SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

CUSTOMIZED LEXICAL ANALYZER

Guided by:

Dr. K. VIJAYA

Submitted By:

Sourav Paul (RA2011003010016)

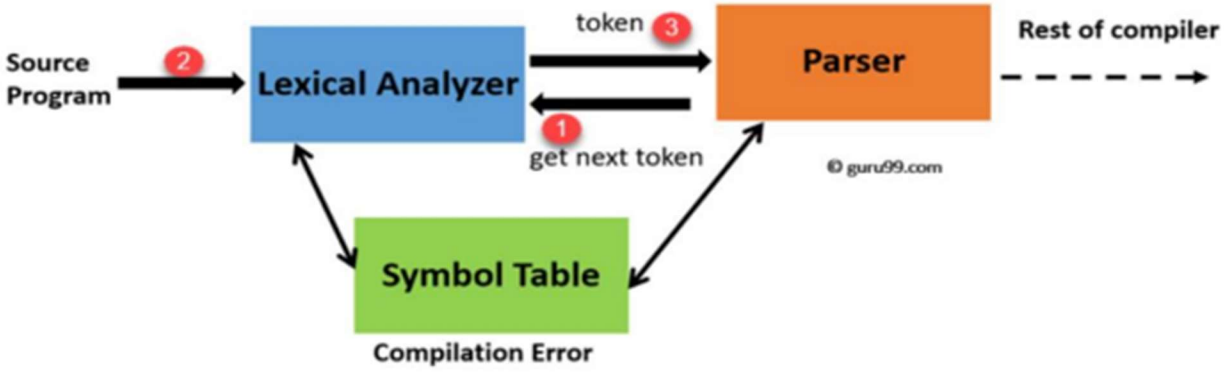
Suraj Pratap Singh(RA2011003010031)

Aim:- To develop a lexical analyzer for customized language.

ABSTRACT:-

A lexical analyzer, also known as a lexer or scanner, is a program component that takes an input stream of characters and converts it into a sequence of tokens for further processing. The task of the lexical analyzer is to identify the basic units or "tokens" in the input stream, such as keywords, identifiers, literals, operators, and punctuation marks. This process involves applying a set of rules or regular expressions to the input stream to recognize each token and associate it with a corresponding token type. The output of the lexer is a stream of token objects, each containing the token type and any associated data or attributes. The resulting stream of tokens can then be fed into a parser or other processing component for further analysis or interpretation. Overall, the lexical analyzer plays a critical role in the compilation or interpretation of programming languages and other formal languages.

Working of a Lexical Analyzer



A lexical analyzer typically performs the following tasks:

1. **Tokenizing:** The scanner categorizes each lexeme into a token, which is a symbol representing a category of lexemes, such as a keyword or an identifier.

2. **Scanning**: The scanner reads the input characters one at a time and groups them into lexemes, which are the smallest meaningful units of a programming language.

3. **Error handling:** The scanner detects and reports any lexical errors, such as invalid characters or illegal sequences of characters.

Our 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.

Customized Language

Special Symbol: ; { } ( ) , #

Keyword: number , character , floating , Boolean , console_in, console_out , automatic , doub , structure , breaker , els , longer , option , socket , option , enumeration , reg , type , external , send , combine , flow , loop , sign , empty , does , if , stat , until , fixed , go , size , vol , const , mini , unsign

Pre-processor Directives: import , define

Library: inputoutput , studinp_out , string

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

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

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

CODE

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

using namespace std;

string keywords =
"number|character|floating|boolean|console_in|console_out|automatic|doub|stru
cture|breaker|els|longer|option|socket|option|enumeration|reg|type|external|s
end|combine|flow|loop|sign|empty|does|if|stat|until|fixed|go|size|vol|const|m
ini|unsign|main";

map<string, string> Make_Regex_Map()
{
    map<string, string> my_map{
        {"\\;|\\{|\\}|\\(|\\)|\\,|\\#", "Special Symbol"},
        {"number|character|floating|boolean|console_in|console_out|automatic|
doub|structure|breaker|els|longer|option|socket|option|enumeration|reg|type|e
xternal|send|combine|flow|loop|sign|empty|does|if|stat|until|fixed|go|size|vo
l|const|mini|unsign|main", "Keywords"},
        {"\\import|define", "Pre-Processor Directive"},
        {"\\inputoutput|\\stdinp_op|\\string", "Library"},
        {"\\*|\\+|\\>>|\\<<|<|>", "Operator"},
        {"[0-9]+", "Integer"},
        {"[^import][^input_output][^number][^main][^console_in][^console_out]
[^;][^>>][^,][^[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";
}
// bool false_Identifier(map<size_t, pair<string, string>> lang_matches)
// {
//     for (auto match = lang_matches.begin(); match != lang_matches.end();
++match)
//     {

