Submitted in partial fulfillment of the requirements, of course, CSE332: Compiler Design Lab, for the B.Sc program in CSE

Project Report

Report Title: Showing lexeme and generating lexeme symbol table

Submitted To:

- Mr. Md. Aynul Hasan Nahid
  Lecturer, Department of
  Computer Science and Engineering - CSE
  Daffodil International University-DIU

Submitted By:

- ✓ Md. Ikramul Hossain
  ID: 201-15-14091
  Section: A
  Department of Computer Science & Engineering - CSE
  Daffodil International University-DIU

Submission Date: **14-12-2022**

Objective:

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or code that a computer's processors use. The file used for writing a C-language contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. The output of the compilation is called object code or sometimes an object module.

Necessary Tools:

A comment is an annotation in a source code with the intention to give a programmer a readable explanation of the code. These annotations are ignored by compilers when compiling your code. Comments should explain, at a higher level of abstraction than the code, what you're trying to do. Following are the most important file management functions or tools are used in our programm.

function	→	**purpose**
fopen ()	→	Creating a file or opening an existing file
fclose ()	→	Closing a file
fprintf ()	→	Writing a block of data to a file
fscanf ()	→	Reading a block data from a file
getc ()	→	Reads a single character from a file
putc ()	→	Writes a single character to a file
getw ()	→	Reads an integer from a file
putw ()	→	Writing an integer to a file

Also use some tools such as codeblocks, input c program file and input-output file etc.

Statement:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. We have designed a lexical analyzer for the C language using lex. It takes as input a C code and outputs a stream of tokens. The tokens displayed as part of the output include keywords, identifiers, signed/unsigned integer/floating point constants, operators, special characters, headers, data-type specifiers, array, single-line comment, multi-line comment, preprocessor directive, pre-defined functions (printf and scanf), user-defined functions and the main function. The token, the type of token and the line number of the token in the C code are being displayed. The line number is displayed so that it is easier to debug the code for errors. Errors in single-line comments, multi-line comments are displayed along with line numbers. The output also contains the symbol table which contains tokens and their type. The symbol table is generated using the hash organisation.

Source code:

