

# LEXICAL ANALYZER FOR C-LANGUAGE



NATIONAL INSTITUTE OF TECHNOLOGY – KARNATAKA, SURATHKAL.

**Date :** 12<sup>th</sup> August, 2020

**Submitted to :** P. Santhi Thilagam

**Team Members:** Sai Nishanth K R(181CO145)  
Srikrishna G Yaji(181COC153)  
Shashank D(181CO248)  
Tarun S Anur(181CO255)

# **ABSTRACT**

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 processor uses.

## **Phases of Compiler:**

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.

The phases are given as below:

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Intermediate Code Generation (ICG)

## **Tokens:**

The following tokens will be detected by the Lexical Analyzer:

1. Data Types: int, char, float, double, struct etc.
2. Comments: Single-line and Multi-line comments,
3. Keywords: for, while, return, break, register, static etc.
4. Valid Identifiers used in the program
5. Operators: +, \*, /, %, &, |
6. Special Symbols: ;, :, {, }, [, ] etc.

## **INDEX**

SL. NO	TOPIC	PG. NO
1	INTRODUCTION	4
2	CODE	4
3	EXPLANATION	5
4	SAMPLE PROGRAMS	12
5	ERROR HANDLING	16
6	FUTURE DEVELOPMENTS	16
7	REFERENCES	17

## **LIST OF FIGURES**

SL. NO	TOPIC	PG. NO
1	CODE	5-11
2	SAMPLE PROGRAM 1	13
3	SAMPLE OUTPUT 1	14
4	SAMPLE PROGRAM 2	15
5	SAMPLE OUTPUT 2	15

# **INTRODUCTION**

## **Lexical analysis:**

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 analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

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

## **Longest Match Rule:**

When the lexical analyser read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed. The lexical analyser also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyser finds a lexeme that matches with any existing reserved word, it should generate an error.

## **Lexemes:**

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyser as an instance of that token.

## CODE:

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

struct symbol_table {
    char type[100];
    char name[100];
    int valid;
}st[500];

struct constant_table {
    char type[100];
    char name[100];
    int valid;
}ct[500];

int hash_function(char *str)
{
    int value = 0;
    for(int i = 0 ; i < strlen(str) ; i++)
    {
        value = 10*value + (str[i] - 'A');
        value = value % 501;
        while(value < 0)
            value = value + 501;
    }
    return value;
}
int lookup_symbolTable(char *str)
{
    int value = hash_function(str);
    if(st[value].valid == 0)
    {
        return 0;
    }
    else if(strcmp(st[value].name,str)==0)
    {
        return 1;
    }
    else
    {
        for(int i = value + 1 ; i!=value ; i = (i+1)%501)
        {
            if(strcmp(st[i].name,str)==0)
            {
                return 1;
            }
        }
    }
}
```

```

        }
    }
    return 0;
}

int lookup_constantTable(char *str)
{
    int value = hash_function(str);
    if(ct[value].valid == 0)
        return 0;
    else if(strcmp(ct[value].name, str) == 0)
        return 1;
    else
    {
        for(int i = value + 1 ; i != value ; i = (i+1)%501)
        {
            if(strcmp(ct[i].name, str) == 0)
            {
                return 1;
            }
        }
        return 0;
    }
}

void insert_symbolTable(char *str1, char *str2)
{
    if(lookup_symbolTable(str1))
    {
        return;
    }
    else
    {
        int value = hash_function(str1);
        if(st[value].valid == 0)
        {
            strcpy(st[value].name, str1);
            strcpy(st[value].type, str2);
            st[value].valid = strlen(str1);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i != value ; i = (i+1)%501)
        {
            if(st[i].valid == 0)
            {
                pos = i;
            }
        }
    }
}

```