//     }

// }
bool isValidIdentifier(string s){
    if(keywords.find(s)!=string::npos) return false;
    if(!(isalpha(s[0]) || s[0]=='_')) return false;
    if(s.find(" ")!=string::npos) return false;
    if(s.find(';')!=string::npos || s.find('(')!=string::npos ||
s.find(')')!=string::npos || s.find('{')!=string::npos ||
s.find('}')!=string::npos || s.find('#')!=string::npos) return false;
```

```
    return true;
}

int main()
{
    ofstream fout;
    cout << endl
         << endl
         << endl;
    cout.fill(' ');
    cout.width(100);
    fout.open("OutputFile");
    char c;
    string filename;

    cout << "ENTER THE SOURCE CODE FILE NAME: Example \"abc.txt\" \n";
    cin >> filename;
    fstream fin(filename, fstream::in);
    string str;
    // Fetching Source Code in String type 'str'
    if (fin.is_open())
    {
        while (fin >> noskipws >> c)
            str = str + c;

        // Making a map which which will define the regex in source code to
its pattern in my language.
        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'.
        int count = 1;
        cout << "\t\t\t\t-------------------------------------------------
--------------------------------------------- \n";
        cout.width(40);
        cout << "\t        NUMBER" << setw(10) << "              TOKEN "
             << " "
             << "             " << setw(20) << " PATTERN \n";
        cout.fill(' ');
```

```
        cout.width(40);

        cout << "\t\t\t\t-------------------------------------------------------
------------------------------------------- \n\n\n";
        // cout<<"\t\t\t\t                                PROCESSING SOURCE
CODE.......                                        \n\n\n";
        // Sleep(5000);

        for (auto match = lang_matches.begin(); match != lang_matches.end();
++match)
        {
            // if(match->second.first[0]=='1' || match->second.first[0] ==
'2' || match->second.first[0] == '3' ||  match->second.first[0] ==
'4'||match->second.first[0] == '5'||match->second.first[0]== '6'||match-
>second.first[0] == '7'||match->second.first[0]== '8'||match->second.first[0]
== '9' && (match->second.first[1]=='a')){
            //     if(match->second.second== "Identifier" || match-
>second.second=="Variable" )
            //     {
            //         cout<<"Wrong Identifier";
            //         break;
            //     }
            // }
            if( (match->second.second=="Identifier" ||match-
>second.second=="Variable") && isValidIdentifier(match-
>second.first)==false){
                        cout<<"Wrong Identifier";
                        break;
            }


            if (!(match->second.first == " ") && !(match->second.first ==
"//"))
            {

                if (match->second.second == "Variable" || match-
>second.second == "Identifier")
                {

                    cout.width(40);
                    if (count < 10)
                    {
                        string double_digits = to_string(count);
                        double_digits = "0" + double_digits;
```

```
                        cout << "\t Token   No :" << double_digits <<
"  |   " << setw(10) << match->second.first << " "
                             << " -------> |" << setw(25) << match-
>second.second << setw(18) << " ,  POINTER TO SYMBOL TABLE    " << endl;
                        fout << "\t Token   No :" << double_digits <<
"  |   " << setw(10) << match->second.first << " "
                             << " -------> |" << setw(25) << match-
>second.second << setw(18) << " ,  POINTER TO SYMBOL TABLE    " << endl;
                        Sleep(1500);
                    }
                    else
                    {
                        cout << "\t Token   No :" << count << "  |   " <<
setw(10) << match->second.first << " "
                             << " -------> |" << setw(25) << match-
>second.second << setw(18) << " ,  POINTER TO SYMBOL TABLE    " << endl;
                        fout << "\t Token   No :" << count << "  |   " <<
setw(10) << match->second.first << " "
                             << " -------> |" << setw(25) << match-
>second.second << setw(18) << " ,  POINTER TO SYMBOL TABLE    " << endl;
                        Sleep(1500);
                    }
                    count++;
                }

                else
                {
                    if (match->second.second == "Operator")
                    {
                        cout.width(40);
                        string op = tell_Lexeme(match->second.first);
                        if (count < 10)
                        {
                            string double_digits = to_string(count);
                            double_digits = "0" + double_digits;
                            cout << "\t Token   No :" << double_digits <<
"  |   " << setw(10) << match->second.first << " "
                                 << " -------> |" << setw(25) << match-
>second.second << " , " << op << "    " << endl;
                            fout << "\t Token   No :" << double_digits <<
"  |   " << setw(10) << match->second.first << " "
                                 << " -------> |" << setw(25) << match-
>second.second << " , " << op << "    " << endl;
                            count++;
                        }
```

