Department of Computer Science Engineering – 6th Semester

COMPILER DESIGN LAB:

LEXICAL ANALYZER FOR A C COMPILER

Submitted by:

Shreyas G 11CO88

Sourabh S 11CO92

Contents

Brief Introduction of the Lexical Analyzer

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. The code for lexical analysis is quite simple and involves the identification of tokens in the program.

A **token** is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, and COMMA). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parenthesis as tokens, but does nothing to ensure that each '(' is matched with a ')'.

Consider this expression in the C programming language,

Sum = 3 + 2;

The program runs through this statement using space as a delimiter to identify different parts. Regular expressions are listed for each type of operator and identifier and the program is designed to output the name of that type. A list of tokens identified by the scanner is given in Section 3.

Lexeme	Token type
sum	Variable encountered
=	Assignment operator encountered
3	Integer encountered
+	Arithmetic operator encountered
2	Integer encountered
;	Special Symbol

List of Tokens that the Scanner Recognizes and What they mean

C Language is a huge language with large no. of constructs. Today also C itself occupies a very large portion of the Programming Environment. Features which have been implemented by us in our 'C' Compiler are very general features of the language. Almost all the basic features are implemented and the advanced features are tried to be implemented to a certain extent.

The List of implemented features is as follows:

1) The C Character Set: A character denotes any alphabet, digit or special symbol used to represent information.

• Digits	0,1,2,3,4,5,6,7,8,9
• Alphabets	A, B… X, Y, Z OR a, b… x, y, z
• Special Symbols	{, }, [,], :, ;, ", ', !, %, &, *, (,), -, _, +, =, <, >, ., /

2) Keywords: Every C word is classified as either a keyword or an identifier. The keywords cannot be used as variable names. Following is the list of keywords in our C Compiler.

ANSI C Keywords: 'break', 'case', 'char', 'continue', 'default', 'do', 'else', 'for', 'goto', 'if', 'int', 'return', 'short', 'struct', 'switch', union', void', 'while'

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

3) Constants: A constant is a quantity that doesn't change. We have implemented int and float constants.

4) Variables (or) identifiers: These are the named locations in memory that are used to store values that may be modified by the program. These locations can contain integer or float constants. A variable can be initialized with a constant at the time of declaration.

5) Data Types and Sizes: There are basically four data types in C. char, int, float and double. We have implemented int & float data types. A float type is used to hold real numbers and occupies four bytes. An int requires 2 bytes and store integer numbers.

6) Comment Lines: We have implemented two types of comments.

7) Single line comment: Anything followed by '//' up to the end of the Line is treated as comment and ignored by the compiler.

8) Multiple line comment: Like C language every characters between '/*' and '*/' is a comment. Compiler ignores everything beginning with '/*' up to First '*/'. It does not support nested comments (comment within other comment).

9) Statement Terminator: Statement in input program must be terminated with a semicolon. The program can also have carriage return, white space or tabs as many or as few as desired.

10) Arithmetic Operators: We have implemented following 4 basic arithmetic operators.+ Addition (plus sign), - Subtraction (minus sign), * Multiplication (asterisk), / Division (forward slash).

We have implemented only Integer Arithmetic.

11) Relational Operator: We have implemented the following relational operators.

- < is less than
- <= is less than or equal to
- > is greater than
- >= is greater than or equal to
- == is equal to
- != is not equal to.

12) Logical Operators: We have implemented all the three logical operators present in C, namely:

- && meaning logical AND
- || meaning logical OR
- ! meaning logical NOT.

13) Shorthand Operators: Shorthand operators are of the form:

V op = exp;

Where V is a variable, exp is an expression and op is a C binary arithmetic operator. The operator op= is known as the shorthand assignment operator. We have implemented all types of short hand operators (‘+’,’-‘,’*’,’/’).

14) Increment and Decrement Operators: The operator ++ adds 1 to the operand while — subtracts 1. Both are unary operators and take the following forms:

- ++m; or m++;
- --m; or m--;

We have implemented both of these operators that can be used in any type of statement.

15) Assignment Operator: In C assignment operator can be used with any valid C expression. The form of the assignment statement is variable name = expression; where, an expression can be simple as constant or a complex expression. The target or the left part of the assignment must not be a function or a constant.

