Project 1 Approach:

I originally approached this project by first getting all the new tokens and reserved words defined, without testing if anything worked. Then I saw the Project 1 approach posted for this project. I restarted, and followed the approach explained by the provided material. I added certain tokens and then compiled with text files containing the new tokens to ensure everything was done correctly.

Once I had all tokens and reserved words added, I tackled the defining of real and bool literal tokens. I did this by looking at this week’s material and then defining the literal.

Then I started working on the modifications necessary to listing.cc to count and print different types of errors in the way specified by the project’s requirements. I used a queue to allow for printing of multiple errors on a previous line and I just added if statements to check the error type and increase the specific error counter in void appendError().

Testing:

Testing of this code involves two main parts. One is to check that all added tokens and reserved words work, and that they all have appropriate token numbers in lexemes.txt, which can be done by using valid input txt files. The other part is to ensure that errors are displayed correctly, which can be done by using txt input files with errors.

Test 1:

I created my own test file, which is a modified version of the supplied test6.txt file. I modified test6 to include all valid tokens possible. I am glad I played around with identifiers, because I found that my original definition of the identifier literal did not work properly, because after the first under score, it required all further underscores to be followed by a letter and a number in order.

Text

Description automatically generated

Input:

Graphical user interface, text, application

Description automatically generated

I looked through Lexemes.txt and everything seems to have appropriate token numbers:

Graphical user interface, application

Description automatically generated

Test 2:

Provided test 1 file, should print compilation successful

Graphical user interface, text, application

Description automatically generated

Test 3:

Provided test 2 file, should print 1 lexical error

Graphical user interface, text, application

Description automatically generated

Test 4:

Provided test 3 file, should print compilation successful

Graphical user interface, text, application

Description automatically generated

Test 5:

Provided test 4 file, should print compilation successful

Graphical user interface, text

Description automatically generated

Test 6:

Provided test 5 file, should print compilation successful

Graphical user interface, text, application

Description automatically generated

Test 7:

Provided test 6 file, should print compilation successful

Text

Description automatically generated

Need to test input files with errors:

Test 8:

Multiple errors on same line

Graphical user interface, text

Description automatically generated

Test 9

Multiple Errors on Multiple lines with illegal identifiers and illegal reals:

Text

Description automatically generated

Note: no matter what I did for the definition of real literal, I could not make it to where having multiple e or plus signs (line 8) to show an error. I have tried putting ? marks in multiple places since it is supposed to mean one or none, but it does not work for some reason.

Test 10:

Another test of correct input.

Text

Description automatically generated

Input: Text

Description automatically generated

Test 11:

Multiple errors multiple lines:

Text

Description automatically generated

Input:

Text

Description automatically generated

Lessons Learned and Improvements

One of the great lessons learned on this project was that using windows subsystems for Linux is easy and makes projects with C and C++ much easier to work through. I found trying to get C++, flex, and bison to run correctly irritating on windows using Cygwin and mingw. I almost just created a VM with an Ubuntu guest but decided to give WSL a try.

The issue with what I got for this project is that real literals are split tokens when using multiple e’s or +/- signs. If the real is something like 3.45ee++123, the real is split at the e’s are considered id’s. I kind of think this is because we do not have any syntax analysis or semantic yet.