```
                        else
                        {
                            cout << "\t Token   No :" << count << "  |   " <<
setw(10) << match->second.first << " "
                                << " ------->  |" << setw(25) << match-
>second.second << " , " << op << "    " << endl;
                            fout << "\t Token   No :" << count << "  |   " <<
setw(10) << match->second.first << " "
                                << " ------->  |" << setw(25) << match-
>second.second << " , " << op << "    " << endl;
                            Sleep(1500);
                            count++;
                        }
                    }
                    else
                    {
                        cout.width(40);
                        if (count < 10)
                        {
                            string double_digits = to_string(count);
                            double_digits = "0" + double_digits;
                            cout << "\t Token   No :" << double_digits <<
"  |   " << setw(10) << match->second.first << " "
                                << " ------->  |" << setw(25) << match-
>second.second << "    " << endl;
                            fout << "\t Token   No :" << double_digits <<
"  |   " << setw(10) << match->second.first << " "
                                << " ------->  |" << setw(25) << match-
>second.second << "    " << endl;
                            count++;
                        }
                        else
                        {
                            cout << "\t Token   No :" << count << "  |   " <<
setw(10) << match->second.first << " "
                                << " ------->  |" << setw(25) << match-
>second.second << "    " << endl;
                            fout << "\t Token   No :" << count << "  |   " <<
setw(10) << match->second.first << " "
                                << " ------->  |" << setw(25) << match-
>second.second << "    " << endl;
                            count++;
                        }
                    }
                }
```

```
            }
        }

        string command = " ";

        while (command != "EXIT")
        {
            cout.fill(' ');
            cout.width(40);
            cout << "\n\n\t PRESS TYPE `EXIT` TO CLOSE WINDOW.\n\t NOTE: AN
OUTPUT FILE WILL BE GENERATED IN THE SAME FOLDER AS `Output.txt` \n";
            cin.width(40);
            cin >> command;

            if (command == "exit" || command == "EXIT" || command == "Exit")
                break;

            else
            {
                cout.fill(' ');
                cout.width(40);
                cout << "Please enter correct word.";
                cin.width(10);
                cin >> command;
            }
        }
    }

    else
    {
        cout.fill(' ');
        cout.width(40);
        cout << "\n FILE NOT FOUND!\n\n";
    }
    return 0;
}
```

Requirements to run the script:

i) Create a Input File (text file) named MySourceCode.txt

ii) Write the code in our customized language for tokenizing by lexical analyzer .

main.cpp | OutputFile | SourceCode.txt

```
#import <inputoutput>
 main(){

    number A , B ;
    console_in >> A >> B;
    console_out << A * B ;
}

```

OUTPUT

A file named OutputFile will get created containing the tokens of the input file .

main.cpp | OutputFile | SourceCode.txt

```
Token ( #  -------> Special Symbol )
Token ( import  -------> Pre-Processor Directive )
Token ( <  -------> Operator , LSHFT )
Token ( >  -------> Operator , RSHFT )
Token ( main  -------> Keywords )
Token ( (  -------> Special Symbol )
Token ( )  -------> Special Symbol )
Token ( {  -------> Special Symbol )
Token ( number  -------> Keywords )
Token ( 'A'  -------> Variable , POINTER TO SYMBOL TABLE )
Token ( ,  -------> Special Symbol )
Token ( 'B'  -------> Variable , POINTER TO SYMBOL TABLE )
Token ( ;  -------> Special Symbol )
Token ( console_in  -------> Keywords )
Token ( >>  -------> Operator , INS )
Token ( 'A'  -------> Variable , POINTER TO SYMBOL TABLE )
Token ( >>  -------> Operator , INS )
Token ( 'B'  -------> Variable , POINTER TO SYMBOL TABLE )
Token ( ;  -------> Special Symbol )
Token ( console_out  -------> Keywords )
Token ( <<  -------> Operator , EXTR )
Token ( 'A'  -------> Variable , POINTER TO SYMBOL TABLE )
Token ( *  -------> Operator , MUL )
Token ( 'B'  -------> Variable , POINTER TO SYMBOL TABLE )
Token ( ;  -------> Special Symbol )
Token ( }  -------> Special Symbol )
```

Result:

Tokenizing of customized language code is done successfully .