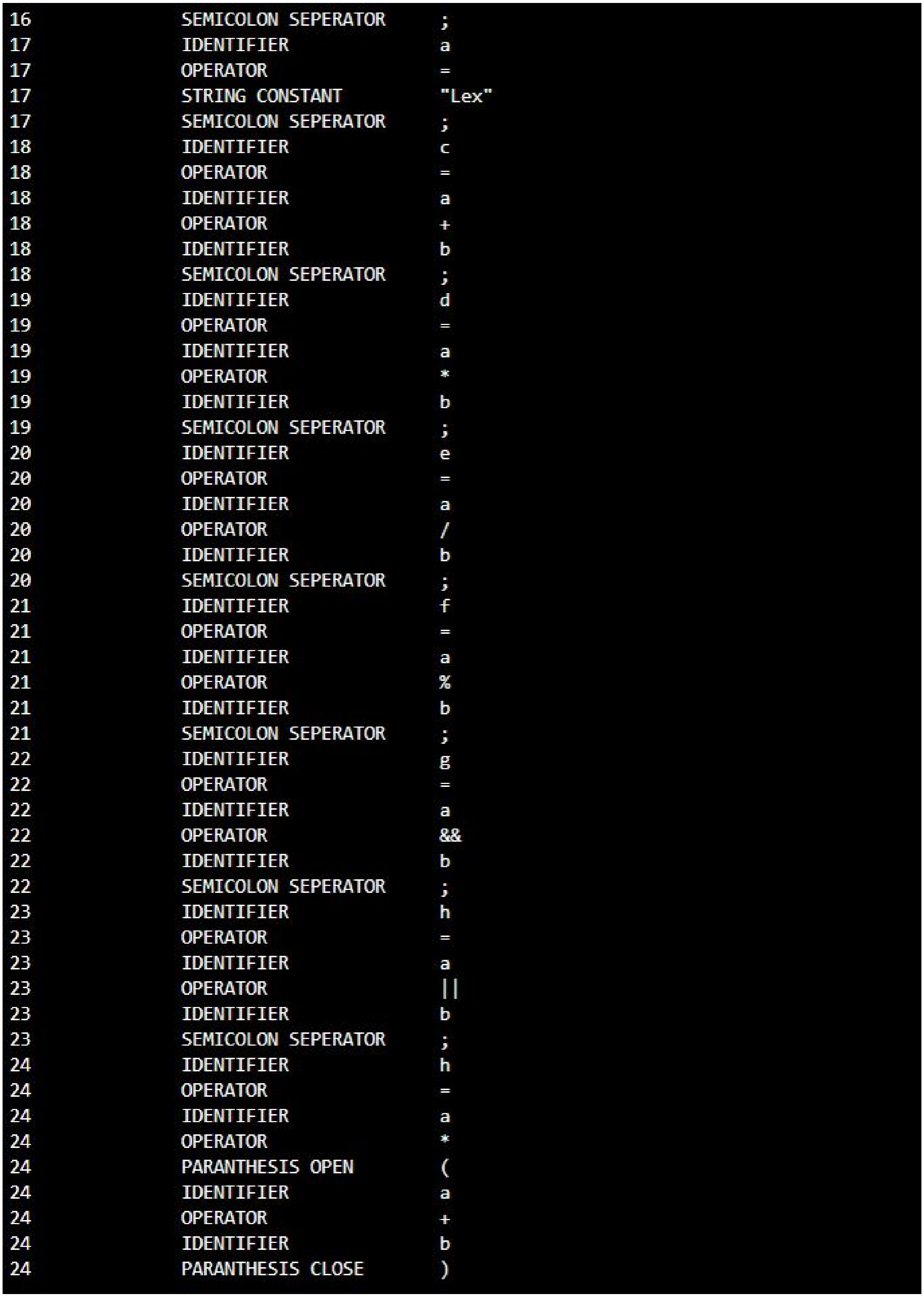


```c
#include <stdio.h>

struct pair
{
    int a;
    int b;
};

int fun(int x)
{
    return x * x;
}

int main()
{
    int a = 2, b, c, d, e, f, g, h;
    a = "Lex";
    c = a + b;
    d = a * b;
    e = a / b;
    f = a % b;
    g = a && b;
    h = a || b;
    h = a * (a + b);
    h = a * a + b * b;
    h = fun(b);

    //This Test case contains operator,structure,delimeters,Function;
}
```