```

        break;
    }
}

strcpy(st[pos].name, str1);
strcpy(st[pos].type, str2);
st[pos].valid = strlen(str1);
}
}

void insert_constantsTable(char *str1, char *str2)
{
    if(lookup_constantTable(str1))
        return;
    else
    {
        int value = hash_function(str1);
        if(ct[value].valid == 0)
        {
            strcpy(ct[value].name, str1);
            strcpy(ct[value].type, str2);
            ct[value].valid = strlen(str1);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%501)
        {
            if(ct[i].valid == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(ct[pos].name, str1);
        strcpy(ct[pos].type, str2);
        ct[pos].valid = strlen(str1);
    }
}

int cbracketsopen = 0;
int cbracketsclose = 0;
int bbracketsopen = 0;
int bbracketsclose = 0;
int fbracketsopen = 0;
int fbracketsclose = 0;
%}

```





```

}
("\\")(\\s|{identifier}|{numerical_constants}|{operators})* {
    printf("In LineNo: %d, ERROR: String usage error in %s\n", yylineno, yytext); exit(1);
}

[\\n\\t ]*\\'(\\s|{identifier}|{numerical_constants}|{operators})* {
    printf("In LineNo: %d, ERROR: Charecter usage error", yylineno); exit(1);
}

{identifier} {
    printf("%s is a identifier\n", yytext);
    insert_symbolTable(yytext, "Identifier");
}

{constants} {
    printf("%s is a constant\n", yytext);
    insert_constantsTable(yytext, "Constant");
    for(int i = 0; i < strlen(yytext); i++) {
        if(yytext[i] == '\\n') yylineno++;
    }
}

{special_symbols} {
    printf("%s is a special symbol\n", yytext);
    if(yytext[0] == '{'){
        fbracketsopen++;
    }
    else if(yytext[0] == '}{
        fbracketsclose++;
    }
    else if(yytext[0] == '('){
        cbracketsopen++;
    }
    else if(yytext[0] == ')'){
        cbracketsclose++;
    }
    else if(yytext[0] == '['){
        bbracketsopen++;
    }
    else if(yytext[0] == ']'){
        bbracketsclose++;
    }
}

{unary_operators}|{special_operators}|{logical_operators}|{relational_operators}|{bitwise_operators}|{binary_operators}|{assignment_operators} {
    printf("%s is an operator\n", yytext);
}

```

```
%%

int main(int argc, char** argv){
    if(argc < 2){
        puts("ERROR: input file unavailable");
        exit(1);
    }
    yyin = fopen(argv[1], "r");

    int i,j;

    for (j=0;j<500;j++)
    {
        st[j].valid = 0;
        ct[j].valid = 0;
    }

    yylex();
    if((bbracketsopen-bbracketsclose)){
        printf("ERROR: brackets error [");
        exit(1);
    }
    if((fbracketsopen-fbracketsclose)){
        printf("ERROR: brackets error {");
        exit(1);
    }
    if((cbracketsopen-cbracketsclose)){
        printf("ERROR: brackets error (");
        exit(1);
    }
    printf("\nSymbol Table\n");
    for (i=0;i<500;i++)
    {
        if(st[i].valid)
        {
            printf("Name : %s \t", st[i].name);
            printf("Type : %s \n", st[i].type);
        }
    }

    printf("\nConstant Table\n");
    for (int i=0;i<500;i++)
    {
        if(ct[i].valid)
        {
            printf("Name : %s \t", ct[i].name);
            printf("Type : %s \n", ct[i].type);
        }
    }
}
```

