Experiment No 2

Aim: Write a program to implement Lexical analyzer

Learning Objective: Converting a sequence of characters into a sequence of tokens.

Theory:

THE ROLE OF LEXICAL ANALYZER

The lexical analyzer is the first phase of a compiler. Its main task is to read the input

characters and produce as output a sequence of tokens that the parser uses for syntax analysis. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

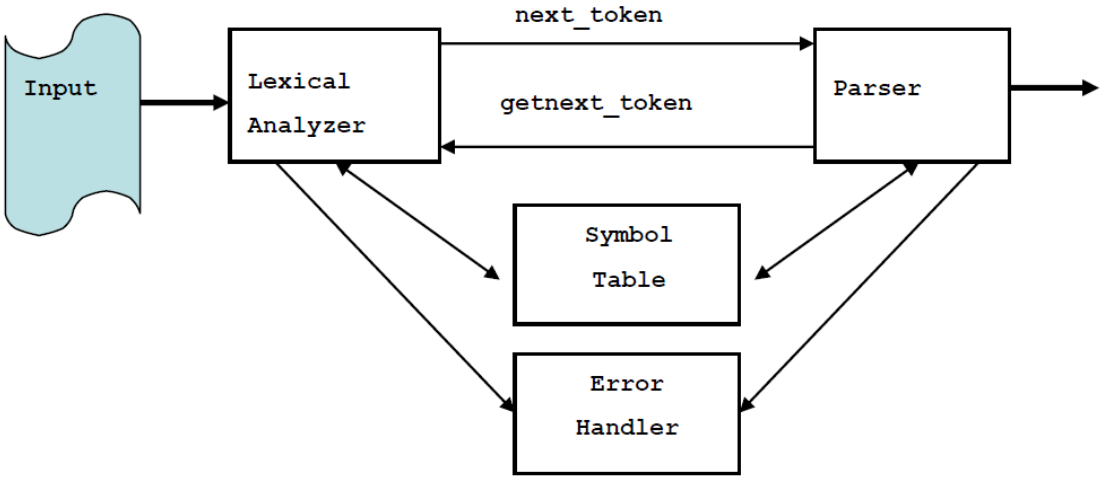


Figure 4.1 Interaction of Lexical Analyzer with Parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white spaces in the form of blank, tab, and new line characters. Another is correlating error messages from the compiler with the source program. Sometimes lexical analyzers are divided into a cascade of two phases first called "scanning" and the second "lexical analysis". The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Implementation Details

1. Read the high level language as source program
2. Convert source program in to categories of tokens such as Identifiers, Keywords, Constants, Literals and Operators.

Code :

```
#include<bits/stdc++.h>
using namespace std;


bool isPunctuator(string ch){
    if (ch == " " || ch == "+" || ch == "-" || ch == "*" ||
        ch == "/" || ch == "," || ch == ";" || ch == ">" ||
        ch == "<" || ch == "=" || ch == "(" || ch == ")" ||
        ch == "[" || ch == "]" || ch == "{" || ch == "}" ||
        ch == "&" || ch == "|" || ch=="$")
        {
            return true;
        }
    return false;
}
bool validIndentifier(string str)
{
   if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isPunctuator(to_string(str[0])) == true|| str[0] ==
'"')
        {
            return false;
        }
    int len = str.size();
    if(len == 1)
    {
        return true;
    }
    else{
        for(int i=1;i<len;i++)
        {
            if(isPunctuator(to_string(str[i])) == true)
            {
                return false;
            }
        }
    }
```

```
    return true;
}

bool isOperator(string ch)
{
        if (ch == "+" || ch == "-" || ch == "*" ||
        ch == "/" || ch == ">" || ch == "<" ||
        ch == "=" || ch == "|" || ch == "&")
    {
       return true;
    }
    return false;
}

bool isKeyword(string str)
{
        if (!str.compare("auto") || !str.compare("break") ||
!str.compare("case") || !str.compare("char") || !str.compare("const")
            || !str.compare("continue") || !str.compare("default") ||
!str.compare("do") || !str.compare("double") || !str.compare("else")
            || !str.compare("enum") || !str.compare("extern") ||
!str.compare("float") || !str.compare("for") || !str.compare("goto")
            || !str.compare("if") || !str.compare("int") ||
!str.compare("long") || !str.compare("register") || !str.compare("return")
            || !str.compare("short") || !str.compare("signed") ||
!str.compare("sizeof") || !str.compare("static") || !str.compare("struct")
            || !str.compare("switch") || !str.compare("typedef") ||
!str.compare("union") || !str.compare("unsigned") || !str.compare("void")
            || !str.compare("volatile") || !str.compare("while") ||
!str.compare("string")
         )
        {
            return true;
        }
    else
    {
       return false;
    }
}


bool isLiteral(string str)
{
    int n = str.size();
    if((str[0] >= 'a' && str[0]<='z')||(str[0]>='A' && str[0]<='Z')||(str[0]
>= '0' && str[0]<='9')||(isPunctuator(to_string(str[0]))))
    {
        return false;
    }
```