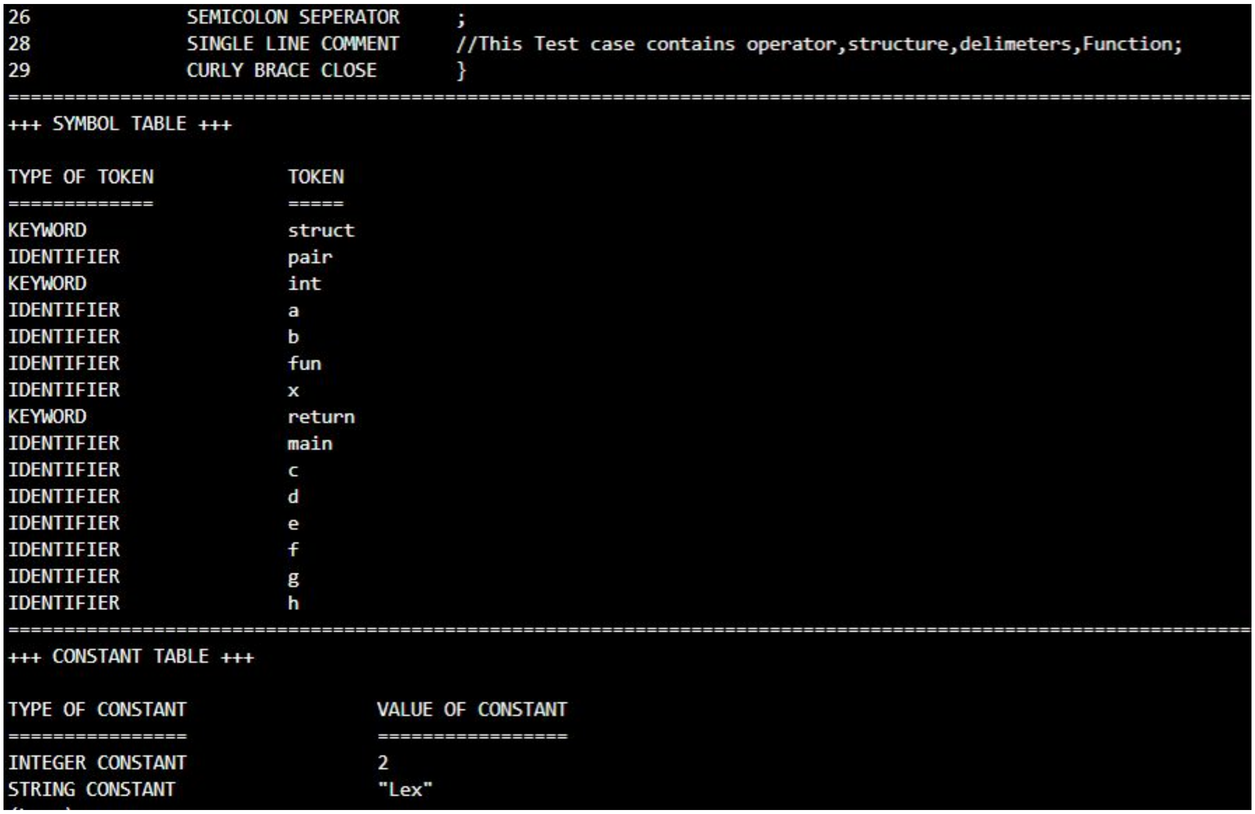
Output of Test Case 2



```
SAJISH KUMAR@DESKTOP-4G3FQV6 MINGW64 ~/Downloads/Phases-of-C-compiler/Scanner (master)
$ ./a.exe
==========================================================================================

LINE NUMBER     TYPE OF TOKEN           VALUE
===========     =============           =====

1               PRE-PROCESSOR           #include <stdio.h>
3               KEYWORD                 struct
3               IDENTIFIER              pair
4               CURLY BRACE OPEN        {
5               KEYWORD                 int
5               IDENTIFIER              a
5               SEMICOLON SEPERATOR     ;
6               KEYWORD                 int
6               IDENTIFIER              b
6               SEMICOLON SEPERATOR     ;
7               CURLY BRACE CLOSE       }
7               SEMICOLON SEPERATOR     ;
9               KEYWORD                 int
9               IDENTIFIER              fun
9               PARANTHESIS OPEN        (
9               KEYWORD                 int
9               IDENTIFIER              x
9               PARANTHESIS CLOSE       )
10              CURLY BRACE OPEN        {
11              KEYWORD                 return
11              IDENTIFIER              x
11              OPERATOR                *
11              IDENTIFIER              x
11              SEMICOLON SEPERATOR     ;
12              CURLY BRACE CLOSE       }
14              KEYWORD                 int
14              IDENTIFIER                      main
14              PARANTHESIS OPEN        (
14              PARANTHESIS CLOSE       )
15              CURLY BRACE OPEN        {
16              KEYWORD                 int
16              IDENTIFIER              a
16              OPERATOR                =
16              INTEGER CONSTANT        2
16              COMMA SEPERATOR         ,
16              IDENTIFIER              b
16              COMMA SEPERATOR         ,
16              IDENTIFIER              c
16              COMMA SEPERATOR         ,
16              IDENTIFIER              d
```



