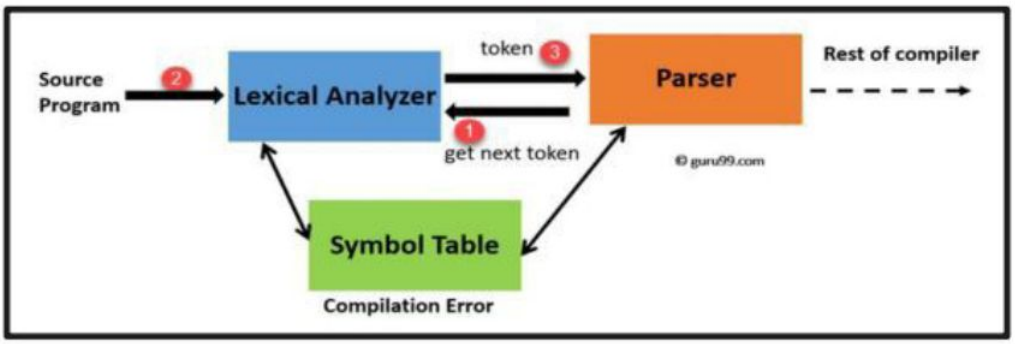
1. Introduction:

LEXICAL ANALYSIS is the very first phase in the compiler designing. It takes the modified source code which is written in the form of sentences. In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyser breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.

Programs that perform lexical analysis are called lexical analysers. A lexical analyser contains tokenizer or scanner. If the lexical analyser detects that the token is invalid, it generates an error. It reads character streams from the source code, checks for legal tokens, and pass the data to the syntax analyser when it demands.

Architecture and Working:



2. My Assumptions:

While designing the lexical analyser for a language L1, I have assumed the following assumptions.

Note: My language L1 will ignore all single line comments '//' ,'whitespaces' and '\n' while reading the tokens.

Special Symbol: ; { } ( ) , #

Keyword: int, char, float, bool, cin, cout, main

Pre-processor Directives: include, define

Library: iostream, studio, string

Operators: *, +, >>, <<,>,<

Numbers/Integers: All numbers Values from 0-9.

Identifies/ Variables: All alphabetic strings except the keywords, numbers, Pre-processor directive and library strings.

3. Lexical Analyzer (LEX.cpp)

```
#include <bits/stdc++.h>
#include <regex>
#include <iterator>
#define deb(x) cout<<#x<<" = "<<x<<endl


using namespace std;

map<string,string> Make_Regex_Map(){

    map<string,string> my_map {
        { "\\;|\\{|\\}|\\(|\\)|\\,|\\#", "Special Symbol"},
        { "int|char|float|bool|cin|cout|main|using|namespace|std","Keywords"},
        { "\\include|define", "Pre-Processor Directive"},
        {"\\iostream|\\stdio|\\string","Library"},
        { "\\*|\\+|\\>>|\\<<|<|>",  "Operator"},
        { "[0-9]+" ,    "Integer" },
        { "[^include][^iostream][^int][^main][^cin][^cout][^;][^>>][^,][^[B ;cin]][a-
z]+" ,   "Identifier" },
        { "[A-Z]+",    "Variable"},
        {"[ ]",""},

    };

    return my_map;
}

map<size_t,pair<string,string>> Match_Language (map<string,string> patterns,string str){

 map< size_t, pair<string,string> > lang_matches;

    for ( auto i = patterns.begin(); i != patterns.end(); ++i )
    {
        regex compare(i->first);
        auto words_begin = sregex_iterator( str.begin(), str.end(), compare );
        auto words_end   = sregex_iterator();
    //MAKING PAIRS OF [STRING OF REGEX 'compare' : 'pattern']
        for ( auto it = words_begin; it != words_end; ++it )
            lang_matches[ it->position() ] = make_pair( it->str(), i->second );
    }
    return lang_matches;
}

string tell_Lexeme(string op){
    if(op=="*") return "MUL";
    else if(op=="+") return "ADD";
    else if(op==">>") return "INS";
    else if(op=="<<") return "EXTR";
    else if(op==">") return "RSHFT";
    else if(op=="<") return "LSHFT";
}
```

```cpp
int main()
{
    ofstream fout;
    fout.open("OutputFile");
    char c;
    fstream fin("SourceCode.txt", fstream::in);
    string str;
    //Fetching Source Code in String type 'str'
    while(fin>> noskipws>>c)
        str=str+c;

    //Making a map which which will define the regex in source code to its pattern in my l
anguage.
    map<string,string> patterns =Make_Regex_Map();

    /*DECLARING MAP 'lang_matches' from 'patterns' map which will pair up the patterns
    from the ['Source Code':'Defined Pattern' via a Regex named 'compare'. */
    map< size_t, pair<string,string> > lang_matches = Match_Language(patterns,str);

    // Writing matches in File ignoring 'spaces' and '\n'.
    for ( auto match = lang_matches.begin(); match != lang_matches.end(); ++match ){

        if(!(match->second.first==" ")&&!(match->second.first=="//")){
            if(match->second.second=="Variable"||match->second.second=="Identifier")
            fout<<"Token (  '"<< match->second.first << "' " <<" -------> "<< match-
>second.second  <<" ,  POINTER TO SYMBOL TABLE )"<<endl;
            else{
                if(match->second.second=="Operator"){
                string op=tell_Lexeme(match->second.first);
                fout<<"Token (  "<< match->second.first << " " <<" -------> "<< match-
>second.second<<" , "<<op<<"  )"  <<endl;
                }
                else
                fout<<"Token (  "<< match->second.first << " " <<" -------> "<< match-
>second.second<<"  )"  <<endl;
            }
        }
    }
    return 0;
}
```

