Week 2: Project 1

John Kucera

Prof. Renee McDonald

CMSC 430 Compiler Theory and Design

24 March 2021

Week 2: Project 1 Documentation

Modifying the Lexical Analyzer/Compilation Listing Generator (C++ with Flex)

Approaching the Project

When it comes to assignments when given skeleton code, the first step I make is to understand everything in the skeleton code. I first made sure to go through all the weekly course material to get a handle on what concepts I would be working with in the program. To my surprise, the course materials contained videos from Dr. Jarc, who explained the skeleton code with incredible detail and focus on the compiler theory concepts. Dr. Jarc laid it out so well that I did not have to do much of my own research. After watching those videos, I had nearly complete understanding of how the source code, flex, bison, and make file all work together to generate the lexical analysis. Additionally, I used the g++ compiler to run the program (with text file examples) to see it work on my own machine.

After understanding the syntax and purpose of the source code, I saw that the requirements were just simple additions and modifications to the lexical analysis process. For example, I saw I had to add lexemes (like relational operators, andop/orop, a new type of comment, etc), so I went to scanner.l and to the translation rules to specify those required lexemes with the same syntax as the others. I also saw that I had to add a token for a real literal, so I first used the syntax for defining a int literal as a model. Then, I wrote out the regular expression that would represent a real literal and used that instead of the reg-ex for an int literal. When given skeleton code and being required to make small additions (like real literal token), it is best to use the other code (like int literal token) as an example that you can follow. This is especially useful when the syntax is still new to me.

Finally, for the requirements of adding details to recorded/displayed errors, I realized it all came down to making counters and incrementing those counters whenever the errors occurred. For example, this applied for the modification I made to lastLine() where the number of lexical, syntactic, and semantic errors were to be displayed. For displaying each error in the middle of output (even with multiple errors on a single line), I created another string that acted as a queue for each error. I appended to that queue each time an error occurred (with the error details) and made sure it got printed after the respective line in the code.