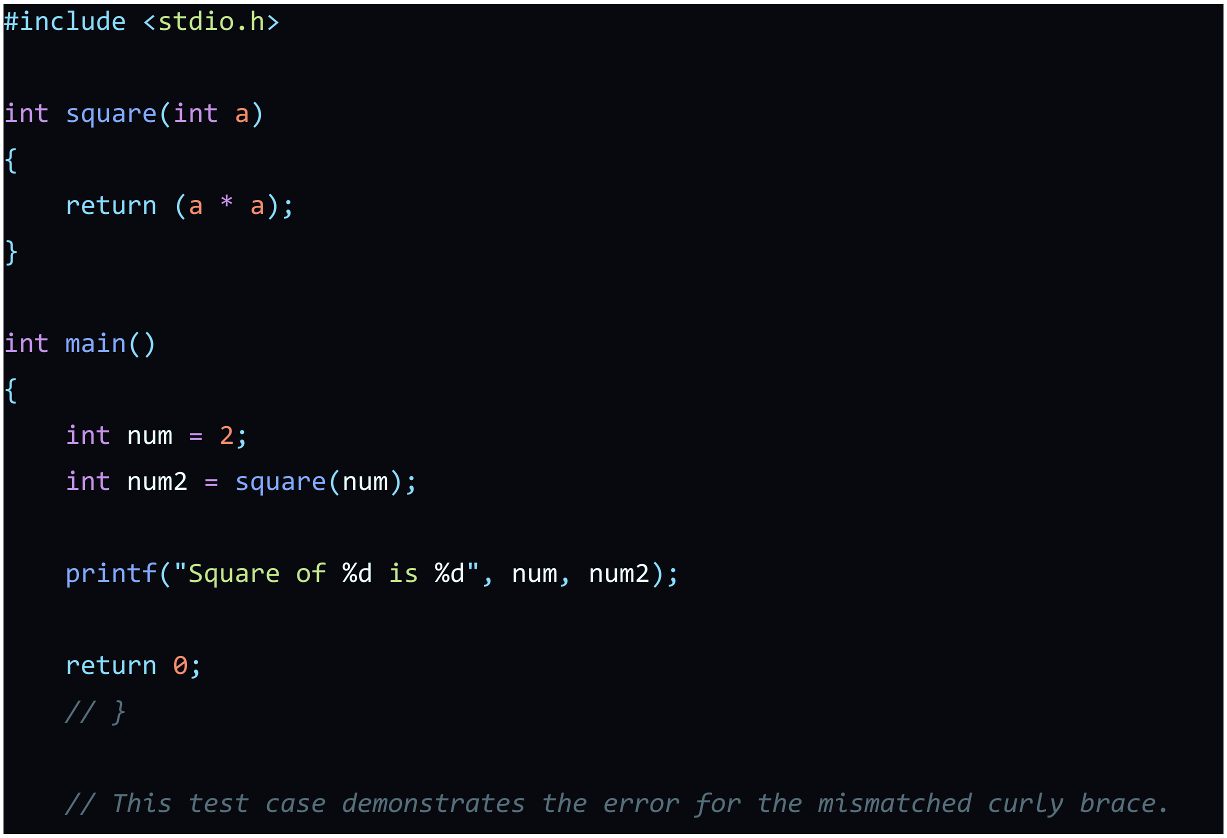
```
16              SEMICOLON SEPERATOR     ;
17              IDENTIFIER              a
17              OPERATOR                =
17              STRING CONSTANT         "Lex"
17              SEMICOLON SEPERATOR     ;
18              IDENTIFIER              c
18              OPERATOR                =
18              IDENTIFIER              a
18              OPERATOR                +
18              IDENTIFIER              b
18              SEMICOLON SEPERATOR     ;
19              IDENTIFIER              d
19              OPERATOR                =
19              IDENTIFIER              a
19              OPERATOR                *
19              IDENTIFIER              b
19              SEMICOLON SEPERATOR     ;
20              IDENTIFIER              e
20              OPERATOR                =
20              IDENTIFIER              a
20              OPERATOR                /
20              IDENTIFIER              b
20              SEMICOLON SEPERATOR     ;
21              IDENTIFIER              f
21              OPERATOR                =
21              IDENTIFIER              a
21              OPERATOR                %
21              IDENTIFIER              b
21              SEMICOLON SEPERATOR     ;
22              IDENTIFIER              g
22              OPERATOR                =
22              IDENTIFIER              a
22              OPERATOR                &&
22              IDENTIFIER              b
22              SEMICOLON SEPERATOR     ;
23              IDENTIFIER              h
23              OPERATOR                =
23              IDENTIFIER              a
23              OPERATOR                ||
23              IDENTIFIER              b
23              SEMICOLON SEPERATOR     ;
24              IDENTIFIER              h
24              OPERATOR                =
24              IDENTIFIER              a
24              OPERATOR                *
24              PARANTHESIS OPEN        (
24              IDENTIFIER              a
24              OPERATOR                +
24              IDENTIFIER              b
24              PARANTHESIS CLOSE       )
```



```
26              SEMICOLON SEPERATOR     ;
28              SINGLE LINE COMMENT     //This Test case contains operator,structure,delimeters,Function;
29              CURLY BRACE CLOSE       }
==============================================================================================================
+++ SYMBOL TABLE +++

TYPE OF TOKEN            TOKEN
=============            =====
KEYWORD                  struct
IDENTIFIER               pair
KEYWORD                  int
IDENTIFIER               a
IDENTIFIER               b
IDENTIFIER               fun
IDENTIFIER               x
KEYWORD                  return
IDENTIFIER               main
IDENTIFIER               c
IDENTIFIER               d
IDENTIFIER               e
IDENTIFIER               f
IDENTIFIER               g
IDENTIFIER               h
==============================================================================================================
+++ CONSTANT TABLE +++

TYPE OF CONSTANT                 VALUE OF CONSTANT
================                 =================
INTEGER CONSTANT                 2
STRING CONSTANT                  "Lex"
```