```
    }  
    }  
    fclose(yyin);  
    return (0);  
}  
  
int yywrap(){  
    return(1);  
}
```

FIGURE 1: code for the lexical analyser

## **EXPLANATION:**

### **Declaration Section:**

This is the part of the program which includes all the header files required for the program to execute. This section also includes the definition of any user defined functions used in the program and also all the global variables and constructs required by the program. The hash values of all the constants and the symbols were generated and stored in the constant table and the symbol tables respectively. Linear Probing hashing technique is followed in case of any collisions in the hash table. Functions to insert and print the symbol tables and the constant tables are also declared in this section.

### **Rules Section:**

This part of the program contains all the rules related to the lexical analysis of the programming language. Regular expressions are used to detect matching grammar of the programming construct. Each of the programming construct has to match with one of the regular expressions, so if it doesn't match with any of them, then the program throws an error to the user. Here we have also printed out what type of token each construct is detected as and also the detected token.

### **User-routines Section:**

This part of program contains the main function in which we open a file to be scanned. The program scans the file after it calls the `yylex()` function and corresponding results are stored in the constant and the symbol tables. The program also prints the final Symbol and Constant table.

# SAMPLE PROGRAMS

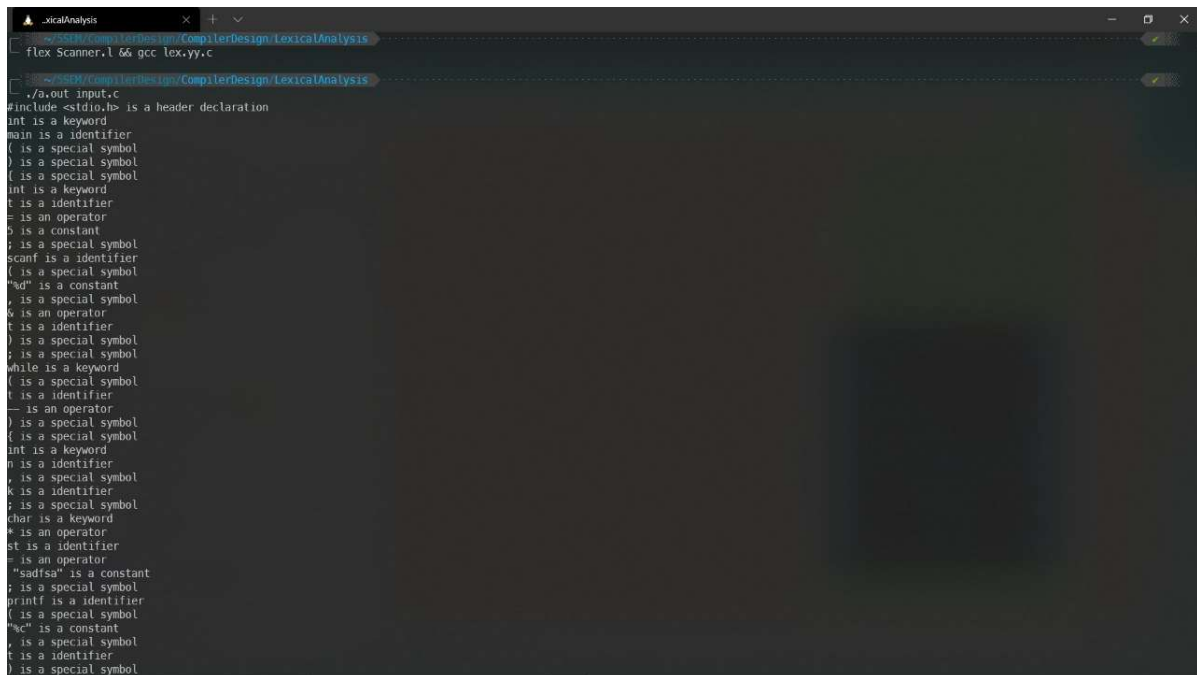
Sample Program 1(error free):



