**Approach**

When beginning this project, I examined the skeleton code in both listing.cc and scanner.l to understand how Flex was being used to process input and produce tokens, how errors were being recorded in listing.cc, and how the overall project structure worked with scanner.l generating scanner.c during compilation. I identified that the main tasks involved modifying or adding regular expressions in scanner.l and ensuring that listing.cc captured and displayed errors as desired.

I then adopted an incremental approach. First, I started with basic tokens such as keywords (else, elsif, etc.) and the “not” and “or” operators (“!” and “|”, respectively) to confirm that additional tokens were correctly parsed and there were no problems with using the Makefile on the updated source code. Next, I extended the rules to include more patterns, in the order suggested in the ‘Project 1 Approach’ pdf document, such as the additional relational operators and the minus and divide symbols. I also introduced support for both // and -- comments, verifying that each comment line did not produce tokens but did cause the line counter to advance without error.

Throughout this process, I tested each new feature by compiling the scanner, providing specific input files, and checking that the output and errors matched expectations. In parallel, I refined error handling and output by editing the appendError, lastLine, and displayErrors functions, as required by the project description. With each iteration I was cautious to ensure I thoroughly re-examined the previous test cases to ensure they were still valid.

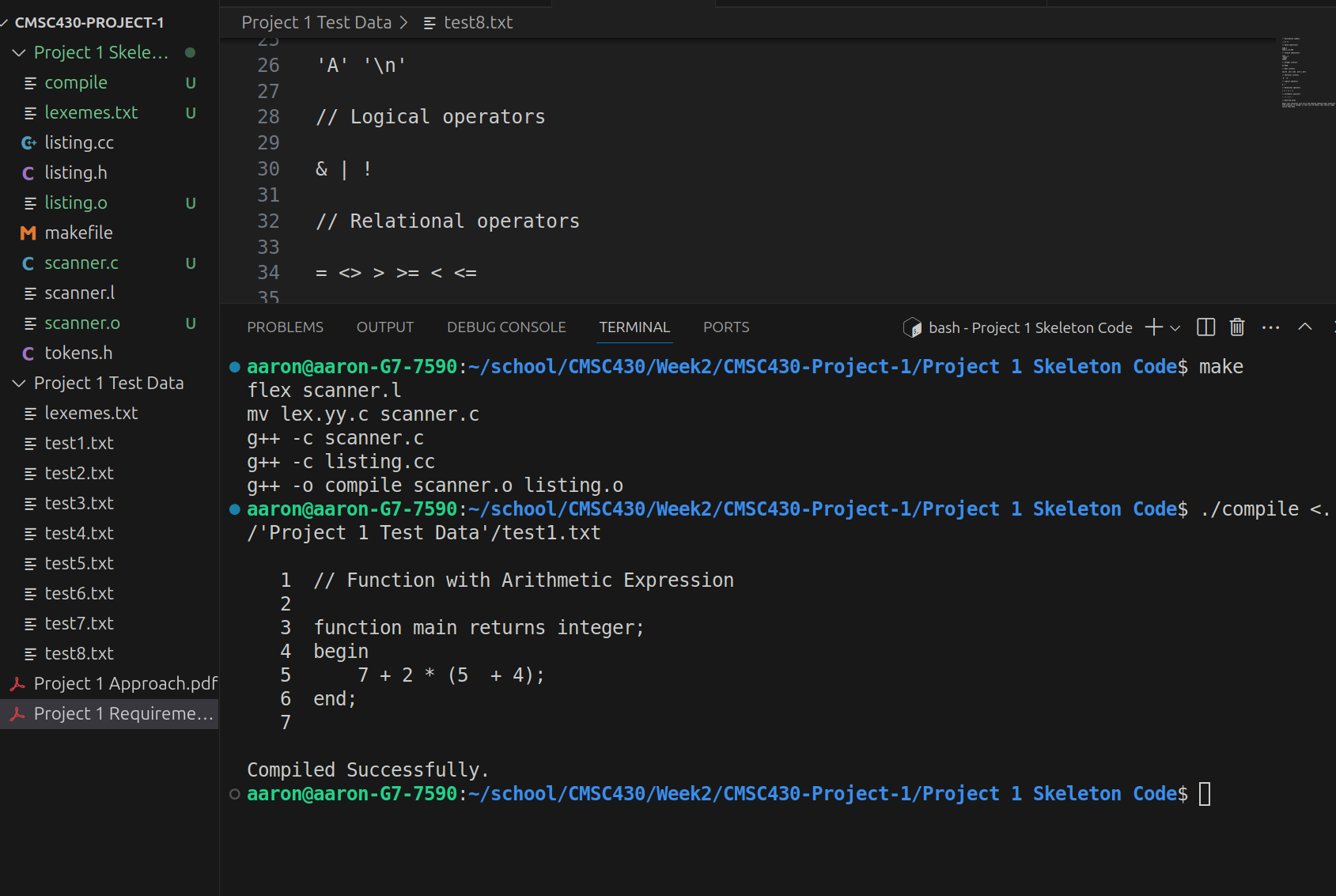
Because regular expressions can be complex, I spent significant time adjusting patterns to handle edge cases. I verified numeric literals carefully, ensuring that exponent parts, hexadecimal notation, and decimal points were all recognized. This iterative testing process kept the codebase relatively stable while allowing for continual refinement.