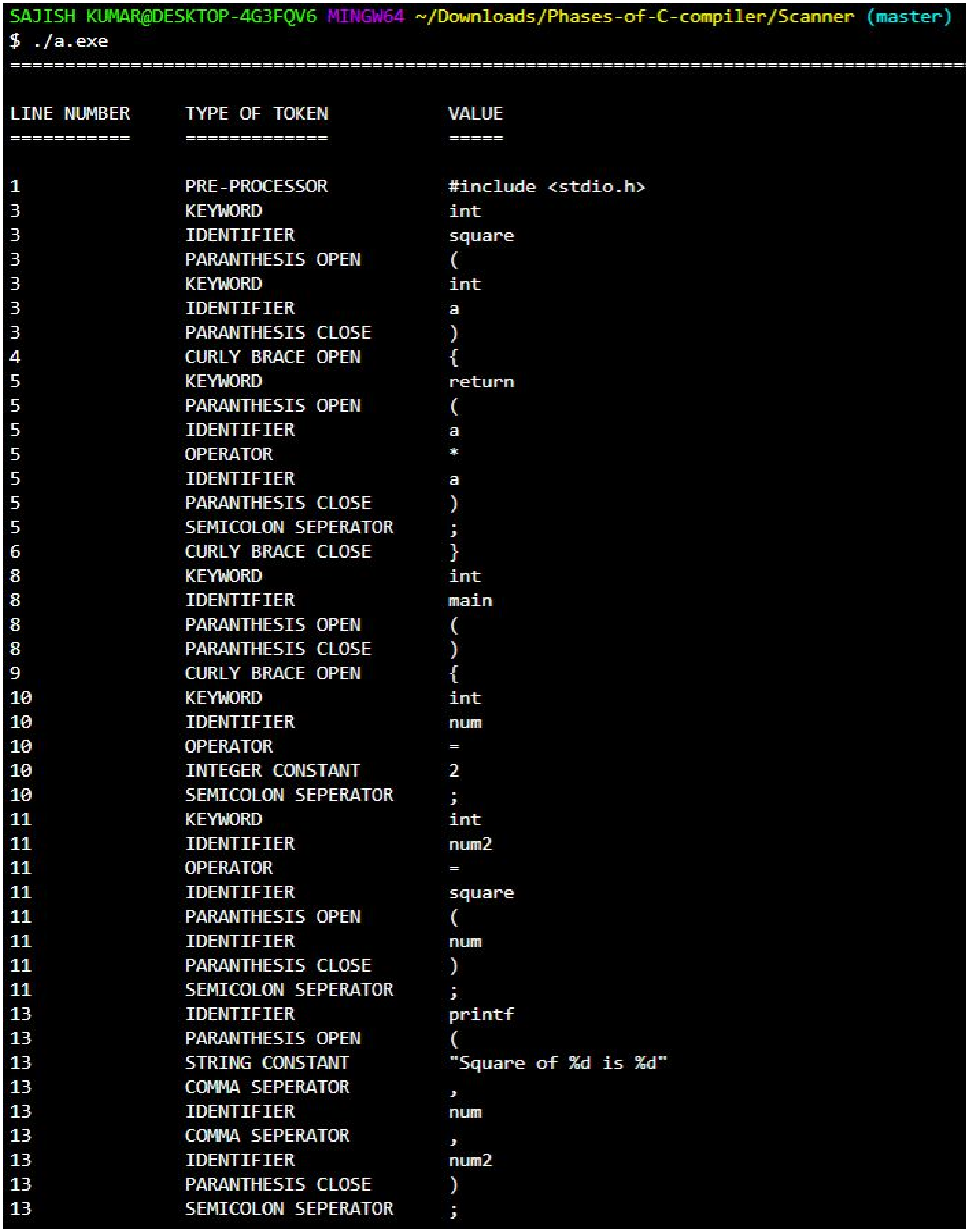
Test Case - 3

This test case aims to demonstrate the unmatched bracket functionality.

