George / Lexical Analyzer

Task: Write a program, using files (both input and output) that reads in a small C/C++ program using a postfix expression and identify the lexemes as to their respective tokens.

Language : C++

Overview: My program lexical analyzes a simple postfix expression from an input file (plain text file) called *list.txt.* It identifies the lexemes present in the postfix expression and outputs them alongside their respective toxen. The output of the lexemes and tokens are organized in a table. The table is sent to an output file called *output.txt*. My lexical analyzer recognizes a specific subset and keywords of the c++ language.

Description: First, I typed up all the headers for my code and made sure to #include <string> because I planned on using the built in string function *.length().* Also, I *used* #include <fstream> to be able to read from /write to files. After, I created an object called *infile* of type ifstream in order to read the postfix expression from the input file *list.txt*. Then, I created an object called *outfile* of type fstream to write to the file *output.txt*. I used the *.open()* command to open both *list.txt* and *output.txt.*

```
ifstream infile; // infile (object) of type ifstream

infile.open("list.txt"); //read from input file called list.txt // list.txt default
file //input file stream

fstream outfile;

outfile.open("output.txt");
```

Then, I created an if statement that would check if the input file does not exist. If the input file does not exist this would make the stream invalid. The if statement uses a boolean context. If not infile then output the error message on to the screen and terminate the program.

```
if(!infile) { // if file does not open //stream variable boolean context cerr <<

  "Failed, could not open file "<< endl; //error message

  exit(1); //termination of program }
```

So, what are my lexemes and tokens ?

Before writing my program I already had in mind the lexemes I wanted my code to recognize and what the name of their respective tokens would be. The chart below shows the lexemes and tokens I used.

Lexemes Tokens
int || double || bool || string || float variable_name
main || void function_name
cout output_stream
a|| b || c || d || e ||f || g || h || i || …… z identifier
== equalto_operator
if || else || while c++_keywords
_ delimiter_op
+ || - || % || / || * id_operation
= assignment_op
{ open_bracket
} close_bracket
(open_parenth
) close_parenth
0 || 1 | 2 || 3 || 4 || …... 9 int_literal
; ending_operator

*The program tests for positive single digit numbers (0 -9). Therefore, if a postfix expression contains an integer with more than one digit each digit in the number is read separately. For instance, the number 657 would be recognized as three separate integer literals being 6, 6, and then 7.

*In addition, my program recognizes only lower case single alphabet letters as lexemes. For example, if a postfix expression used the variables *ABCD* or *abc* my program would not recognize it as a lexeme.

Main code:

I declared a string array called alpha that would represent the 26 letters in the alphabet. The lower case letters in the string array represent the different possible identifiers. Then, I declared a string and assigned it the variable name *postfix*. *Postfix* is the string I read from the input file.

```
string alpha[] = {"a","b", "c", "d", "e", "f","g", "h", "i", "j", "k","l", "m", "n",
"o", "p","q", "r", "s", "t", "u","v", "w", "x", "y", "z"}; //string array
```

```
string postfix; // treat postfix expression in input file as a string
```

Then I decided to use a while loop. The condition in my while loop is (*infile* >> *postfix.*) I used the extraction operator so that my filestream (*infile*) reads in my string (*postfix*). With the >> operator I am able to read the postfix expression **word by word** rather than line by line . The space in between each word is used as a delimiter. Therefore, the values in the postfix expression that are separated by a space are considered two different values and read as individual words. I made sure that each individual value in the postfix expression is separated by a space before writing a new one. Then I declared six *if statements* that would check for specific string values. If postfix is equal to (==) a specific string it is outputted to the *output.txt* file on the left hand side of the table and the designated token name is outputted on the right hand side of the table.

```
while (infile >> postfix) { // reading my string word by word // infile acts as cin

       if(postfix == "main" || postfix == "void"){

       outfile << postfix << " |" << " Function_Name" << endl; }
```

After the six if statements I declared a *for loop*. A string is an array of characters and to read each character in the postfix expression I will need to read each index of that array. Thus, I decided that I would use a for loop to read through each index in the string. I initialized int i to 0 so the first index read in the string is 0. Next, I made the condition statement *i is less than postfix.length()*. I used the built-in string function .length(). The variable i will start from 0 and go up to the length of the string. I incremented i (*i*++) so each index that is read will start from 0 and it will keep going by 1 until the condition statement is no longer true. The for loop is then followed by nine if statements that check for specific characters (lexemes). These specific char values represent the new group of lexemes we want to recognize in the string. Next, I closed the for loop then the while loop.

```
for(int i =0; i < postfix.length(); i++) { // reads each char in the string

  if(postfix[i] == '{'){

     outfile << postfix[i] << " |" << " Open_bracket''<< endl; //recognizing a
open bracket

    }
```

Lastly, I called *close()* for infile and outfile.

```
infile.close();

outfile.close();

return 0; }
```

Test Results:

1.Testing the error routine if it is not infile. This test uses an invalid file. *(Image 1a- test code , Image 1b test results)*

```
ifstream infile;   // infile (object) of type ifstream
infile.open("listttttttt.txt"); //read from input file called

ofstream outfile;
outfile.open("output.txt");

if(!infile) {
    cerr << "Failed, could not open list.txt file "<< endl;
    exit(1); //termination of program
```

Image1a

```
Failed, could not open file
```

Image 1b error routine works!