Lex Program:

```
%{
int lineno = 1;
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define AUTO 1
#define BREAK 2
#define CASE 3
#define CHAR 4
#define CONST 5
#define CONTINUE 6
#define DEFAULT 7
#define DO 8
#define DOUBLE 9
#define ELSE 10
#define ENUM 11
#define EXTERN 12
#define FLOAT 13
#define FOR 14
#define GOTO 15
#define IF 16
#define INT 17
#define LONG 18
#define REGISTER 19
#define RETURN 20
#define SHORT 21
#define SIGNED 22
#define SIZEOF 23
#define STATIC 24
#define STRUCT 25
#define SWITCH 26
#define TYPEDEF 27
#define UNION 28
#define UNSIGNED 29
#define VOID 30
#define VOLATILE 31
#define WHILE 32

#define IDENTIFIER 33
#define SLC 34
#define MLCS 35
#define MLCE 36

#define LEQ 37
#define GEQ 38
#define EQEQ 39
#define NEQ 40
#define LOR 41
```

```
#define LAND 42
#define ASSIGN 43
#define PLUS 44
#define SUB 45
#define MULT 46
#define DIV 47
#define MOD 48
#define LESSER 49
#define GREATER 50
#define INCR 51
#define DECR 52

#define COMMA 53
#define SEMI 54

#define HEADER 55
#define MAIN 56

#define PRINTF 57
#define SCANF 58
#define DEFINE 59

#define INT_CONST 60
#define FLOAT_CONST 61

#define TYPE_SPEC 62

#define DQ 63

#define OBO 64
#define OBC 65
#define CBO 66
#define CBC 67
#define HASH 68

#define ARR 69
#define FUNC 70

#define NUM_ERR 71
#define UNKNOWN 72

#define CHAR_CONST 73
#define SIGNED_CONST 74
#define STRING_CONST 75
%}

alpha [A-Za-z]
digit [0-9]
und [_]
space [ ]
tab [	]
line [\n]
char \'.\'
at [@]
string \"(.^([%d]|[%f]|[%s]|[%c]))\"
```

```
%%
{space}* {}
{tab}* {}
{string} return STRING_CONST;
{char} return CHAR_CONST;
{line} {lineno++;}
auto return AUTO;
break return BREAK;
case return CASE;
char return CHAR;
const return CONST;
continue return CONTINUE;
default return DEFAULT;
do return DO;
double return DOUBLE;
else return ELSE;
enum return ENUM;
extern return EXTERN;
float return FLOAT;
for return FOR;
goto return GOTO;
if return IF;
int return INT;
long return LONG;
register return REGISTER;
return return RETURN;
short return SHORT;
signed return SIGNED;
sizeof return SIZEOF;
static return STATIC;
struct return STRUCT;
switch return SWITCH;
typedef return TYPEDEF;
union return UNION;
unsigned return UNSIGNED;
void return VOID;
volatile return VOLATILE;
while return WHILE;

printf return PRINTF;
scanf return SCANF;

{alpha}({alpha}|{digit}|{und})* return IDENTIFIER;

[+-][0-9]{digit}*(\.{digit}+)? return SIGNED_CONST;

"//" return SLC;
"/*" return MLCS;
"*/" return MLCE;

"<=" return LEQ;
">=" return GEQ;
"==" return EQEQ;
"!=" return NEQ;
```

```
"||" return LOR;
"&&" return LAND;
"=" return ASSIGN;
"+" return PLUS;
"-" return SUB;
"*" return MULT;
"/" return DIV;
"%" return MOD;
"<" return LESSER;
">" return GREATER;
"++" return INCR;
"--" return DECR;

"," return COMMA;
";" return SEMI;

"#include<stdio.h>" return HEADER;
"#include <stdio.h>" return HEADER;
"main()" return MAIN;

{digit}+ return INT_CONST;
({digit}+)\.({digit}+) return FLOAT_CONST;

"%d"|"%f"|"%u"|"%s" return TYPE_SPEC;
"\"" return DQ;
"(" return OBO;
")" return OBC;
"{" return CBO;
"}" return CBC;
"#" return HASH;

{alpha}({alpha}|{digit}|{und})*\[{digit}*\] return ARR;
{alpha}({alpha}|{digit}|{und})*\(({alpha}|{digit}|{und}|{space})*\) return FUNC;
({digit}+)\.({digit}+)\.({digit}|\.)* return NUM_ERR;
({digit}|{at})+({alpha}|{digit}|{und}|{at})* return UNKNOWN;
%%

struct node
{
        char token[100];
        char attr[100];
        struct node *next;
};

struct hash
{
        struct node *head;
        int count;
};

struct hash hashTable[1000];
int eleCount = 1000;

struct node * createNode(char *token, char *attr)
{
```

```
	struct node *newnode;
	newnode = (struct node *) malloc(sizeof(struct node));
	strcpy(newnode->token, token);
	strcpy(newnode->attr, attr);
	newnode->next = NULL;
	return newnode;
}

int hashIndex(char *token)
{
	int hi=0;
	int l,i;
	for(i=0;token[i]!='\0';i++)
	{
		hi = hi + (int)token[i];
	}
	hi = hi%eleCount;
	return hi;
}

void insertToHash(char *token, char *attr)
{
	int flag=0;
	int hi;
	hi = hashIndex(token);
	struct node *newnode = createNode(token, attr);
	/* head of list for the bucket with index "hashIndex" */
	if (hashTable[hi].head==NULL)
	{
		hashTable[hi].head = newnode;
	    hashTable[hi].count = 1;
	    return;
	}
	struct node *myNode;
	myNode = hashTable[hi].head;
	while (myNode != NULL)
	{
	if (strcmp(myNode->token, token)==0)
	{
		flag = 1;
		break;
	}
	myNode = myNode->next;
	}
	if(!flag)
	{
		//adding new node to the list
		newnode->next = (hashTable[hi].head);
		//update the head of the list and no of nodes in the current bucket
		hashTable[hi].head = newnode;
		hashTable[hi].count++;
	}
	return;
}
```

```
void display()
{
	struct node *myNode;
	int i,j, k=1;
	printf("-------------------------------------------------------------------");
	printf("\nSNo \t|\tToken \t\t|\tToken Type \t\n");
    printf("-------------------------------------------------------------------\n");
	for (i = 0; i < eleCount; i++)
	{
	if (hashTable[i].count == 0)
		continue;
	myNode = hashTable[i].head;
	if (!myNode)
		continue;
	while (myNode != NULL)
	{
		printf("%d\t\t", k++);
		printf("%s\t\t\t", myNode->token);
		printf("%s\t\n", myNode->attr);
		myNode = myNode->next;
	}
	}
	return;
}

int main()
{
	int scan, slcline=0, mlc=0, mlcline=0, dq=0, dqline=0;
	yyin = fopen("input.c","r");
	printf("\n\n");
	scan = yylex();
	while(scan)
	{
		if(lineno == slcline)
		{
			scan = yylex();
			continue;
		}
		if(lineno!=dqline && dqline!=0)
		{
			if(dq%2!=0)
				printf("\n******** ERROR!! INCOMPLETE STRING at Line %d ********\n\n",
dqline);
			dq=0;
		}
		if((scan>=1 && scan<=32) && mlc==0)
		{
			printf("%s\t\t\tKEYWORD\t\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "KEYWORD");
		}
		if(scan==33 && mlc==0)
		{
			printf("%s\t\t\tIDENTIFIER\t\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "IDENTIFIER");
		}
```

```
		if(scan==34)
		{
			printf("%s\t\t\tSingleline Comment\t\tLine %d\n", yytext, lineno);
			slcline = lineno;
		}
		if(scan==35 && mlc==0)
		{
			printf("%s\t\t\tMultiline Comment Start\t\tLine %d\n", yytext, lineno);
			mlcline = lineno;
			mlc = 1;
		}
		if(scan==36 && mlc==0)
		{
			printf("\n******** ERROR!! UNMATCHED MULTILINE COMMENT END %s at Line
%d ********\n\n", yytext, lineno);
		}
		if(scan==36 && mlc==1)
		{
			mlc = 0;
			printf("%s\t\t\tMultiline Comment End\t\tLine %d\n", yytext, lineno);
		}
		if((scan>=37 && scan<=52) && mlc==0)
		{
			printf("%s\t\t\tOPERATOR\t\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "OPERATOR");
		}
		if((scan==53||scan==54||scan==63||(scan>=64 && scan<=68)) && mlc==0)
		{
			printf("%s\t\t\tSPECIAL SYMBOL\t\t\tLine %d\n", yytext, lineno);
			if(scan==63)
			{
				dq++;
				dqline = lineno;
			}
			insertToHash(yytext, "SPECIAL SYMBOL");
		}
		if(scan==55 && mlc==0)
		{
			printf("%s\tHEADER\t\t\t\tLine %d\n",yytext, lineno);
		}
		if(scan==56 && mlc==0)
		{
			printf("%s\t\t\tMAIN FUNCTION\t\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "IDENTIFIER");
		}
		if((scan==57 || scan==58) && mlc==0)
		{
			printf("%s\t\t\tPRE DEFINED FUNCTION\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "PRE DEFINED FUNCTION");
		}
		if(scan==59 && mlc==0)
		{
			printf("%s\t\t\tPRE PROCESSOR DIRECTIVE\t\tLine %d\n", yytext, lineno);
		}
		if(scan==60 && mlc==0)
```