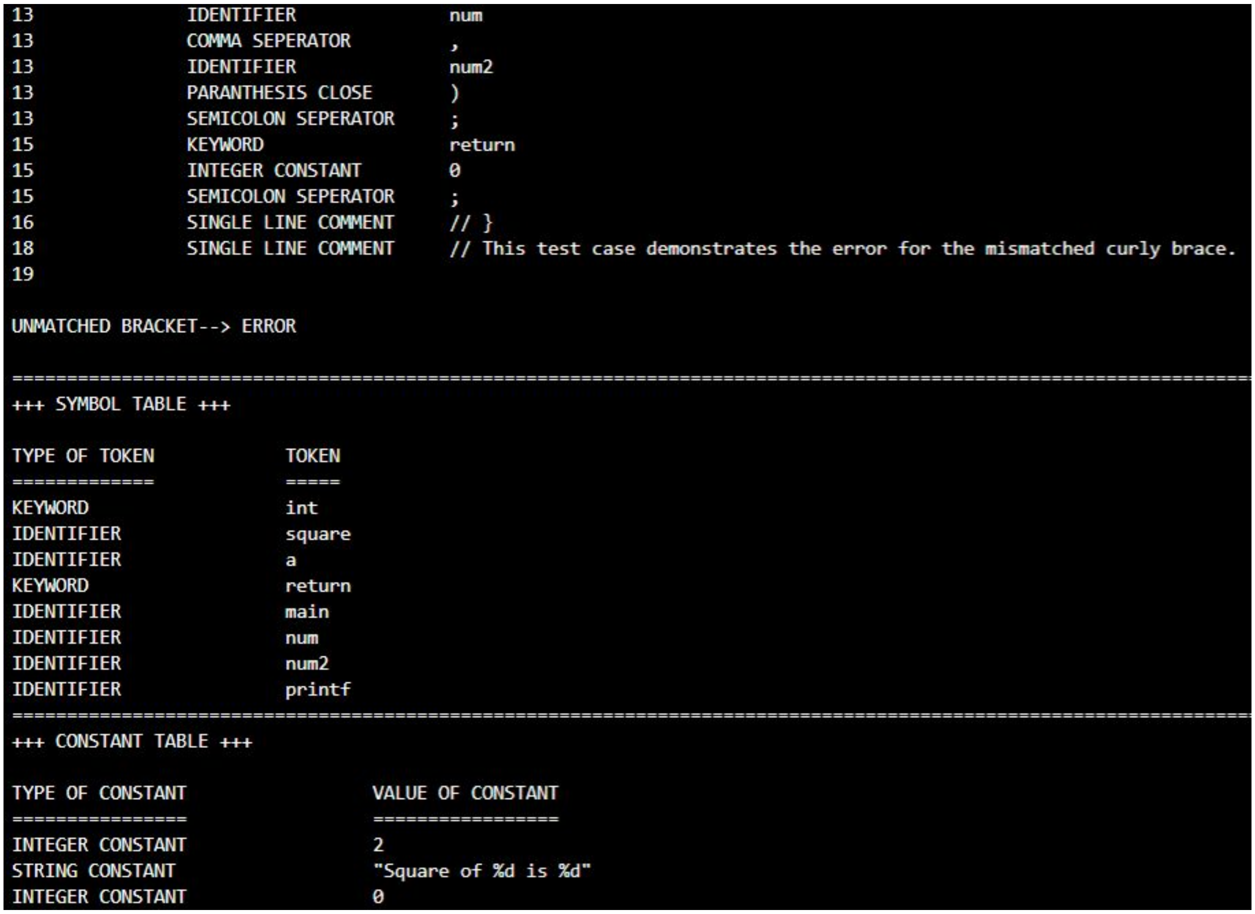


```
#include <stdio.h>

int square(int a)
{
    return (a * a);
}

int main()
{
    int num = 2;
    int num2 = square(num);

    printf("Square of %d is %d", num, num2);

    return 0;
    // }

    // This test case demonstrates the error for the mismatched curly brace.
```

