John Kucera

Prof. Renee McDonald

CMSC 430 Compiler Theory and Design

7 April 2021

Week 4: Project 2 Documentation

**Modifying the Syntactical Analyzer Generator (C++ with Flex and Bison)**

**Approaching the Project**

Just like the last project, when given skeleton code, my first step is always to understand everything about that given code. That way, I can modify it comfortably as if it were my own code. The videos from this week’s course material were again extremely helpful for me when trying to understand the code. Dr. Duane J. Jarc explained the code very well, going over the syntax and purpose of every section. I was not only able to understand how the code relates to the course readings, but also how this project ties directly to the lexical analyzer from two weeks ago.

In modifying the code, I knew the first thing I had to do was replace some of the skeleton code with my own from the lexical analyzer, then alter scanner.l so that it would not interfere with parser.y. For example, parser.y (new skeleton code) already holds the main method, so I had that part out of scanner.l so there would not be two main methods causing an error.

After adding to the token declarations in parser.y, the next big step was modifying the grammar productions. The first I made sure to include were syntax error detections, which I realized was only necessary in the productions near the root. Including error productions with the nonterminals function\_header, variable, and statement were all that I needed to cover all possible production errors.

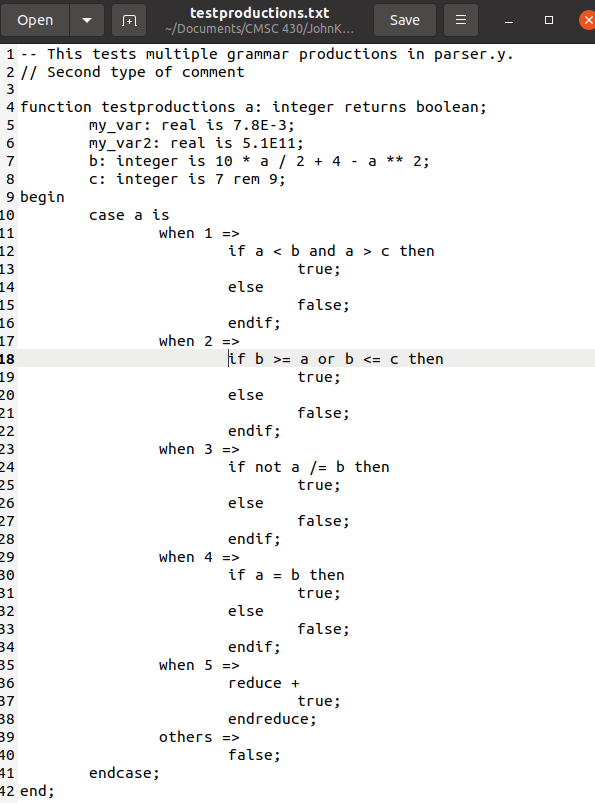
Another thing that took a lot of thinking and planning was maintaining proper precedence among the operators. I was confused at first on how to manage precedence with the bison file syntax but realized how simple it was when I took a step back and saw how the parser works overall. Adding precedence is simply like adding more branches. For example, an input term could be an OROP or something of higher precedence. It could then be an ANDOP or something of even higher precedence. Then a RELOP or something of even higher precedence, and so on.

**Test Cases**

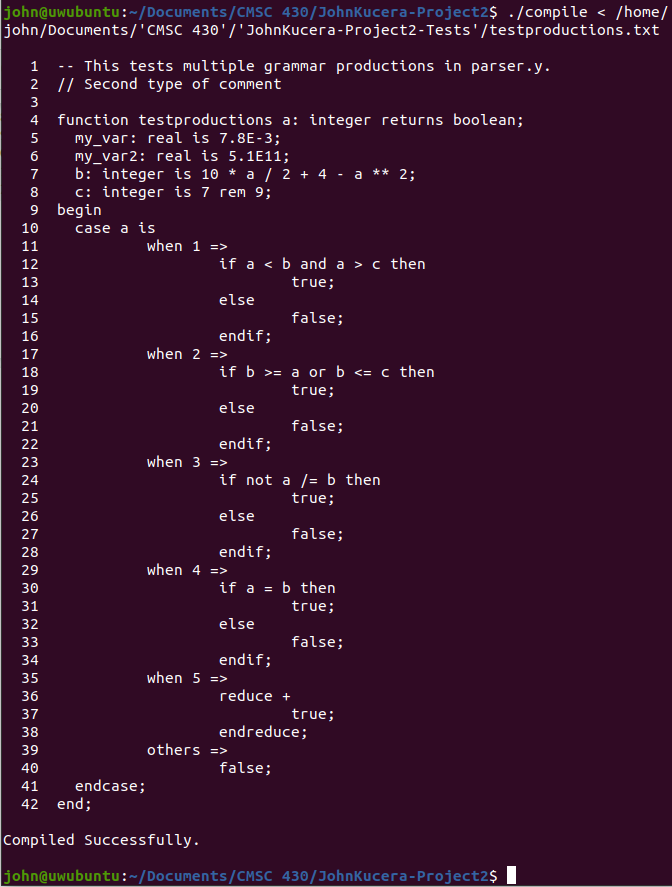
**(Test Case 1)**

Aspect tested: An input file containing most of the grammar productions

Input file: testproductions.txt. Contains most productions listed in parser.y, no errors



