Assignment 1

Problem Statement : Implement a Lexical Analyzer using LEX for a subset of C. Cross check your output with Stanford LEX.

Objectives: To understand the first phase of compiler and implementation with lex tool.

Outcome: Successful implemetation of lexical analyzer using lex tool.

Software Requirements: LEX,YACC,GCC

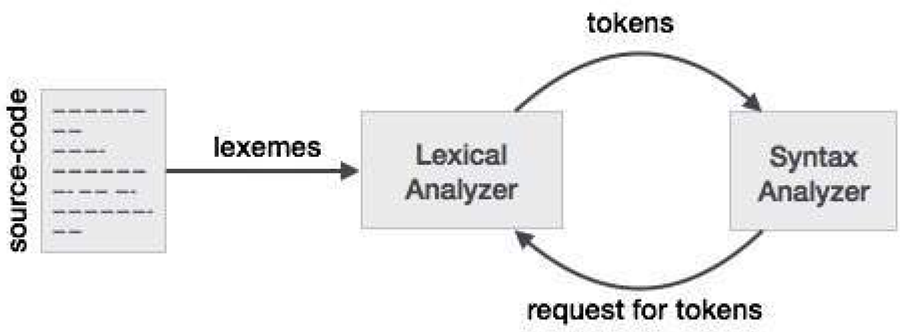
Hardware Requirements: 4GB RAM, 500 GB HDD

Theory:

Lexical Analysis:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors 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.



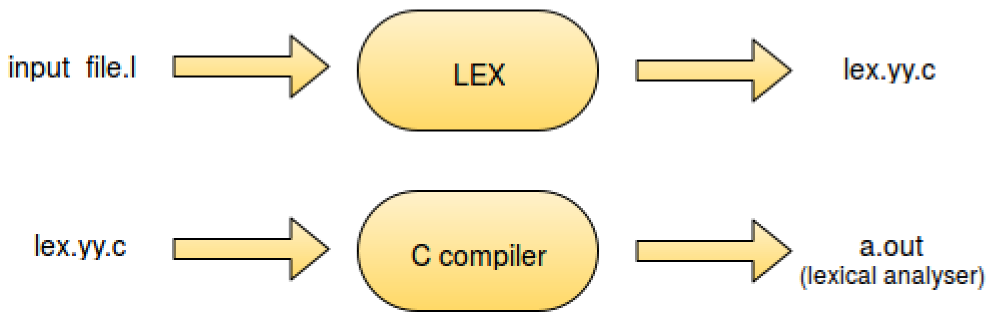
Tokens: A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”

Pattern: A pattern is a description which is used by the token. In the case of a keyword which uses as a token, the pattern is a sequence of characters.

Lex:

LEX is a tool used to generate a lexical analyzer. Technically, LEX translates a set of regular expression specifications (given as input in input_file.l) into a C implementation of a corresponding finite state machine (lex.yy.c). This C program, when compiled, yields an executable lexical analyzer.



The source program is fed as the input to the the lexical analyzer which produces a sequence of tokens as output.Conceptually, a lexical analyzer scans a given source program and produces an output of tokens.

The structure of LEX programs

A lex program consists of three parts: the definition section, the rules section, and the user subroutines.

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

Declarations:

The declarations section consists of two parts, **auxiliary declarations** and **regular definitions**.

The auxiliary declarations are copied as such by LEX to the output *lex.yy.c* file. This C code consists of instructions to the C compiler and are not processed by the LEX tool.The auxiliary declarations (which are optional) are written in C language and are enclosed within ' %{ ' and ' %} ' . It is generally used to declare functions, include header files, or define global variables and constants.

Rules

Rules in a LEX program consists of two parts :

1. The pattern to be matched

2. The corresponding action to be executed

```
p1 {action 1}
p2 {action 2}
p3 {action 3}
     ...
     ...
     ...
     ...
pn {action n}
```

Where, each p is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.

LEX obtains the regular expressions of the symbols 'number' and 'op' from the declarations section and generates code into a function *yylex()* in the *lex.yy.c* file. This function checks the input stream for the first match to one of the patterns specified and executes code in the action part corresponding to the pattern.

Auxiliary functions

LEX generates C code for the rules specified in the Rules section and places this code into a single function called *yylex()*. In addition to this LEX generated code, the programmer may wish to add his own code to the *lex.yy.c* file. The auxiliary functions section allows the programmer to achieve this.