Output of Test Case 3



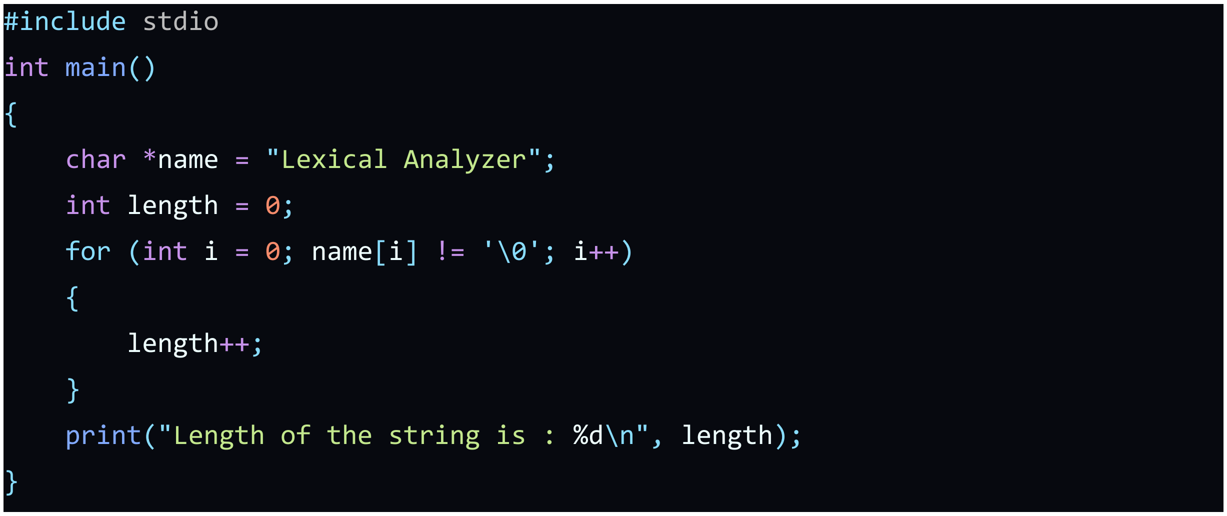
```
SAJISH KUMAR@DESKTOP-4G3FQV6 MINGW64 ~/Downloads/Phases-of-C-compiler/Scanner (master)
$ ./a.exe
==============================================================================================

LINE NUMBER     TYPE OF TOKEN           VALUE
===========     =============           =====

1               PRE-PROCESSOR           #include <stdio.h>
3               KEYWORD                 int
3               IDENTIFIER              square
3               PARANTHESIS OPEN        (
3               KEYWORD                 int
3               IDENTIFIER              a
3               PARANTHESIS CLOSE       )
4               CURLY BRACE OPEN        {
5               KEYWORD                 return
5               PARANTHESIS OPEN        (
5               IDENTIFIER              a
5               OPERATOR                *
5               IDENTIFIER              a
5               PARANTHESIS CLOSE       )
5               SEMICOLON SEPERATOR     ;
6               CURLY BRACE CLOSE       }
8               KEYWORD                 int
8               IDENTIFIER              main
8               PARANTHESIS OPEN        (
8               PARANTHESIS CLOSE       )
9               CURLY BRACE OPEN        {
10              KEYWORD                 int
10              IDENTIFIER              num
10              OPERATOR                =
10              INTEGER CONSTANT        2
10              SEMICOLON SEPERATOR     ;
11              KEYWORD                 int
11              IDENTIFIER              num2
11              OPERATOR                =
11              IDENTIFIER              square
11              PARANTHESIS OPEN        (
11              IDENTIFIER              num
11              PARANTHESIS CLOSE       )
11              SEMICOLON SEPERATOR     ;
13              IDENTIFIER              printf
13              PARANTHESIS OPEN        (
13              STRING CONSTANT         "Square of %d is %d"
13              COMMA SEPERATOR         ,
13              IDENTIFIER              num
13              COMMA SEPERATOR         ,
13              IDENTIFIER              num2
13              PARANTHESIS CLOSE       )
13              SEMICOLON SEPERATOR     ;
```



```
13              IDENTIFIER              num
13              COMMA SEPERATOR         ,
13              IDENTIFIER              num2
13              PARANTHESIS CLOSE       )
13              SEMICOLON SEPERATOR     ;
15              KEYWORD                 return
15              INTEGER CONSTANT        0
15              SEMICOLON SEPERATOR     ;
16              SINGLE LINE COMMENT     // }
18              SINGLE LINE COMMENT     // This test case demonstrates the error for the mismatched curly brace.
19

UNMATCHED BRACKET--> ERROR

==================================================================================================================
+++ SYMBOL TABLE +++

TYPE OF TOKEN           TOKEN
=============           =====
KEYWORD                 int
IDENTIFIER              square
IDENTIFIER              a
KEYWORD                 return
IDENTIFIER              main
IDENTIFIER              num
IDENTIFIER              num2
IDENTIFIER              printf
==================================================================================================================
+++ CONSTANT TABLE +++

TYPE OF CONSTANT                VALUE OF CONSTANT
================                =================
INTEGER CONSTANT                2
STRING CONSTANT                 "Square of %d is %d"
INTEGER CONSTANT                0
```