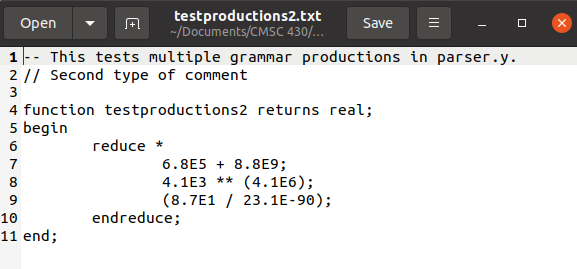
Compilation output: SUCCESS. All grammar productions are valid, last line prints “Compiled successfully.”



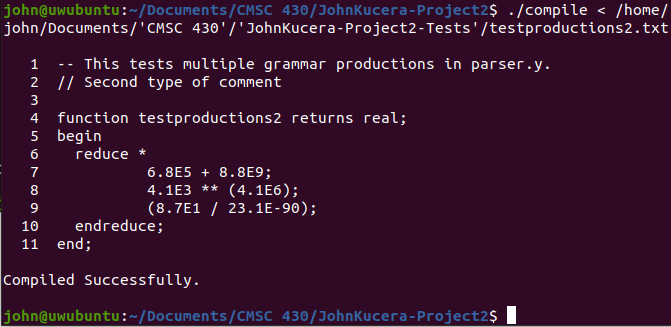
**(Test Case 2)**

Aspect tested: An input file containing more grammar productions

Input file: testproductions2.txt. Contains more productions listed in parser.y, such as having no parameters, no variables, and using parentheses. No errors.



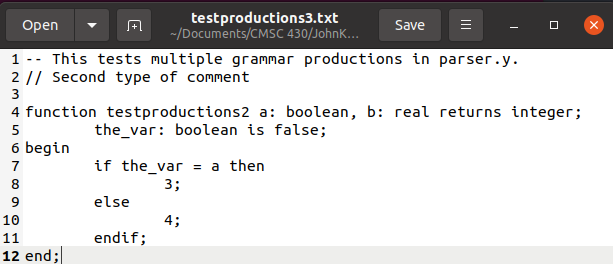
Compilation output: All grammar productions are valid, last line prints “Compiled successfully.”



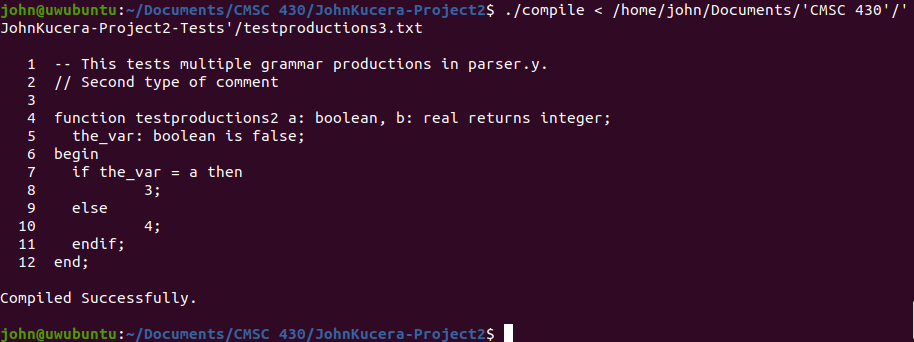
**(Test Case 3)**

Aspect tested: An input file containing more grammar productions

Input file: testproductions3.txt. Contains more productions listed in parser.y, such as having multiple parameters and having a single variable. No errors.



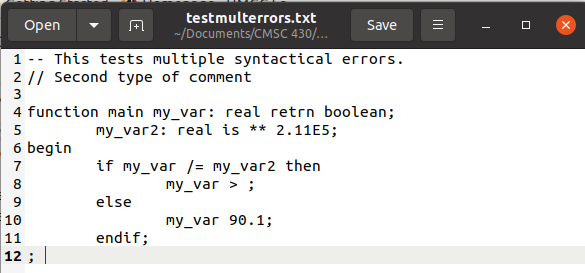
Compilation output: All grammar productions are valid, last line prints “Compiled successfully.”