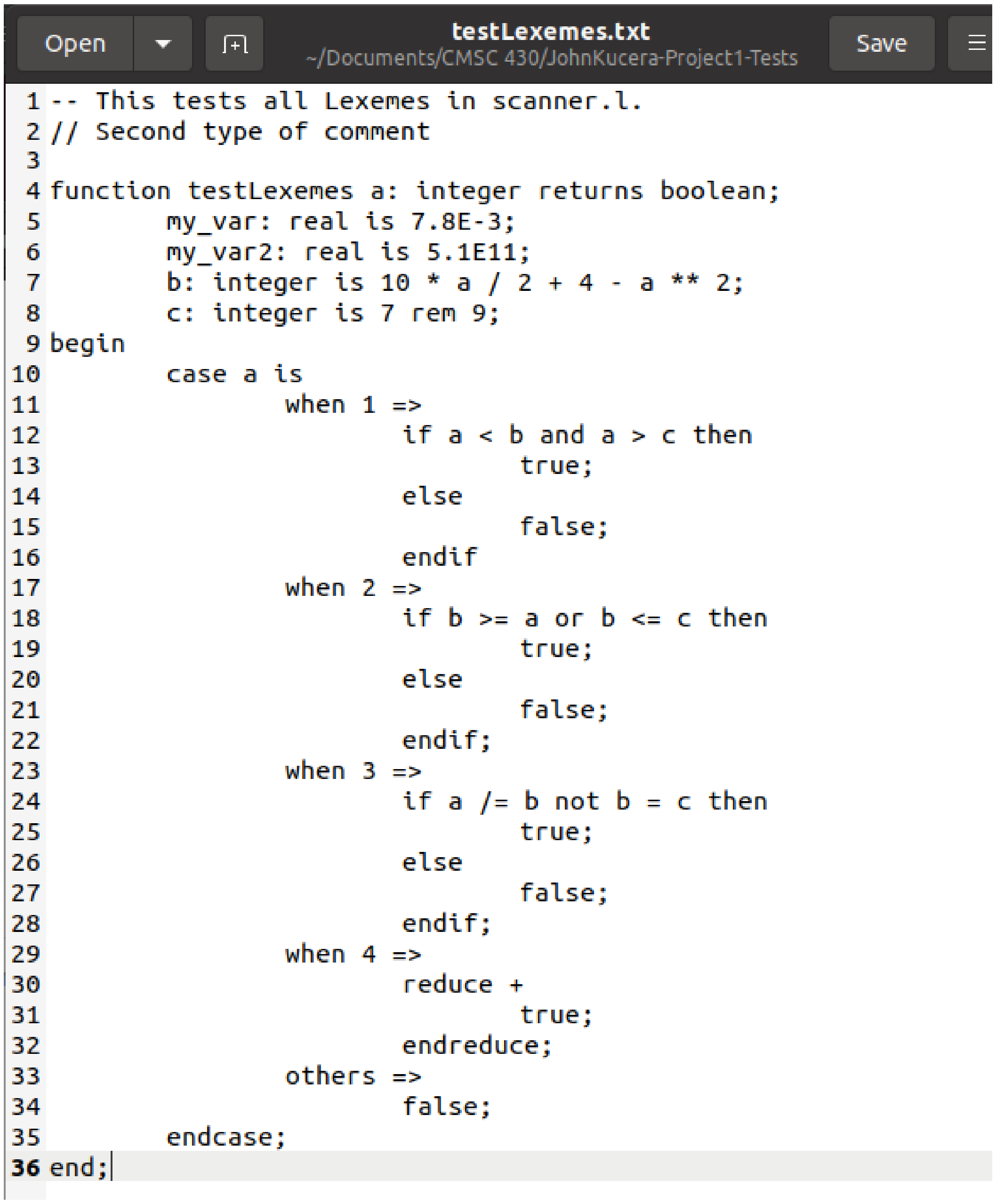
Week 2: Project 1

Test Cases

(Test Case 1)

Aspect tested: An input file containing all lexemes, including newly added ones

Input file: testLexemes.txt. Contains all lexemes listed in scanner.l, in addition to the new ones.