The auxiliary declarations and auxiliary functions are copied as such to the lex.yy.c file

Once the code is written, *lex.yy.c* maybe generated using the command *lex "filename.l"* and compiled as *gcc lex.yy.c*

The yyvariables

yyin()

yyin is a variable of the type FILE* and points to the input file. *yyin* is defined by LEX automatically. If the programmer assigns an input file to *yyin* in the auxiliary functions section, then *yyin* is set to point to that file. Otherwise LEX assigns *yyin* to stdin(console input).

yylex()

yylex() is a function of return type int. LEX automatically defines *yylex()* in *lex.yy.c* but does not call it. The programmer must call yylex() in the Auxiliary functions section of the LEX program. LEX generates code for the definition of yylex() according to the rules specified in the Rules section.

yywrap()

LEX declares the function yywrap() of return-type int in the file *lex.yy.c* . LEX does not provide any definition for yywrap(). yylex() makes a call to yywrap() when it encounters the end of input. If yywrap() returns zero (indicating *false*) yylex() assumes there is more input and it continues scanning from the location pointed to by yyin. If yywrap() returns a non-zero value (indicating true), yylex() terminates the scanning process and returns 0 (i.e. “wraps up”). If the programmer wishes to scan more than one input file using the generated lexical analyzer, it can be simply done by setting yyin to a new input file in yywrap() and return 0.

As LEX does not define yywrap() in lex.yy.c file but makes a call to it under yylex(), the programmer must define it in the Auxiliary functions section or provide %option noyywrap in the declarations section. This options removes the call to yywrap() in the lex.yy.c file. It is **mandatory** to either define yywrap() or indicate the absence using the %option feature. If not, LEX will flag an error.

Conclusion:

Thus we’ve implemented lexical analyzer for a subset of C program using lex tool.

Code:

```
%{

#include<stdio.h>

%}

identifier [a-zA-Z][a-zA-Z0-9]*

%%

#.* {printf("\n%s is a preprocessor directive",yytext);}
```

```
float|int|char|double|long|void|include|typedef printf("%s IS A KEYWORD\n",yytext);
{identifier}(\[[0-9]*\])? printf("%s is an identifier\n",yytext);
["|(|)|{|}|#|;|,|%|&] printf("%s IS A SPECIAL SYMBOL\n", yytext);
["]+[a-zA-Z0-9]+["] printf("%s IS A MESSAGE\n", yytext);
\".*\" printf("%s is a STRING\n",yytext);
[+|-|*|/] printf("%s IS A ARITHMATIC OPERATOR\n", yytext);
{identifier}\( {printf("%s is a FUNCTION\n ",yytext);}
[a-zA-Z]*.h printf("%s IS A HEADER FILE\n", yytext);
[0-9]+ printf("%s IS A CONSANT\n", yytext);
= printf("%s IS AN ASSIGNMENT OPERATOR \n", yytext);
[<|>|<=|>=|<>] printf("%s IS A RELATIONAL OPERATOR\n", yytext);
[&&|!=|==] printf("%s IS A LOGICAL OPERATOR\n", yytext);

%%
int yywrap(void)
{
return 1;
}
int main()
{
FILE *fp=fopen("code.c","r");
yyin = fp;
yylex();
yywrap();
return 0;
}


/*
```

code.c-

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

OUTPUT-

```
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment1$ gcc
lex.yy.c
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment1$ lex
assign1.l
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment1$ gcc
lex.yy.c
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment1$ ./a.out

#include<stdio.h> is a preprocessor directive
#include<conio.h> is a preprocessor directive
void IS A KEYWORD
 main( is a FUNCTION
 ) IS A SPECIAL SYMBOL
```

```
{ IS A SPECIAL SYMBOL
int IS A KEYWORD
 a is an identifier
, IS A SPECIAL SYMBOL
b is an identifier
, IS A SPECIAL SYMBOL
c is an identifier
; IS A SPECIAL SYMBOL
a is an identifier
= IS AN ASSIGNMENT OPERATOR
1 IS A CONSANT
; IS A SPECIAL SYMBOL
b is an identifier
= IS AN ASSIGNMENT OPERATOR
2 IS A CONSANT
; IS A SPECIAL SYMBOL

c is an identifier
= IS AN ASSIGNMENT OPERATOR
a is an identifier
+ IS A ARITHMATIC OPERATOR
b is an identifier
; IS A SPECIAL SYMBOL
printf( is a FUNCTION
 "Sum:%d" is a STRING
, IS A SPECIAL SYMBOL
c is an identifier
) IS A SPECIAL SYMBOL
; IS A SPECIAL SYMBOL
} IS A SPECIAL SYMBOL
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment1$

*/
```