**Test Plan**

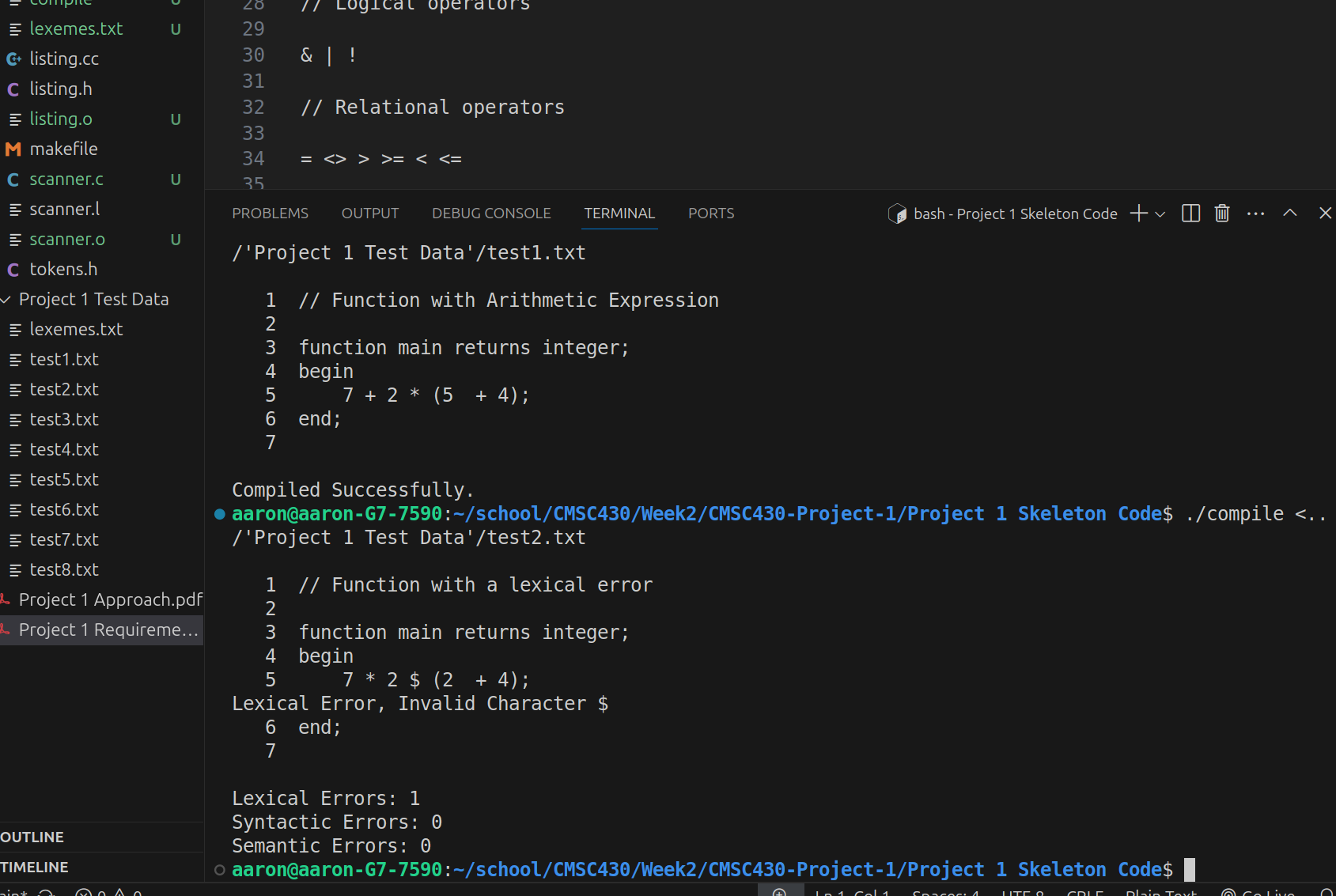
I compiled the lexer using the Makefile to produce scanner.c, followed by an automatic compilation step, again, thanks to the Makefile. This produces multiple object files and an executable, able to be run by feeding in the attached test text files.

