PRACTICAL FILE

BE (CSE) 6th Semester

COMPILER DESIGN (CS654)

March 2021 – May 2021

Submitted By

Jatin Ghai
(UE183042)

Submitted To

Dr. Akashdeep
Assistant Professor, Computer Science and Engineering

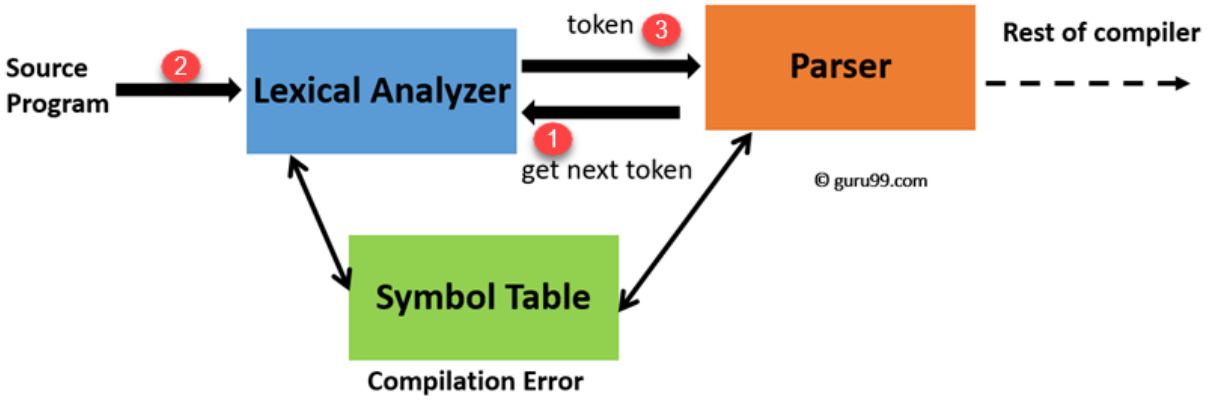
Computer Science and Engineering
University Institute of Engineering and Technology
Panjab University, Chandigarh – 160014, INDIA
2021

Experiment No. 2

AIM:-To Design a Basic Lexical Analyser

Lexical analysis is the very first phase in the compiler designing. A Lexer 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 analyzer 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 analyzers or lexers. A lexer contains a tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Lexical analyzer scans the entire source code of the program. It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser.



Here is how this works-

1. "Get next token" is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parser.

Lexical Analyzer skips whitespaces and comments while creating these tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