4. Source File (SourceFile.txt)

LEX.cpp | SourceCode.txt | OutputFile

SourceCode.txt

```cpp
#include <iostream>
#define LIMIT 5
using namespace std ;
int main(){
    // this comment and program is written by akshit mangotra to avoid lexical analyzer for checking comment
    int A , B ;
    cin >> A >> B;
    cout << A * B ;
}
```

5. OUTPUT FILE CONTAINING TOKEN (OutputFile.txt)

LEX.cpp | SourceCode.txt | OutputFile

OutputFile

```
1   Token (  #  -------> Special Symbol  )
2   Token (  include  -------> Pre-Processor Directive  )
3   Token (  <  -------> Operator , LSHFT  )
4   Token (  iostream  -------> Library  )
5   Token (  >  -------> Operator , RSHFT  )
6   Token (  #  -------> Special Symbol  )
7   Token (  define  -------> Pre-Processor Directive  )
8   Token (  'LIMIT'  -------> Variable ,  POINTER TO SYMBOL TABLE )
9   Token (  5  -------> Integer  )
10  Token (  using  -------> Keywords  )
11  Token (  namespace  -------> Keywords  )
12  Token (  std  -------> Keywords  )
13  Token (  ;  -------> Special Symbol  )
14  Token (  int  -------> Keywords  )
15  Token (  main  -------> Keywords  )
16  Token (  (  -------> Special Symbol  )
17  Token (  )  -------> Special Symbol  )
18  Token (  {  -------> Special Symbol  )
19  Token (  int  -------> Keywords  )
20  Token (  'A'  -------> Variable ,  POINTER TO SYMBOL TABLE )
21  Token (  ,  -------> Special Symbol  )
22  Token (  'B'  -------> Variable ,  POINTER TO SYMBOL TABLE )
23  Token (  ;  -------> Special Symbol  )
24  Token (  cin  -------> Keywords  )
25  Token (  >>  -------> Operator , INS  )
26  Token (  'A'  -------> Variable ,  POINTER TO SYMBOL TABLE )
27  Token (  >>  -------> Operator , INS  )
28  Token (  'B'  -------> Variable ,  POINTER TO SYMBOL TABLE )
29  Token (  ;  -------> Special Symbol  )
30  Token (  cout  -------> Keywords  )
31  Token (  <<  -------> Operator , EXTR  )
32  Token (  'A'  -------> Variable ,  POINTER TO SYMBOL TABLE )
33  Token (  *  -------> Operator , MUL  )
34  Token (  'B'  -------> Variable ,  POINTER TO SYMBOL TABLE )
35  Token (  ;  -------> Special Symbol  )
36  Token (  }  -------> Special Symbol  )
37
```