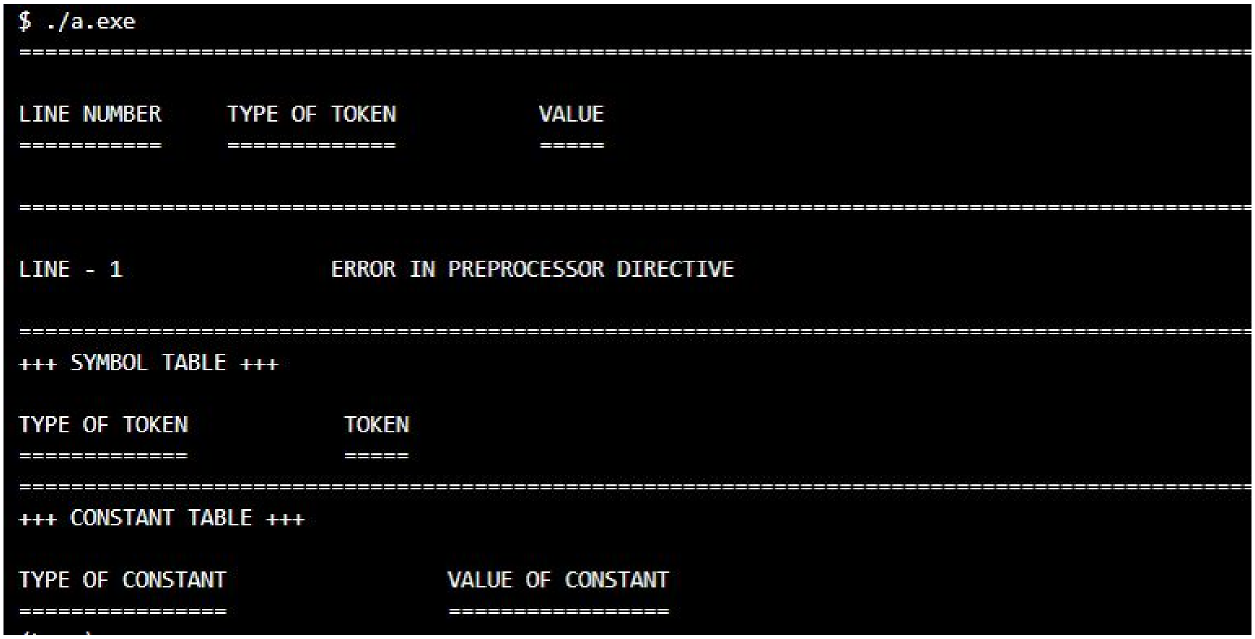
Test Case - 4

This test case aims to demonstrate the error in the preprocessor statement.



```
#include stdio
int main()
{
    char *name = "Lexical Analyzer";
    int length = 0;
    for (int i = 0; name[i] != '\0'; i++)
    {
        length++;
    }
    print("Length of the string is : %d\n", length);
}
```

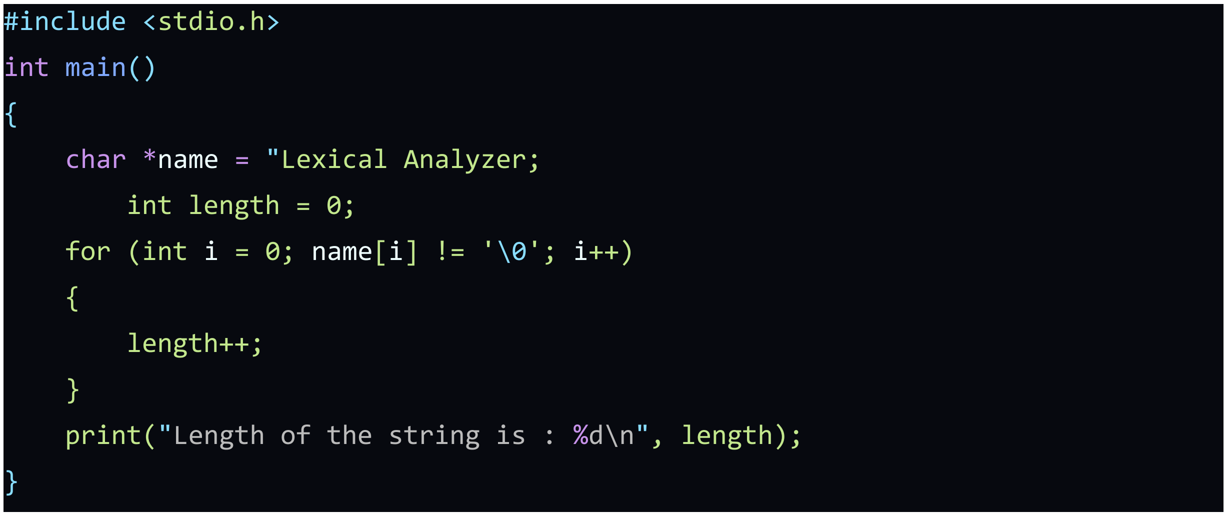
Output of Test Case-4



```
$ ./a.exe
================================================================================================

LINE NUMBER     TYPE OF TOKEN           VALUE
===========     =============           =====

================================================================================================

LINE - 1                ERROR IN PREPROCESSOR DIRECTIVE

================================================================================================
+++ SYMBOL TABLE +++

TYPE OF TOKEN             TOKEN
=============             =====
================================================================================================
+++ CONSTANT TABLE +++

TYPE OF CONSTANT                  VALUE OF CONSTANT
================                  =================
```