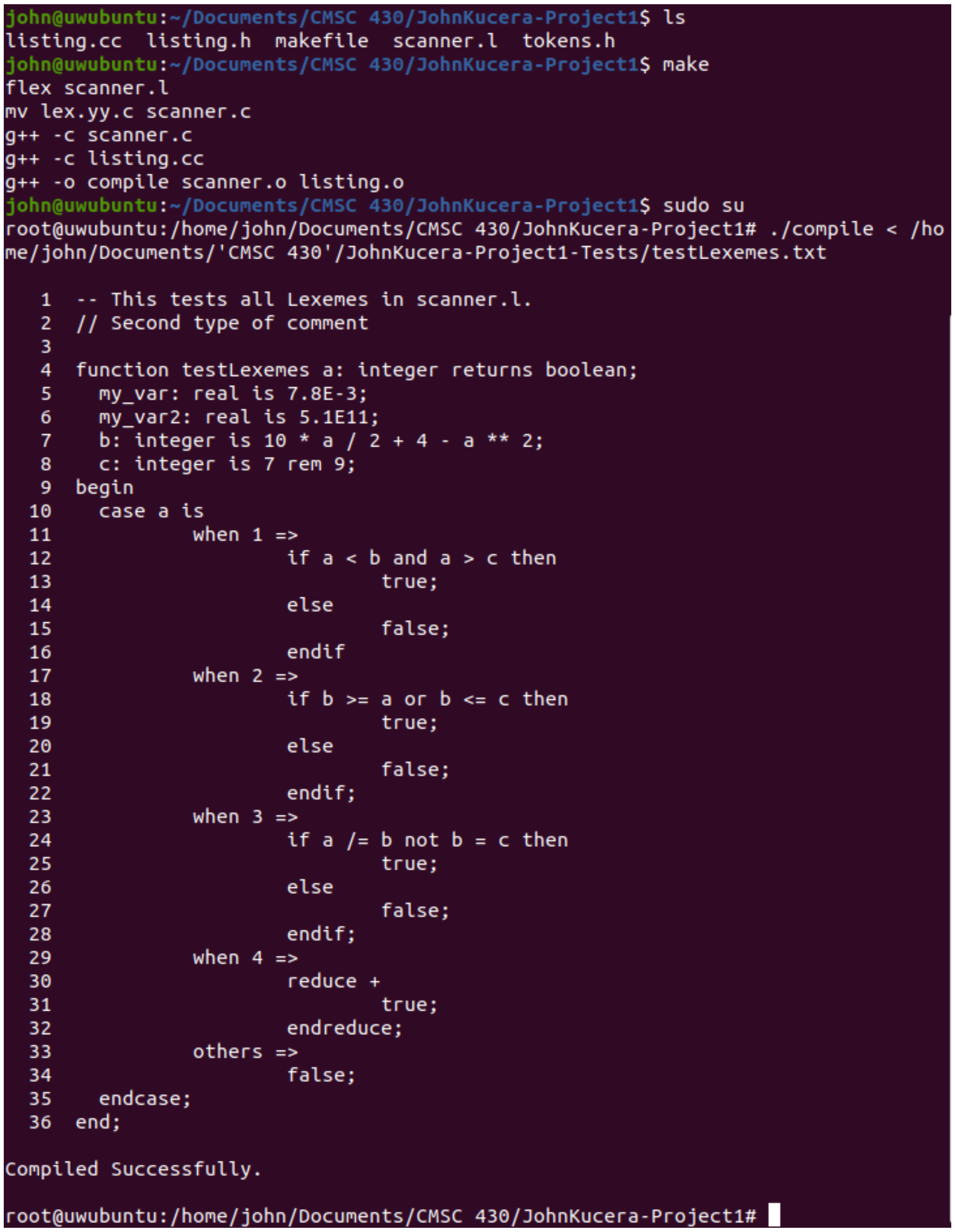


```
-- This tests all Lexemes in scanner.l.
// Second type of comment

function testLexemes a: integer returns boolean;
        my_var: real is 7.8E-3;
        my_var2: real is 5.1E11;
        b: integer is 10 * a / 2 + 4 - a ** 2;
        c: integer is 7 rem 9;
begin
        case a is
                when 1 =>
                        if a < b and a > c then
                                true;
                        else
                                false;
                        endif
                when 2 =>
                        if b >= a or b <= c then
                                true;
                        else
                                false;
                        endif;
                when 3 =>
                        if a /= b not b = c then
                                true;
                        else
                                false;
                        endif;
                when 4 =>
                        reduce +
                                true;
                        endreduce;
                others =>
                        false;
        endcase;
end;
```

Week 2: Project 1

Compilation output: SUCCESS. All lexemes are valid, last line prints “Compiled successfully.”

This screenshot also shows demonstration of using make to build the project before compiling.