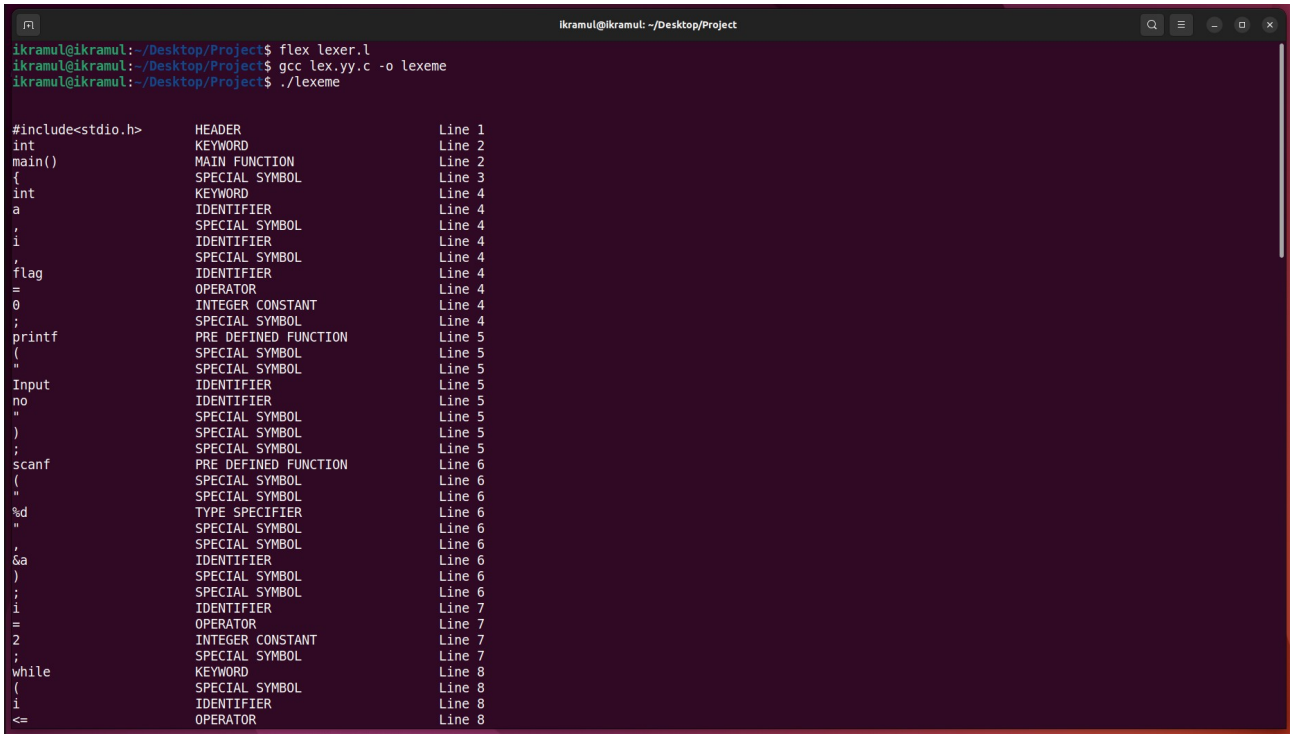
```
		{
			printf("%s\t\t\tINTEGER CONSTANT\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "INTEGER CONSTANT");
		}
		if(scan==61 && mlc==0)
		{
			printf("%s\t\t\tFLOATING POINT CONSTANT\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "FLOATING POINT CONSTANT");
		}
		if(scan==62 && mlc==0)
		{
			printf("%s\t\t\tTYPE SPECIFIER\t\t\tLine %d\n", yytext, lineno);
		}
		if(scan==69 && mlc==0)
		{
			printf("%s\t\t\tARRAY\t\t\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "ARRAY");
		}
		if(scan==70 && mlc==0)
		{
			printf("%s\t\t\tUSER DEFINED FUNCTION\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "USER DEFINED FUNCTION");
		}
		if(scan==71 && mlc==0)
		{
			printf("\n******** ERROR!! CONSTANT ERROR %s at Line %d ********\n\n", yytext,
lineno);
		}
		if(scan==72 && mlc==0)
		{
			printf("\n******** ERROR!! UNKNOWN TOKEN %s at Line %d ********\n\n", yytext,
lineno);
		}
		if(scan==73 && mlc==0)
		{
			printf("%s\t\t\tCHARACTER CONSTANT\t\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "CHARACTER CONSTANT");
		}
		if(scan==74 && mlc==0)
		{
			printf("%s\t\t\tSIGNED CONSTANT\t\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "SIGNED CONSTANT");
		}
		if(scan==75 && mlc==0)
		{
			printf("%s\t\t\tSTRING CONSTANT\t\t\tLine %d\n", yytext, lineno);
			insertToHash(yytext, "STRING CONSTANT");
		}
		scan = yylex();
	}
	if(mlc==1)
		printf("\n******** ERROR!! UNMATCHED COMMENT STARTING at Line %d ********\n\
n",mlcline);
	printf("\n");
	printf("\n\t******** SYMBOL TABLE ********\t\t\n");
```