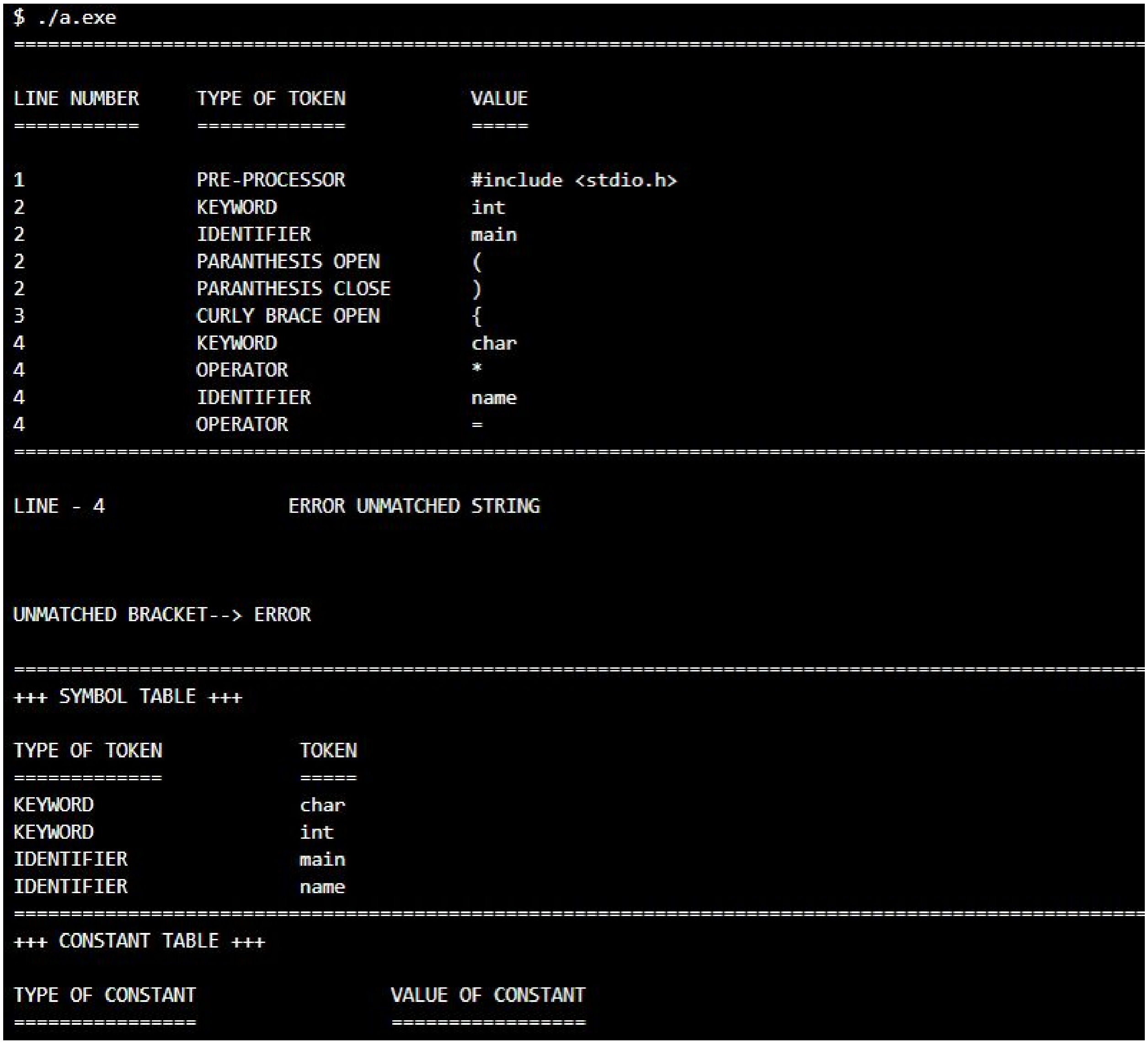
Test Case - 5

This test case illustrates the error in matching strings. If a string has started and isn’t closed, then this error is raised. The test case below has an unmatched string in line 4. Because of this, the curly brace after the main() function also becomes unmatched.



```
#include <stdio.h>
int main()
{
    char *name = "Lexical Analyzer;
        int length = 0;
    for (int i = 0; name[i] != '\0'; i++)
    {
        length++;
    }
    print("Length of the string is : %d\n", length);
}
```

Output of Test Case-5



```
$ ./a.exe
===================================================================================================

LINE NUMBER     TYPE OF TOKEN           VALUE
===========     =============           =====

1               PRE-PROCESSOR           #include <stdio.h>
2               KEYWORD                 int
2               IDENTIFIER              main
2               PARANTHESIS OPEN        (
2               PARANTHESIS CLOSE       )
3               CURLY BRACE OPEN        {
4               KEYWORD                 char
4               OPERATOR                *
4               IDENTIFIER              name
4               OPERATOR                =
===================================================================================================

LINE - 4                ERROR UNMATCHED STRING



UNMATCHED BRACKET--> ERROR

===================================================================================================
+++ SYMBOL TABLE +++

TYPE OF TOKEN            TOKEN
=============            =====
KEYWORD                  char
KEYWORD                  int
IDENTIFIER               main
IDENTIFIER               name
===================================================================================================
+++ CONSTANT TABLE +++

TYPE OF CONSTANT                 VALUE OF CONSTANT
================                 =================
```

Implementation

Regular expression for identifiers: The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.
[a-z|A-Z]([a-z|A-Z]|[0-9])*

Multiline comments: This has been supported by checking the
occurence of ‘/*’ and ‘*/’ in the code. The statements between them have been excluded.

Errors for unmatched and nested comments have also been displayed. Error Handling for incomplete String: Open and close quote missing, both kinds of errors have been handled in the rules written in the script.

Error Handling for Nested Comments: This use-case has been handled by checking for occurrence of multiple successive ‘/*’ or ‘*/’ in the C code, and by omitting the text in between them.

At the end of the token recognition, the lexer prints a list of all the tokens present in the program. As and when successive tokens are encountered, their respective values are stored in the symbol table structure and then later displayed.

To test the above Lex Code we have created five test cases which have single and multi line comments, errors in them to test and verify the correct working for our program. From the results of our test cases we can clearly see that the program works as designed.

Future Work

The lexical analyser that was created in this project helps in breaking source code into tokens defined by the C programming language.

In the next phase, the parser will be designed which 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 table, 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.

References

Introduction to Lex https://www.javatpoint.com/lex
Stack Overflow https://stackoverflow.com/
Lex Analyser http://dinosaur.compilertools.net/lex/index.html