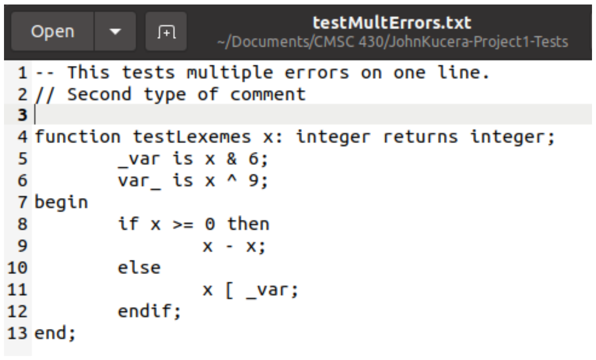
```
john@uwubuntu:~/Documents/CMSC 430/JohnKucera-Project1$ ls
listing.cc  listing.h  makefile  scanner.l  tokens.h
john@uwubuntu:~/Documents/CMSC 430/JohnKucera-Project1$ make
flex scanner.l
mv lex.yy.c scanner.c
g++ -c scanner.c
g++ -c listing.cc
g++ -o compile scanner.o listing.o
john@uwubuntu:~/Documents/CMSC 430/JohnKucera-Project1$ sudo su
root@uwubuntu:/home/john/Documents/CMSC 430/JohnKucera-Project1# ./compile < /ho
me/john/Documents/'CMSC 430'/JohnKucera-Project1-Tests/testLexemes.txt

   1  -- This tests all Lexemes in scanner.l.
   2  // Second type of comment
   3
   4  function testLexemes a: integer returns boolean;
   5    my_var: real is 7.8E-3;
   6    my_var2: real is 5.1E11;
   7    b: integer is 10 * a / 2 + 4 - a ** 2;
   8    c: integer is 7 rem 9;
   9  begin
  10    case a is
  11            when 1 =>
  12                    if a < b and a > c then
  13                            true;
  14                    else
  15                            false;
  16                    endif
  17            when 2 =>
  18                    if b >= a or b <= c then
  19                            true;
  20                    else
  21                            false;
  22                    endif;
  23            when 3 =>
  24                    if a /= b not b = c then
  25                            true;
  26                    else
  27                            false;
  28                    endif;
  29            when 4 =>
  30                    reduce +
  31                            true;
  32                    endreduce;
  33            others =>
  34                    false;
  35    endcase;
  36  end;

Compiled Successfully.

root@uwubuntu:/home/john/Documents/CMSC 430/JohnKucera-Project1# 
```

Week 2: Project 1

(Test Case 2)

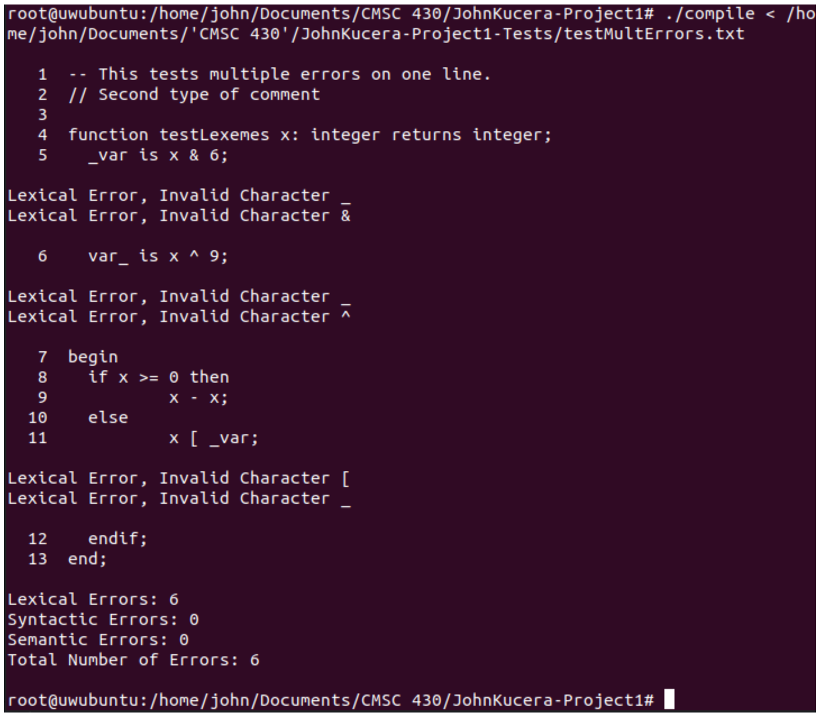
Aspect tested: An input file containing multiple errors on a single line

Input file: testMultErrors.txt. Contains multiple lines where there are multiple lexical errors, with improper identifiers (improperly placed underscore) and characters that are not allowed.



```
-- This tests multiple errors on one line.
// Second type of comment

function testLexemes x: integer returns integer;
        _var is x & 6;
        var_ is x ^ 9;
begin
        if x >= 0 then
                x - x;
        else
                x [ _var;
        endif;
end;
```

Compilation output: SUCCESS. Error messages are displayed after each line that has an error, even if there are multiple errors in that line. At the end, a count of errors of each type in addition to total number of errors is displayed.