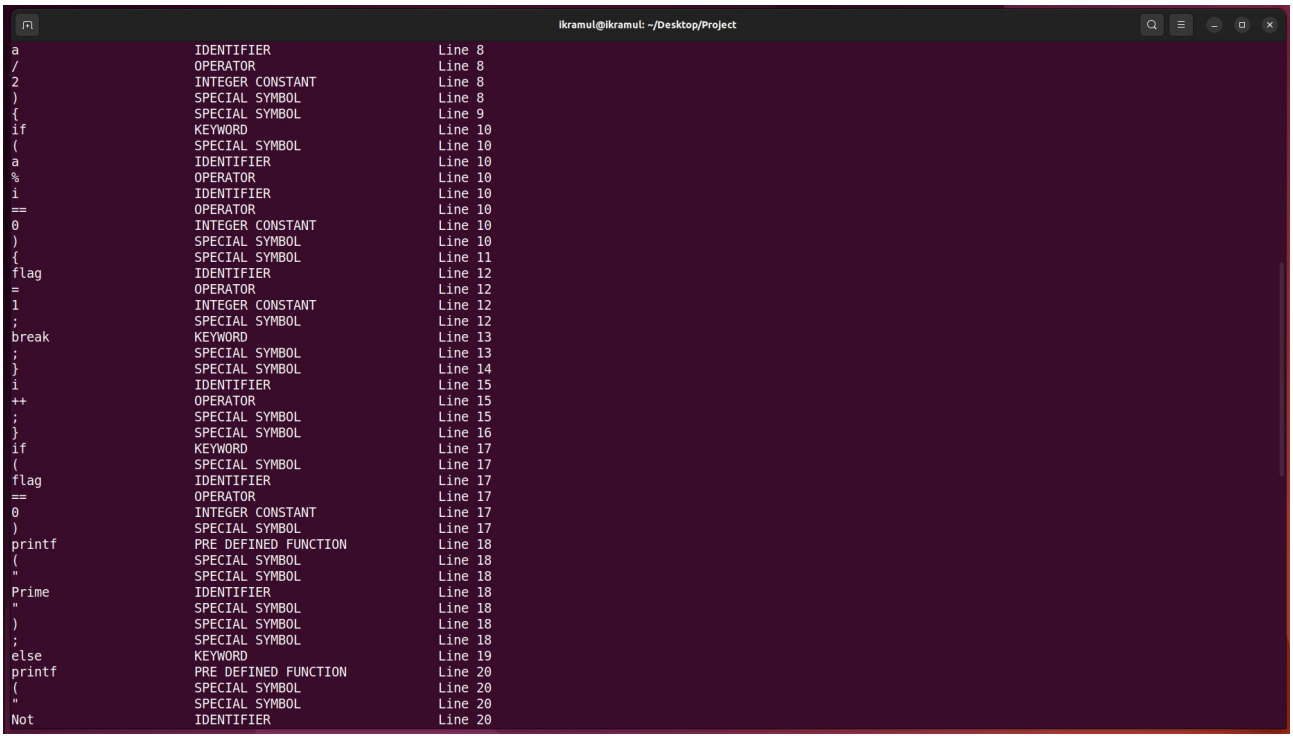
```
            display();
        printf("------------------------------------------------------------------\n\n");
}
int yywrap()
{
            return 1;
}
```