```
//#pragma GCC optimize "trapv"
#include <stdio.h>
/*
    comment
    comment
*/
int main() {
    //fios;
    int t = 5;
    scanf("%d", &t);
    while(t--) {
        int n, k;
        char* st = "samplestring";
        printf("%c", t);
        scanf("%d %d", &n, &k);
        if(n >= k) {
            if((n-k) % 2 == 1) {
                puts("1");
            } else {
                puts("0");
            }
        } else {
            printf("%d\n", k - n);
        }
    }
}
```

FIGURE 2: Sample program 1

## Sample Output 1:

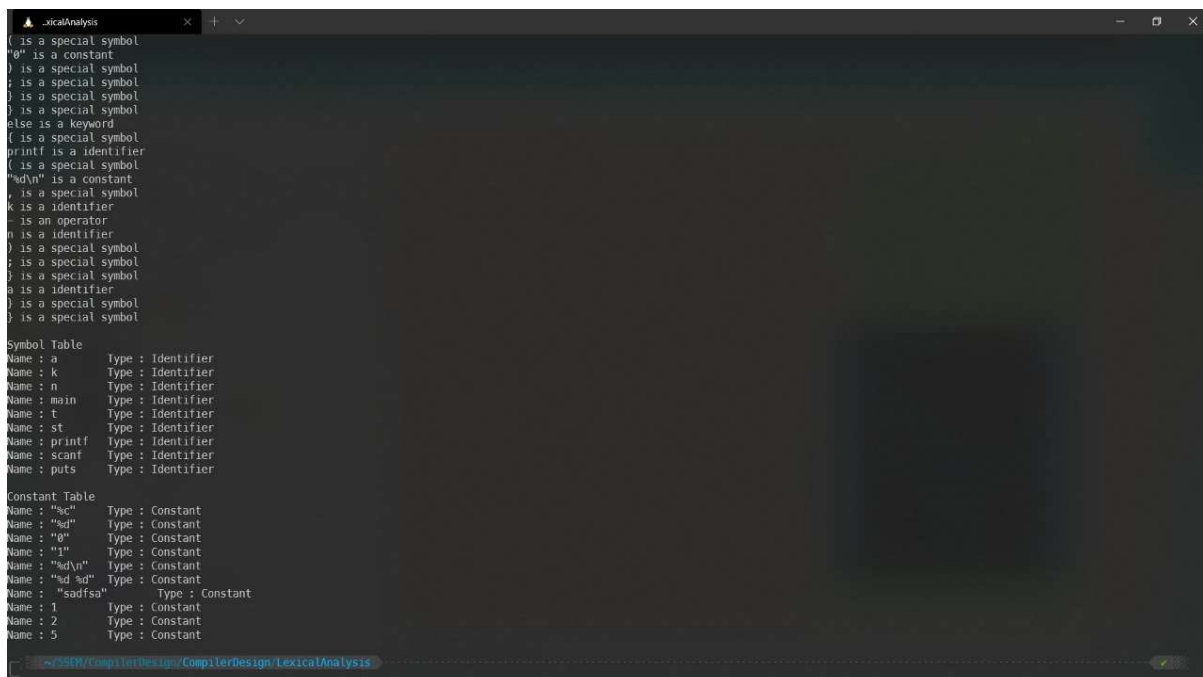


```

./a.out input.c
#include <stdio.h> is a header declaration
int is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
t is a identifier
= is an operator
5 is a constant
; is a special symbol
scanf is a identifier
( is a special symbol
"%d" is a constant
, is a special symbol
& is an operator
t is a identifier
) is a special symbol
; is a special symbol
while is a keyword
( is a special symbol
t is a identifier
-- is an operator
) is a special symbol
{ is a special symbol
int is a keyword
n is a identifier
, is a special symbol
k is a identifier
; is a special symbol
char is a keyword
* is an operator
st is a identifier
= is an operator
"sadfsa" is a constant
; is a special symbol
printf is a identifier
( is a special symbol
"%c" is a constant
, is a special symbol
t is a identifier
) is a special symbol

```

FIGURE 3(a): Sample output 1



```

( is a special symbol
"0" is a constant
) is a special symbol
; is a special symbol
) is a special symbol
) is a special symbol
else is a keyword
( is a special symbol
printf is a identifier
( is a special symbol
"%d\n" is a constant
, is a special symbol
k is a identifier
- is an operator
n is a identifier
) is a special symbol
; is a special symbol
) is a special symbol
a is a identifier
) is a special symbol
) is a special symbol


Symbol Table
Name : a      Type : Identifier
Name : k      Type : Identifier
Name : n      Type : Identifier
Name : main   Type : Identifier
Name : t      Type : Identifier
Name : st     Type : Identifier
Name : printf Type : Identifier
Name : scanf  Type : Identifier
Name : puts   Type : Identifier