2.Testing the *while loop* to make sure the string is read word by word. I am using the input file *list.txt with the following postfix expression* to test the while loop. *(Image 2a- test code, Image 2b- test results)*

```
list.txt    list2.txt
list.txt
int main () {
int z ;
z = 6_4_+_2_*_
cout << z ;
}
```

----------->*list.txt (postfix expression)*

```
string postfix; // treat postfix expression in input file as a string


while (infile >> postfix) {
    outfile << postfix; //Test code to make sure each word is read


  /*if(postfix == "main" || postfix == "void"){
        outfile << postfix  << "          |" << "       Function_Name" << endl;
      }

    if(postfix == alpha[0] || postfix == alpha[1] || postfix == alpha[2] || postfix == a
        outfile << postfix << "             |" << "         identifier" << endl;
      }

    if(postfix == "int" || postfix == "double" || postfix == "string" || postfix == "boo
        outfile << postfix  << "          |" << "      Variable_Literal" << endl;
      }

    if(postfix == "if" || postfix == "else" || postfix == "while"){
```

Image 2a

```
output.txt
 1  ==================================
 2  Lexemes     |        Tokens
 3  ==================================
 4  int
 5  main
 6  ()
 7  {
 8  int
 9  z
10  ;
11  z
12  =
13  6_4_+_2_*_
14  cout
15  <<
16  z
17  ;
18  }
```

Image 2b

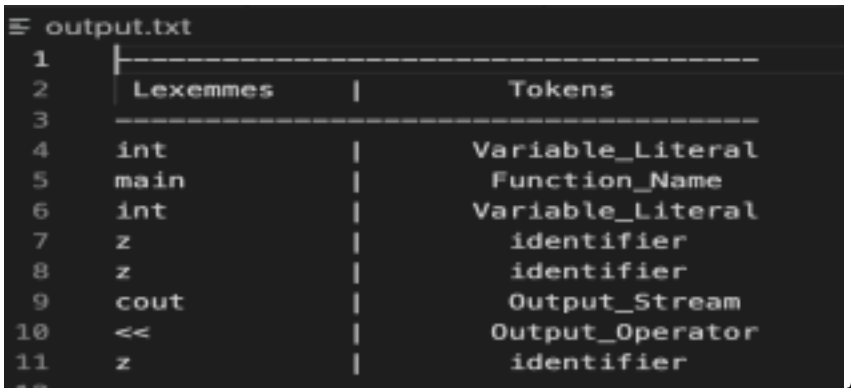
While loop works !

3. Testing the first group of *if statements*. The code should be able to recognize the lexemes present in the string and assign it to its appropriate token. (*Image 3a- test code, Image 3b-*

```
while (infile >> postfix) {
    |

  if(postfix == "main" || postfix == "void"){
      outfile << postfix  << "          |" << "       Function_Name" << endl;
    }

  if(postfix == alpha[0] || postfix == alpha[1] || postfix == alpha[2] || postfix == alpha[3] ||postfix == alpha[4]  || postfix == alpha[5]
      outfile << postfix << "             |" << "         identifier" << endl;
    }

  if(postfix == "int" || postfix == "double" || postfix == "string" || postfix == "bool" || postfix == "float"){
      outfile << postfix  << "          |" << "      Variable_Literal" << endl;
    }

  if(postfix == "if" || postfix == "else" ||  postfix == "while"){
      outfile << postfix  << "          |" << "       C++_Keywords" << endl;
    }

  if(postfix == "cout" ){
      outfile << postfix  << "          |" << "       Output_Stream" << endl;
    }

  if(postfix == "<<" ){
      outfile << postfix  << "          |" << "       Output_Operator" << endl;
    }

  if(postfix == "==" ){
      outfile << postfix  << "          |" << "       Equalto_Operator" << endl;
    }
```

Image 3a

----> the lexemes & tokens available for this test run



output.txt

Lexemmes	Tokens
int	Variable_Literal
main	Function_Name
int	Variable_Literal
z	identifier
z	identifier
cout	Output_Stream
<<	Output_Operator
z	identifier

Image 3b *If statements work !!*

4. Testing the for loop and if statements. The for loop should be able to read the individual characters in the string and identify them as lexemes. With the if statements the lexemes should be designated to their respective tokens. (*Image 4a- test code, Image 4b- test results)*

Image 4a

```
output.txt
1   ------------------------------------
2    Lexemes     |        Tokens
3   ------------------------------------
4   (            |        Open_parenth
5   )            |        Close_parenth
6   {            |        Open_bracket
7   ;            |        Ending_Operator
8   =            |         Assign_op
9   6            |        Int_Literal
10  _            |        Delimiter_op
11  4            |        Int_Literal
12  _            |        Delimiter_op
13  +            |        id_Operation
14  _            |        Delimiter_op
15  2            |        Int_Literal
16  _            |        Delimiter_op
17  *            |        id_Operation
18  _            |        Delimiter_op
19  ;            |        Ending_Operator
20  }            |        Close_bracket
```

Image 4b for loop and if statements work !