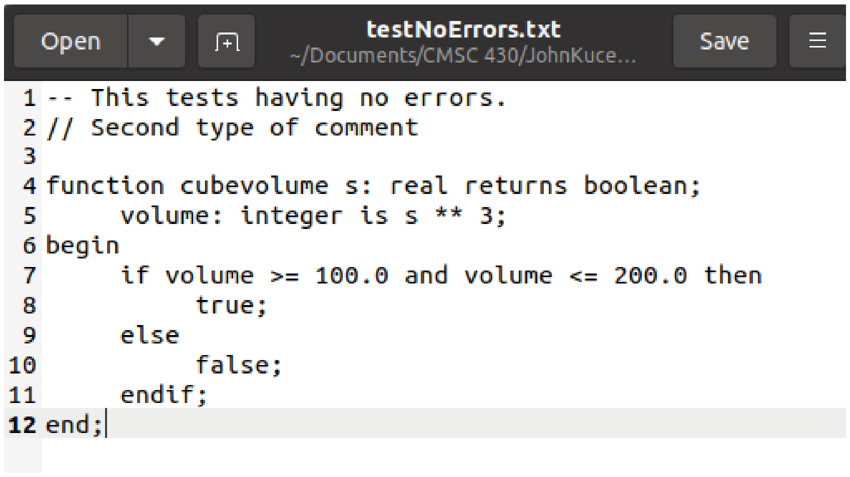
```
root@uwubuntu:/home/john/Documents/CMSC 430/JohnKucera-Project1# ./compile < /ho
me/john/Documents/'CMSC 430'/JohnKucera-Project1-Tests/testMultErrors.txt

   1  -- This tests multiple errors on one line.
   2  // Second type of comment
   3
   4  function testLexemes x: integer returns integer;
   5    _var is x & 6;

Lexical Error, Invalid Character _
Lexical Error, Invalid Character &

   6    var_ is x ^ 9;

Lexical Error, Invalid Character _
Lexical Error, Invalid Character ^

   7  begin
   8    if x >= 0 then
   9            x - x;
  10    else
  11            x [ _var;

Lexical Error, Invalid Character [
Lexical Error, Invalid Character _

  12    endif;
  13  end;

Lexical Errors: 6
Syntactic Errors: 0
Semantic Errors: 0
Total Number of Errors: 6

root@uwubuntu:/home/john/Documents/CMSC 430/JohnKucera-Project1# 
```

Week 2: Project 1

(Test Case 3)