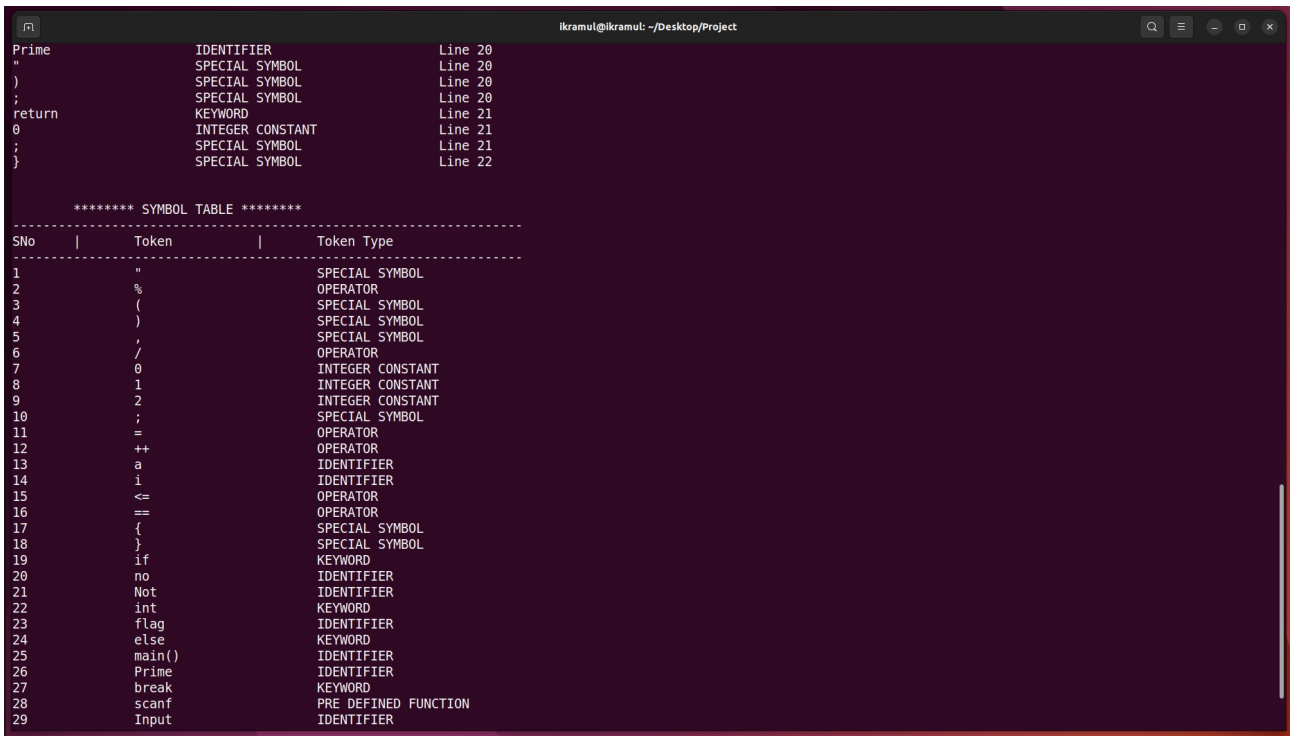
Input File:

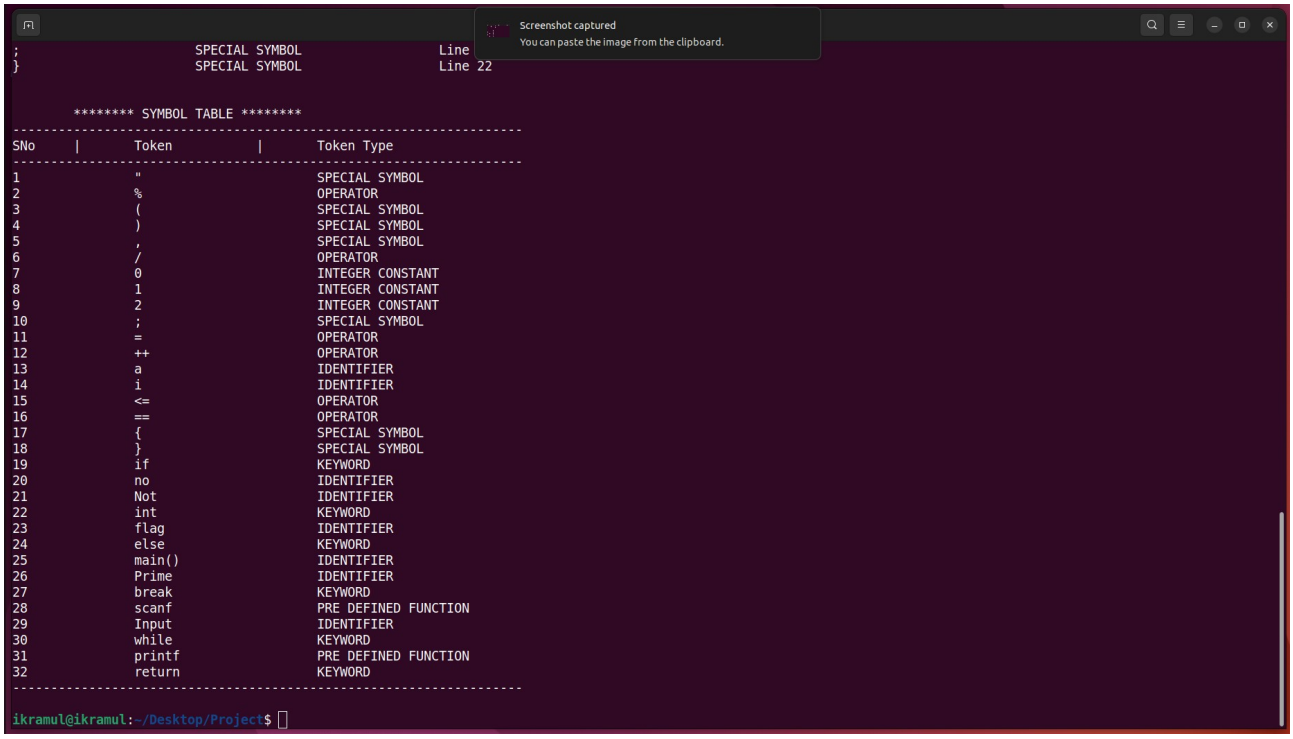
```
#include<stdio.h>
int main()
{
   int a,i,flag=0;
   printf("Input no");
   scanf("%d",&a);
   i=2;
   while(i <= a/2)
   {
      if(a%i == 0)
      {
         flag=1;
         break;
      }
      i++;
   }
   if(flag==0)
      printf("Prime");
   else
      printf("Not Prime");
   return 0;
}
```