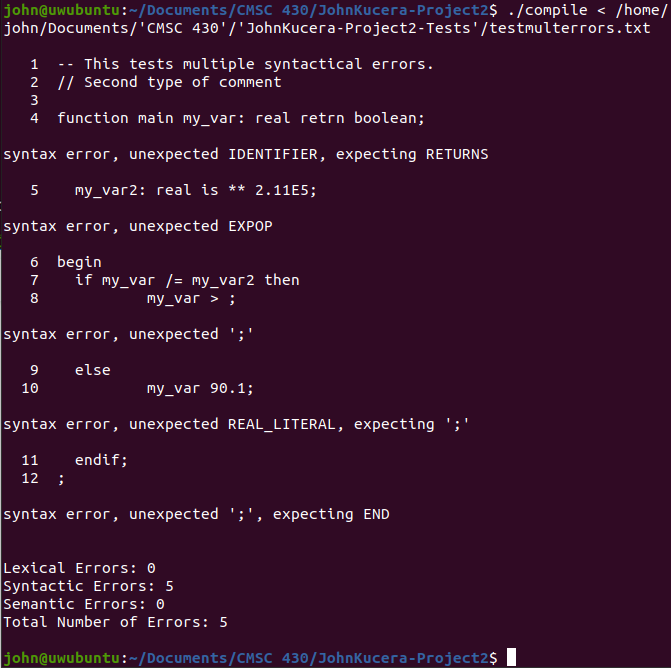
**(Test Case 4)**

Aspect tested: An input file containing multiple errors

Input file: testmulterrors.txt. Contains multiple lines where there are syntax errors, one being in the function\_header.



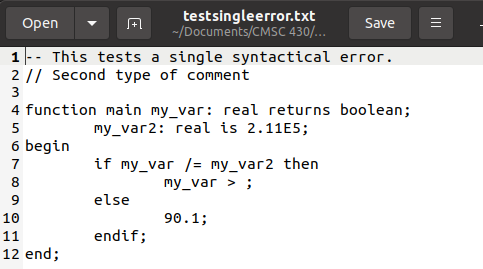
Compilation output: SUCCESS. Error messages are displayed after each line that has an error. At the end, a count of errors of each type in addition to total number of errors is displayed.



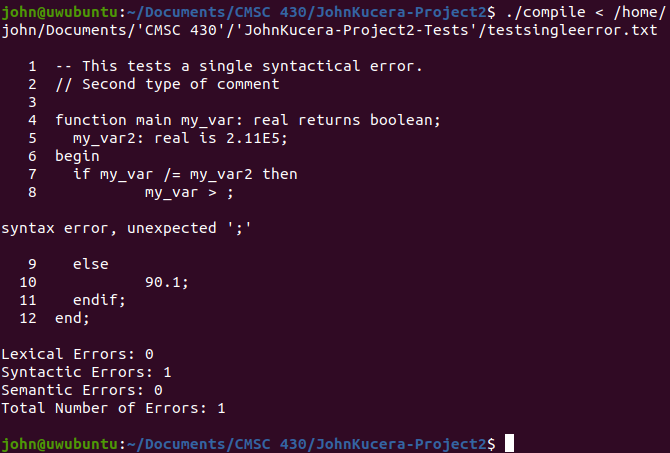
**(Test Case 5)**

Aspect tested: An input file containing a single error

Input file: testsingleerror.txt. Contains one line with a syntax error.



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



**(Test Cases: ERRORS)**

Aspect(s) tested: A variety of specific syntax errors in the grammar productions.

Input file(s): testproductions.txt, testproductions2.txt. I modified parts of the files that would raise errors, bit by bit, while inputting them into the parser each time (and then reverted the files back to normal. The attached test files do NOT have these errors). This table shows the error messages raised for syntax errors in each grammar production.

|  |  |
| --- | --- |
| **Production tested** | **Screenshot of error and error message** |
| function\_header, colon missing |  |
| function\_header, mispelled |  |
| function, text between function\_header and variable |  |
| variable, colon missing |  |
| variable, misspelled IDENTIFIER |  |
| variable, improper literal |  |
| body, missing begin |  |
| MULOP,  no int or real on one side |  |
| ADDOP,  no int or real on one side |  |
| EXPOP,  no int or real on one side |  |
| REMOP, no int or real on one side |  |
| case, int\_literal missing |  |
| case, improper arrow |  |
| or, improper expression on one side |  |
| and, improper expression on one side |  |
| if, missing semicolon |  |
| /= relop, improper expression on one side |  |
| = relop, improper expression on one side |  |
| reduce, missing operator |  |
| others, improper statement |  |
| case, missing endcase |  |
| missing end |  |
| not, improper expression |  |
| parentheses, improper expression inside |  |
| reduce, missing endreduce |  |
| if, missing endif |  |

**Lessons Learned**

A big lesson I learned from this project was that it is sometimes a good idea to take a step back from the individual code I am working on and see the entire program as a whole and how it flows. This was evident when struggling in the precedence section that I previously mentioned. I was so focused on the fact that I do not know bison syntax well and that there must be some function or logic that I was unaware of. In reality, I forgot that the grammar production section is representative of a bottom-up parser. It is a root with branches and leaves. So, I just added more branches on top of branches to represent the precedence. I would not have come to this conclusion if I did not see the program as a whole and how much it should flow like a parser.

One thing I would like to improve on is also that precedence section. My non-terminals are labeled just as “precedence\_1” and “precedence\_2”, higher numbers representing higher precedence. For the sake of clarity, I wonder if there is some other naming standard for precedence in grammar productions. On a related note, I also wonder if there is a more convenient way for implementing precedence in these grammar productions. I understand that it is a parser and it is usually best to stick to the root-branches-leaves flow of the program, but it still makes me add seven different grammar productions just for operator precedence. If I knew a way to simplify this section of the code, I definitely would do it.