5. Testing that all parts of the program are successful. The table should be displayed in the output file. (cpp file attached) (*Image 5b- test results)*

```
output.txt
1   ------------------------------------
2    Lexemes     |        Tokens
3   ------------------------------------
4   int          |      Variable_Literal
5   main         |       Function_Name
6   (            |        Open_parenth
7   )            |        Close_parenth
8   {            |        Open_bracket
9   int          |      Variable_Literal
10  z            |        identifier
11  ;            |        Ending_Operator
12  z            |        identifier
13  =            |         Assign_op
14  6            |        Int_Literal
15  _            |        Delimiter_op
16  4            |        Int_Literal
17  _            |        Delimiter_op
18  +            |        id_Operation
19  _            |        Delimiter_op
20  2            |        Int_Literal
21  _            |        Delimiter_op
22  *            |        id_Operation
23  _            |        Delimiter_op
24  cout         |        Output_Stream
25  <<           |       Output_Operator
26  z            |        identifier
27  ;            |        Ending_Operator
28  }            |        Close_bracket
29
```

Image 5b all the lexemes are identified and assigned to their respective token !

// To test all the lexemes and tokens used in my code I created two extra input files.

6. Testing input file called *list2.txt*.

list.txt list2.txt

list2.txt

```
int main () {
int z ;
int b ;
z = 6_4_+_2_*_
b = 6_4_+_
cout << z_b_+_ ; }
```

-----> *postfix expression in list2.txt*

list.txt list2.txt list3.txt output.txt

output.txt

```
|-----------------------------------|
 Lexemmes     |        Tokens
=====================================
int           |     Variable_Literal
main          |      Function_Name
(             |       Open_parenth
)             |       Close_parenth
{             |       Open_bracket
int           |     Variable_Literal
z             |        identifier
;             |       Ending_Operator
int           |     Variable_Literal
```

list.txt list2.txt list3.txt

list3.txt

```
double y ;
y = 9_+_4_*_2_
if ( y == 8 ) {
    cout << y ;
    else {
     y = 5_2_+_
    }
}
```

--------> *the code recognizes the lexemes and assigns the propriate tokens.*

Testing input file called *list3.txt*.

```
9           |       Int_Literal
_           |       Delimiter_op
4           |       Int_Literal
_           |       Delimiter_op
+           |       id_Operation
_           |       Delimiter_op
cout        |       Output_Stream
<<          |      Output_Operator
_           |       Delimiter_op
_           |       Delimiter_op
+           |       id_Operation
_           |       Delimiter_op
;           |       Ending_Operator
}           |       Close_bracket
```

-----> *postfix expression in list3.txt*

---> *uses c++ keywords*

output.txt

```
==================================
Lexemmes    |        Tokens
----------------------------------
double          |      Variable_Literal
y           |        identifier
;           |        Ending_Operator
y           |        identifier
=           |         Assign_op
9           |        Int_Literal
_           |        Delimiter_op
+           |        id_Operation
_           |        Delimiter_op
4           |        Int_Literal
_           |        Delimiter_op
*           |        id_Operation
_           |        Delimiter_op
2           |        Int_Literal
_           |        Delimiter_op
if          |       C++_Keywords
(           |        Open_parenth
y           |        identifier
==          |       Equalto_Operator
=           |         Assign_op
=           |         Assign_op
8           |        Int_Literal
)           |        Close_parenth
{           |        Open_bracket
cout        |        Output_Stream
<<          |       Output_Operator
y           |        identifier
;           |        Ending_Operator
else        |       C++_Keywords
{           |        Open_bracket
y           |        identifier
=           |         Assign_op
5           |        Int_Literal
_           |        Delimiter_op
2           |        Int_Literal
_           |        Delimiter_op
+           |        id_Operation
_           |        Delimiter_op
}           |        Close_bracket
}           |        Close_bracket
```

----> *the code recognizes the lexemes and assigns the appropriate token!*

Conclusion/ Plans: I plan on updating my lexical analyzer so that it can analyze more complex postfix expressions. I plan on making it that it can recognize more than single digit numbers, and variables with more than one letter. I want to create an error routine that recognizes an invalid postfix expression. Another error message I want it to include is " _ delimiter missing_" if there is no delimiters or the proper amount in the postfix expression.

In addition, I want it to be able to recognize a larger subset of the c++ language and more keywords and maybe even recognize some of the C language. In conclusion, I really enjoyed this project and it gave me a lot of more insight as to how a lexical analyzer works.