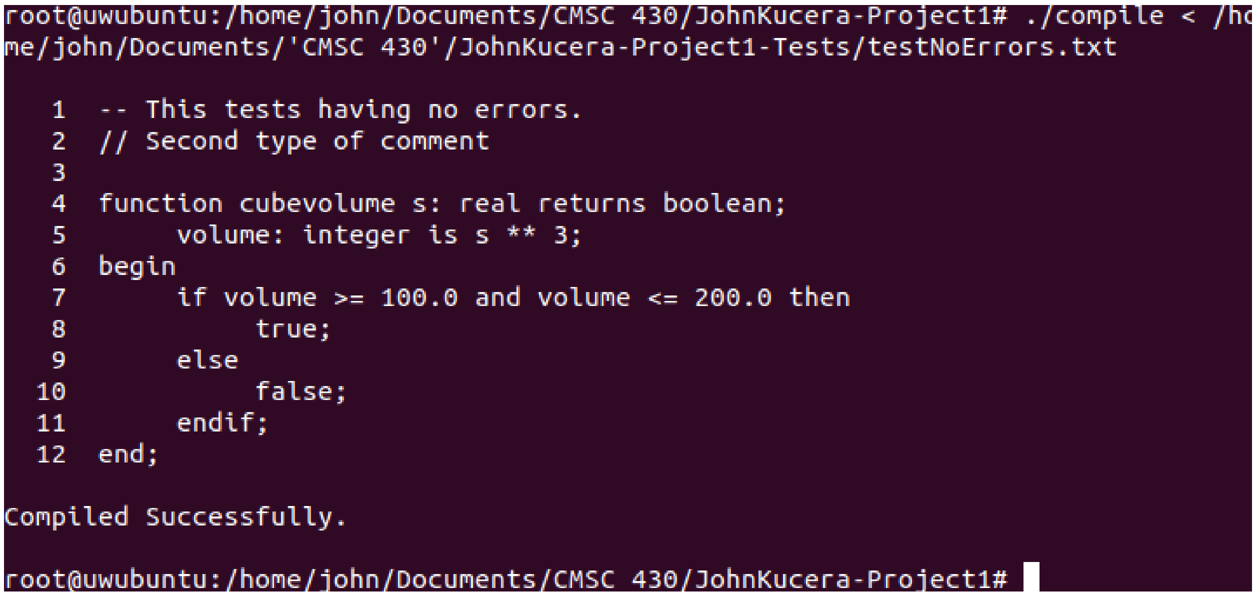
Aspect tested: An input file containing no errors

Input file: testNoErrors.txt. Contains a properly written program using the created grammar.



```
-- This tests having no errors.
// Second type of comment

function cubevolume s: real returns boolean;
     volume: integer is s ** 3;
begin
     if volume >= 100.0 and volume <= 200.0 then
          true;
     else
          false;
     endif;
end;
```

Compilation output: SUCCESS. No errors are raised or displayed; all input is properly printed as in the text file with the correct line numbering. Since there are no errors, the message “Compiled Successfully” is printed at the end.



```
root@uwubuntu:/home/john/Documents/CMSC 430/JohnKucera-Project1# ./compile < /ho
me/john/Documents/'CMSC 430'/JohnKucera-Project1-Tests/testNoErrors.txt

   1  -- This tests having no errors.
   2  // Second type of comment
   3
   4  function cubevolume s: real returns boolean;
   5       volume: integer is s ** 3;
   6  begin
   7       if volume >= 100.0 and volume <= 200.0 then
   8            true;
   9       else
  10            false;
  11       endif;
  12  end;

Compiled Successfully.

root@uwubuntu:/home/john/Documents/CMSC 430/JohnKucera-Project1#
```

Week 2: Project 1

Lessons Learned

A big lesson I learned for this project was simply to not panic when encountering so many new things at once. I'm still a little new to C++, and I hardly have any experience with Linux tools (although I do have experience with simple Linux commands). For the first week I was just sitting on flex and bison, not really understanding what to do with them or how to use them. When I first downloaded the skeleton code and saw the .l file, I had no idea what it was and just felt too inexperienced to have a good time with this. Of course, all that worry disappeared when I just went into the course material and Dr. Jarc laid out the skeleton code in full detail that I think any beginner could understand. Even if those videos were not enough, I found many outside resources that taught the basics of lex and yacc which solidified my understanding of my assignment. Simply put, it's intimidating when encountering a language(s) or tool(s) that are new that you feel you have to figure it out on your own. However, in reality, there are more resources and help out there than I usually expect there to be for me.

An improvement that could be made to all of the source code in general is some comments here and there so that there can be descriptions of all sections and functions. However, the original skeleton code had no in-code comments, so I did not add any of my own either (besides the comment blocks in the first lines). Otherwise, I think the code is all smoothly written right now with an efficient flow.