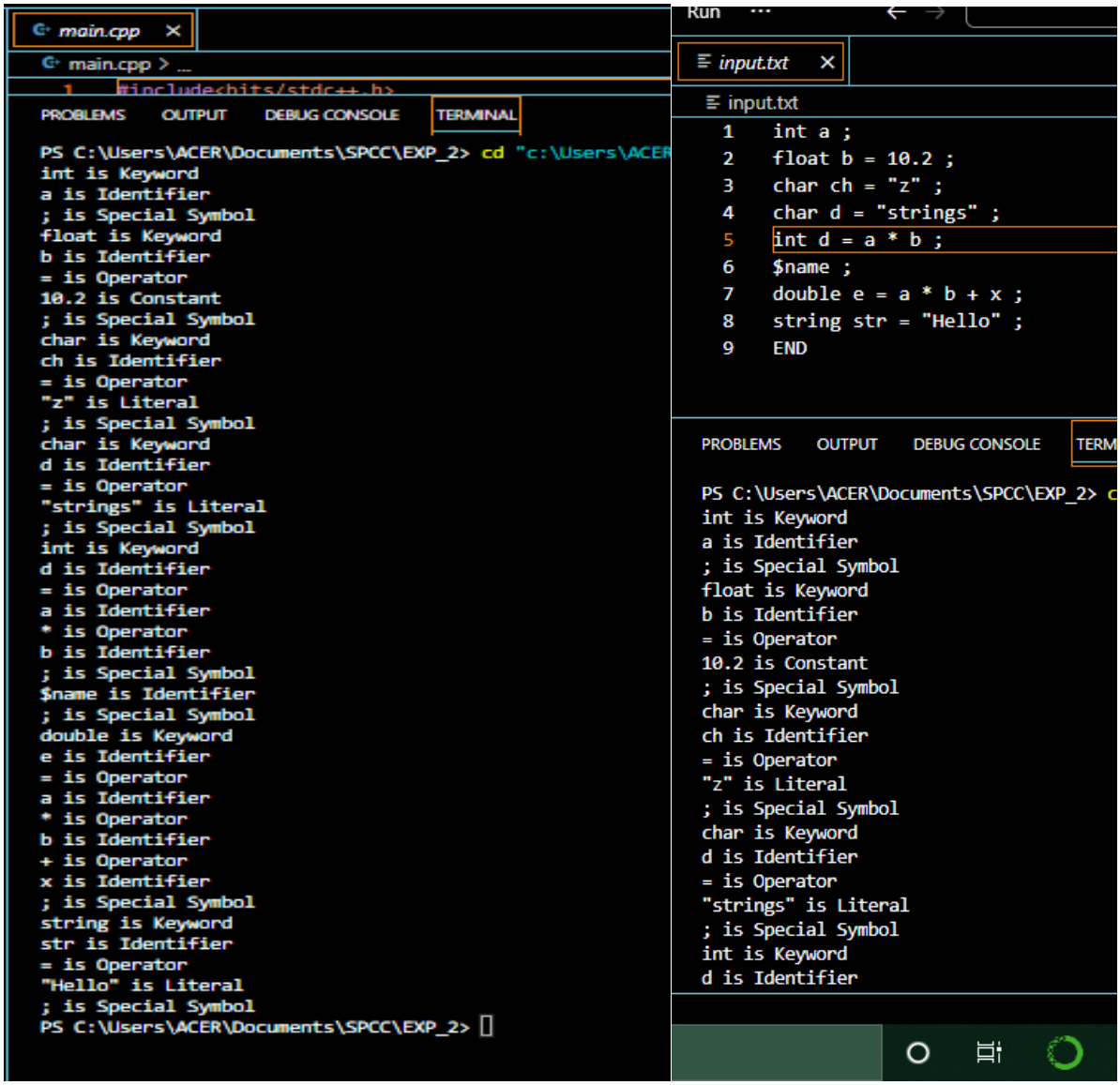
```
    return true;
}
bool isConstant(string str)
{
     int i, len = str.size(),numOfDecimal = 0;
    if (len == 0)
    {
        return false;
    }
    for (i = 0 ; i < len ; i++)
    {
        if (numOfDecimal > 1 && str[i] == '.')
        {
            return false;
        } else if (numOfDecimal <= 1)
        {
            numOfDecimal++;
        }
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            {
                return false;
            }
    }
    return true;
}


int main(){

    fstream Inputfile;
    string word ,filename = "./input.txt";
    Inputfile.open(filename.c_str());
    vector<string>words;

    while (Inputfile >> word)
    {
        words.push_back(word);
    }

    for(int  j=0 ;j<words.size()-1;j++)
    {
        string i = words[j];
        if(isKeyword(i))
        {
            cout<<i<<" is "<<"Keyword"<<endl;
        }
```

```
        else if(isOperator(i))
        {
            cout<<i<<" is "<<"Operator"<<endl;
        }
        else if(isPunctuator(i))
        {
            cout<<i<<" is "<<"Special Symbol"<<endl;
        }
        else if(validIndentifier(i))
        {
            cout<<i<<" is "<<"Identifier"<<endl;
        }
        else if(isLiteral(i))
        {
            cout<<i<<" is "<<"Literal"<<endl;
        }
        else{
            cout<<i<<" is "<<"Constant"<<endl;
        }
    }



    return 0;
}
```

Test cases:

1. Input undefined token

Conclusion:



Post Lab Questions:

1. Explain the role of automata theory in compiler design.
2. What are the errors that are handled by Lexical analysis phase?