Source Code:

```
#include<bits/stdc++.h>
using namespace std;
vector<string> reserved =
{"asm","auto","bool","break","case","catch","char","class","cons
t","const_cast","continue","default","delete","do","double","dyn
amic_cast","else","enum","explicit","export","extern","false","f
loat","for","friend","goto","if","inline","int","long","mutable"
,"namespace","new","operator","private","protected","public","re
gister","reinterpret_cast","return","short","signed","sizeof","s
tatic","static_cast","struct","switch","template","this","throw"
,"true","try","typedef","typeid","typename","union","unsigned","
using","virtual","void","volatile","wchar_t","while"};
vector<string> libraries =
{"bits/stdc++.h","cin","cout","iostream","string","unordered_map
","vector"};
vector<string> type =
{"NONE","Operator","Keywords","Identifiers","Literals","Punctuat
ors"};

void func(unordered_map <string, int>& umap,string &code);
int main(){
    string text,code;
    unordered_map <string,int> umap;
    ifstream file("tt.cpp");

    while (getline (file, text)) {
        code += text;
    }
    func(umap, code);

    for(int i = 1; i < type.size(); i++){
        for(auto y: umap){
            if( y.second == i){
                cout << type[i] << " " << y.first << endl;
            }
        }
    }
```

```
    file.close();
}

void func(unordered_map <string,int>& umap,string &code){
    string st;
    for(int i = 0; i < code.size(); i++){

        while(isspace(code[i]) || code[i] == '\n' || code[i] ==
';') i++;
        st = "";
        if(code[i] == ',' || code[i] == '(' || code[i] == ')' ||
code[i] == '{' || code[i] == '}' || code[i] == '#' || code[i] ==
';'){

            st = code[i];
            umap[st] = 5;//punctuators
        }else if( code[i] == '+' || code[i] == '=' || code[i] ==
'-' || code[i] == '*' || code[i] == '^' || code[i] == '%' ||
code[i] == '!' ||code[i] == '/' || code [i] == '<' || code[i] ==
'>' || code [i] == '&'){

            st = "";
            st = code[i];
            i++;
            if( code[i-1] == '<' || code[i-1] == '>'){
                if( code[i] == '<' || code[i] == '>' ){
                    st.push_back(code[i]);
                    umap[st] = 1; //operator;
                }else {
                    i--;
                    umap[st] = 5; //;
                }
            }
            else if( code[i] == '+' || code[i] == '=' || code[i]
== '-' || code[i] == '*' || code[i] == '^' || code[i] == '%' ||
code[i] == '!' ||code[i] == '/'){
                st.push_back(code[i]);
                umap[st] = 1; //operator;
            }else{
```

```
                i--;
                umap[st] = 1; //operator;
            }
        }else if((i>=2 && code[i-1] == '=' && code[i-2] != '=')
|| (i>=3 && code[i-2] == '=' && code[i-3] != '=') ){
            st = code[i];
            i++;
            while(code[i] != ',' && code[i] != ';'){
                if(isspace(code[i])) continue;
                st.push_back(code[i]);
                i++;
            }
            umap[st] = 4; //literals
        }else if( code[i] == '\"' ){
            st = code[i];
            i++;
            while( code[i] != '\"'){
                st.push_back(code[i]);
                i++;
            }
            st.push_back(code[i]);
            umap[st] =  4; //literals
        }else if( code[i] == '\'' ){
            st = code[i];
            i++;
            while( code[i] != '\''){
                st.push_back(code[i]);
                i++;
            }
            st.push_back(code[i]);
            umap[st] =  4; //literals
        }else{
            while(i< code.size() && ( !isspace(code[i]) &&
code[i] != ';' && code[i] != '=' && code[i] != '(' && code[i] !=
'<' && code[i] != '>' && code[i] != ')' && code[i] != '+' &&
code[i] != '-' && code[i] != '/' && code[i] != '*' && code[i] !=
',' && code[i] != '}')){
                st.push_back(code[i]);
                i++;
```

```
            }
            i--;
            if(i+1 < code.size() && code[i+1]=='('){
                umap[st] = 3; //Identifiers
            }else if
(binary_search(reserved.begin(),reserved.end(),st) ){
                umap[st] = 2; //keyword;
            }else if
(binary_search(libraries.begin(),libraries.end(),st) ){
                umap[st] = 3; //Identifiers
            }else{
                umap[st] = 3; //Identifiers
            }
        }
    }
}
```

Testing File Code:

```
#include<iostream>
#include <string>
using namespace std;
int main(){
    int a = 9;
    float b = 9.86;
    string s = "abs";
    cout << s << a << b << endl;
}
```

Output:

```
C:\Users\jatin\Desktop>lexical_analyser.exe
Operator <<
Operator =
Keywords float
Keywords using
Keywords namespace
Keywords int
Identifiers cout
Identifiers string
Identifiers endl
Identifiers iostream
Identifiers include
Identifiers std
Identifiers main
Identifiers s
Identifiers b
Identifiers a
Literals "abs"
Literals 9.86
Literals 9
Punctuators }
Punctuators >
Punctuators #
Punctuators )
Punctuators <
Punctuators (
Punctuators {
```

Algorithm:

1. First I read all the lines of code and append it into a single string.
2. From that string I extract a character in a for loop and check if it matches any condition, then sub conditions and categories it.
3. Incase character is a blank space I increment the counter to the next character.

Learning from the experiment:

1. I learned about lexical analysis, tokens, lexemes and to separate different tokens from a code and categorise them.