16) Operator Precedence & Associativity of Operators: Precedence of operators in our compiler is:

/ > * > + > -

Associativity of all these operators is from left to right

17) The IF Statement: A if statement with only one condition is called a simple if statement. The form of the if statement is as follows:

```
If (expression)

{   Statement list;

} statement x;
```

The keyword 'if' must be followed by a set of parentheses containing the expression to be tested. Expression may be a single condition or many conditions connected by using '&&' and '||'.

Form of condition: arithmetic_exp rel_op arithmetic_exp

Example: a == b; a +b! = c; a+b >= c+d;

Expression: condition1 && condition2; condtion1 || condition2;

The statement block may be a single statement or a group of statements.

18) The If...else Statement: The form of the if...else statement.

The form of the if...else statement is as follows:

```
if (expression)

{   true block statement(s);

}

else

{false block statement(s);

} statement x;
```

19) Nesting of if…else Statement: We have implemented Nesting of if statements which supports a if statement within another if or else. An else statement always refers to the nearest if statement that is within the same block as the else and is not already associated with a if statement

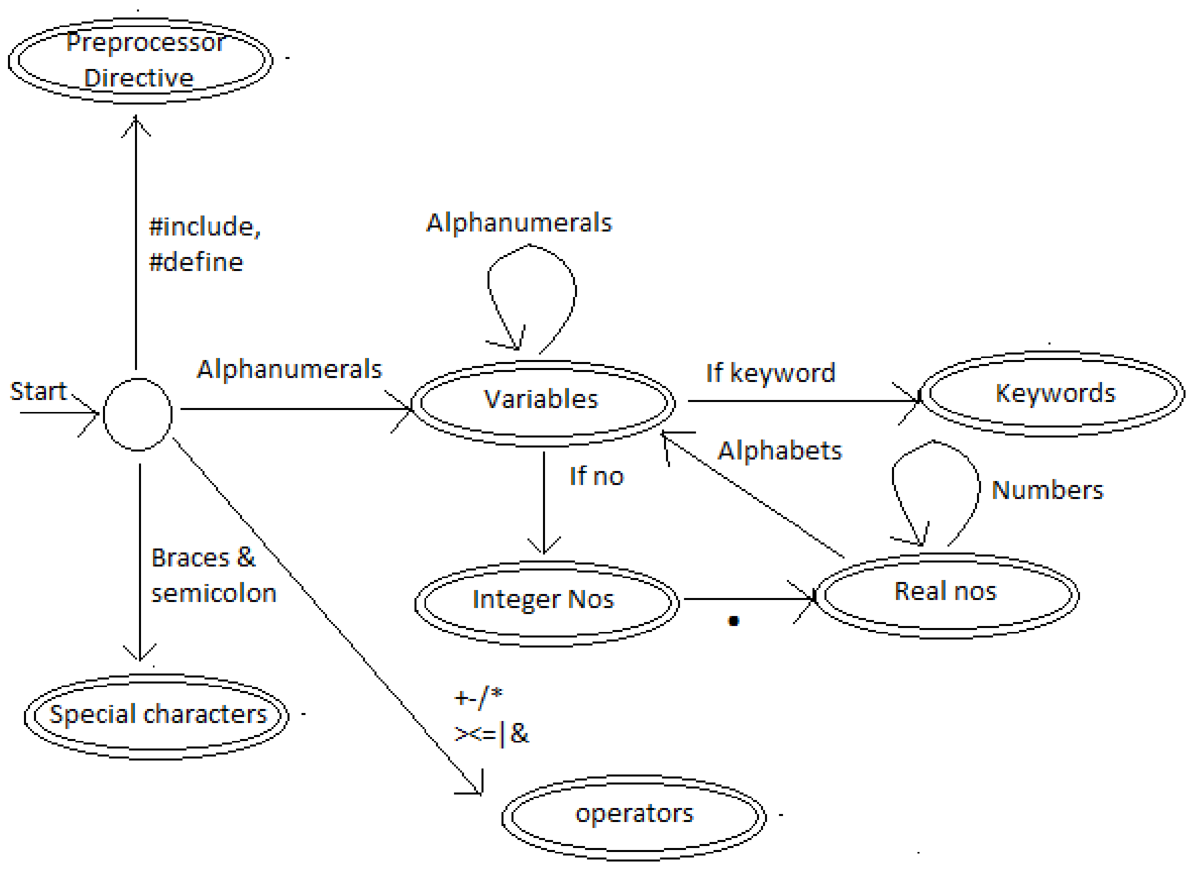
20) Loop Control Structures: The C language provides three loop constructs for performing loop operations. They are:

while loop, do…while loop and for loop.