Output:


ikramul@ikramul: ~/Desktop/Project
ikramul@ikramul:~/Desktop/Project$ flex lexer.l
ikramul@ikramul:~/Desktop/Project$ gcc lex.yy.c -o lexeme
ikramul@ikramul:~/Desktop/Project$ ./lexeme
#include<stdio.h> HEADER Line 1
int KEYWORD Line 2
main() MAIN FUNCTION Line 2
{ SPECIAL SYMBOL Line 3
int KEYWORD Line 4
a IDENTIFIER Line 4
, SPECIAL SYMBOL Line 4
i IDENTIFIER Line 4
, SPECIAL SYMBOL Line 4
flag IDENTIFIER Line 4
= OPERATOR Line 4
0 INTEGER CONSTANT Line 4
; SPECIAL SYMBOL Line 4
printf PRE DEFINED FUNCTION Line 5
(SPECIAL SYMBOL Line 5
" SPECIAL SYMBOL Line 5
Input IDENTIFIER Line 5
no IDENTIFIER Line 5
" SPECIAL SYMBOL Line 5
) SPECIAL SYMBOL Line 5
; SPECIAL SYMBOL Line 5
scanf PRE DEFINED FUNCTION Line 6
(SPECIAL SYMBOL Line 6
" SPECIAL SYMBOL Line 6
%d TYPE SPECIFIER Line 6
" SPECIAL SYMBOL Line 6
, SPECIAL SYMBOL Line 6
&a IDENTIFIER Line 6
) SPECIAL SYMBOL Line 6
; SPECIAL SYMBOL Line 6
i IDENTIFIER Line 7
= OPERATOR Line 7
2 INTEGER CONSTANT Line 7
; SPECIAL SYMBOL Line 7
while KEYWORD Line 8
(SPECIAL SYMBOL Line 8
i IDENTIFIER Line 8
<= OPERATOR Line 8


ikramul@ikramul: ~/Desktop/Project
a IDENTIFIER Line 8
/ OPERATOR Line 8
2 INTEGER CONSTANT Line 8
) SPECIAL SYMBOL Line 8
{ SPECIAL SYMBOL Line 9
if KEYWORD Line 10
(SPECIAL SYMBOL Line 10
a IDENTIFIER Line 10
% OPERATOR Line 10
i IDENTIFIER Line 10
== OPERATOR Line 10
0 INTEGER CONSTANT Line 10
) SPECIAL SYMBOL Line 10
{ SPECIAL SYMBOL Line 11
flag IDENTIFIER Line 12
= OPERATOR Line 12
1 INTEGER CONSTANT Line 12
; SPECIAL SYMBOL Line 12
break KEYWORD Line 13
; SPECIAL SYMBOL Line 13
} SPECIAL SYMBOL Line 14
i IDENTIFIER Line 15
++ OPERATOR Line 15
; SPECIAL SYMBOL Line 15
} SPECIAL SYMBOL Line 16
if KEYWORD Line 17
(SPECIAL SYMBOL Line 17
flag IDENTIFIER Line 17
== OPERATOR Line 17
0 INTEGER CONSTANT Line 17
) SPECIAL SYMBOL Line 17
printf PRE DEFINED FUNCTION Line 18
(SPECIAL SYMBOL Line 18
" SPECIAL SYMBOL Line 18
Prime IDENTIFIER Line 18
" SPECIAL SYMBOL Line 18
) SPECIAL SYMBOL Line 18
; SPECIAL SYMBOL Line 18
else KEYWORD Line 19
printf PRE DEFINED FUNCTION Line 20
(SPECIAL SYMBOL Line 20
" SPECIAL SYMBOL Line 20
Not IDENTIFIER Line 20





Conclusion:

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

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