Constant Table
Name : "%c"   Type : Constant
Name : "%d"   Type : Constant
Name : "%g"   Type : Constant
Name : "%i"   Type : Constant
Name : "%d\n" Type : Constant
Name : "%d %d" Type : Constant
Name : "sadfsa" Type : Constant
Name : 1      Type : Constant
Name : 2      Type : Constant
Name : 5      Type : Constant

```

FIGURE 3(b): Sample output 1

Sample Program 2(with error):



```
#include<stdio.h>

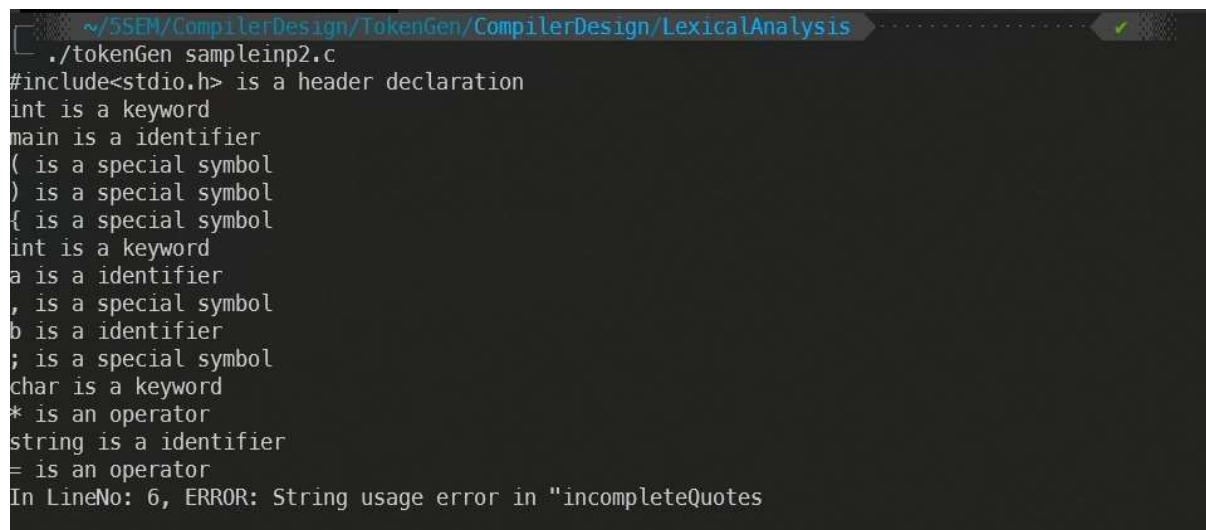
int main() {
    int a, b;

    char *string = "incompleteQuotes;

    return 0;
}
```

FIGURE 4: Sample Program 2

Sample Output 2:



```
~/5SEM/CompilerDesign/TokenGen/CompilerDesign/LexicalAnalysis
./tokenGen sampleinp2.c
#include<stdio.h> is a header declaration
int is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
a is a identifier
, is a special symbol
b is a identifier
; is a special symbol
char is a keyword
* is an operator
string is a identifier
= is an operator
In LineNo: 6, ERROR: String usage error in "incompleteQuotes
```

FIGURE 5: Sample output 2

## **ERROR HANDLING**

The program throws the following type of errors along with the line number of the error code:

1. Identifier error: The program generates an error if it recognizes any numerical constant in front of an identifier and throws an error.  
The regular expression declared which matches both the numerical constant and identifier matched together throws the error.
2. String error: The program throws an error if it doesn't match either the beginning or ending of a double quotes (").  
The regular expression declared with only one of the double quotes and any sequence of characters following or preceding them throws this error.
3. Character error: The program throws an error if it doesn't match either the beginning or ending of single quotes (').  
The regular expression declared with only one of the single quotes and any sequence of characters following or preceding them throws this error.
4. Brackets error: The program also throws an error in case of unbalanced opening and closing brackets (, {, [, ], }, ). Variables declared for each opening and closing brackets are checked for equality. The program throws an error in case of each of unbalanced brackets.

## **FUTURE DEVELOPMENTS**

1. The line number for the bracket error isn't being recognized by the scanner.



## **BIBLIOGRAPHY**

1. Wikipedia
2. <https://www.lexico.com>