We have implemented the while, do...while and for loop in our 'C' compiler.

21) Nesting of Loops: Nesting of loops, that is one loop occurs within another loop. We can nest both for and while loops within it selves or within others.

Picture of the DFA Underlying our Scanner



Code

```
%{
char *yylval;
int yylineno;
char c;
#include<string.h>
#include<math.h>

%}

digit [0-9]
id [_a-zA-Z][a-zA-Z0-9]*
space [ \t]
ar [+-*/=]
%%

{digit}+      {
 yylval=strdup(yytext+2);
if(yylval[yyleng-1]>='A' && yylval[yyleng-1]<='z')
{
printf("Variable encountered\n");
}
else
printf("%s - Integer encountered\n",yytext);
}

\n ++yylineno;

{digit}*"."{digit}+    printf("%s - Float encountered\n",yytext);

\"[^"\n]*["\n]   {
                        yylval=strdup(yytext+1);
                        if(yylval[yyleng-2] !="")
                                printf("Illegal Termination of String \n");
                        else
                                yylval[yyleng-2]=0;
                                printf("%s-Constant String \n",yylval);
                      }

"+"|"-"|"*"|"/"    { printf(" %s - Arithmetic Operator encountered\n", yytext );}

">"|"<"|"=="|"!="|">="|"<="      {  printf("%s - Relational Operator encountered\n",yytext);}

"&&"|"||"|"!"    printf("%s - Logical Operator encountered\n",yytext);

"="|"+="|"-="|"*="|"/="   printf("%s - Assignment Operator encountered\n",yytext);

"++"|"--"    printf("%s - Unary Arithmetic Operator encountered\n",yytext);

#include[ ]*<[^\n]*>|#define[^\n]*|#include[ ]*\"[^\n]*\"    {printf("%s - Preprocessor
Directive\n",yytext);}
```

```
if|else|do|while|for|switch|case|continue|break printf("%s - Keyword\n",yytext);
int|char|float                                          printf("%s - Keyword\n",yytext);
printf|scanf|fopen|fclose|getchar|getch|clrscr  printf("%s - Keyword\n",yytext);
void|main|static|extern|auto|strcmp|strcpy|strcat|strlen {
                                                        printf("%s - Keyword\n",yytext);}
{id}        {
 yylval=strdup(yytext+1);
if(yylval[yyleng-1]>=0 && yylval[yyleng-1]<=9)
{
printf("Variable encoutered\n");
}
else

printf("%s - Identifier\n",yytext);
}

"("|")"|"{"|"}"|"["|"]"|";"|","     printf("%s - Special Symbol\n",yytext);

[[:blank:]]*
"/*"       {  for( ; ; )
                {
                          while((c=input())!='*' && c!=EOF);

                      if(c=='*')
                         {
                                  while((c=input())=='*');

                                  if(c=='/'){

                                  break;}
                         }
                         if(c==EOF)
                         {
                         printf("Illegal EOF encountered\n");
                         break;
                         }
                }
          }


. printf("Line no. %d : Illegal token found %s\n",yylineno,yytext);
%%
int main(int argc,char *argv[])
{

        yyin=fopen(argv[1],"r");
        yylex();
        fclose(yyin);
        return(0);
}
```

Assumptions

1. “printf”, “scanf”, “sizeof” and similar other predefined functions are considered as keywords

2. While recognizing function declaration, it is assumed that a function declaration can have any number of arguments or parameters.

3. There may be a slight error in the line number whenever a multiline comment spanning multiple lines is encountered since the newline within the comments may not increment the line number.

4. No special distinction has been made between array variables and normal variables.

Conclusion

The scanner successfully recognizes all the tokens mentioned. It lists the tokens and their types side by side one after the other. This output can now be fed to the next stage of compilation.