Because the test text files were so incredibly comprehensive and included all the symbols which were supposed to be added, they served as the perfect examples of the language. Following the testing order mentioned in the ‘Project 1 Approach’ pdf document, I was able to verify that I was implementing additions and changes correctly as I made them. I used all the text files to verify I properly implemented every element mentioned in the requirements document. Please see below for a screenshot of each test text file going through lexical analysis.

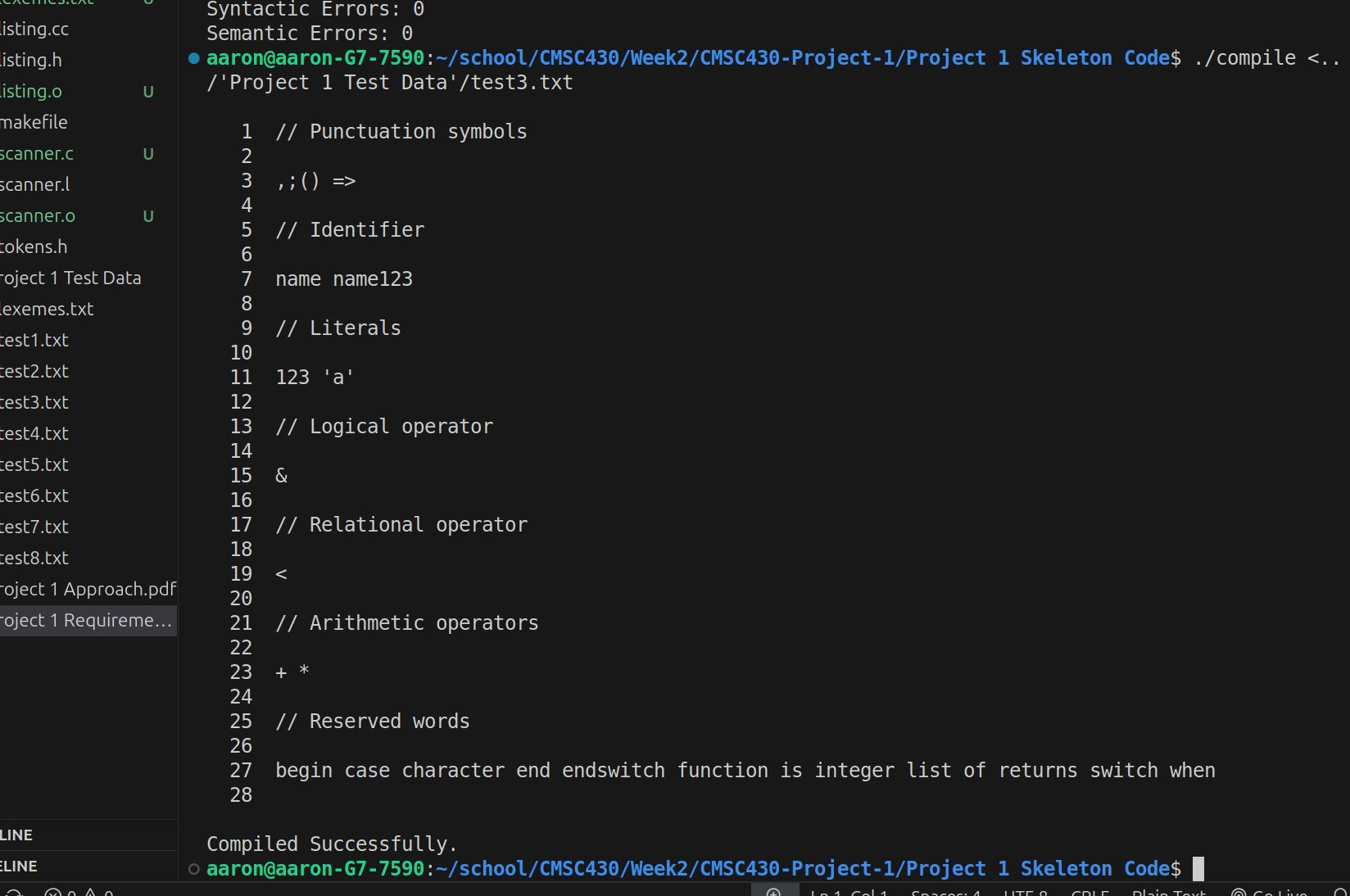
Test1.txt



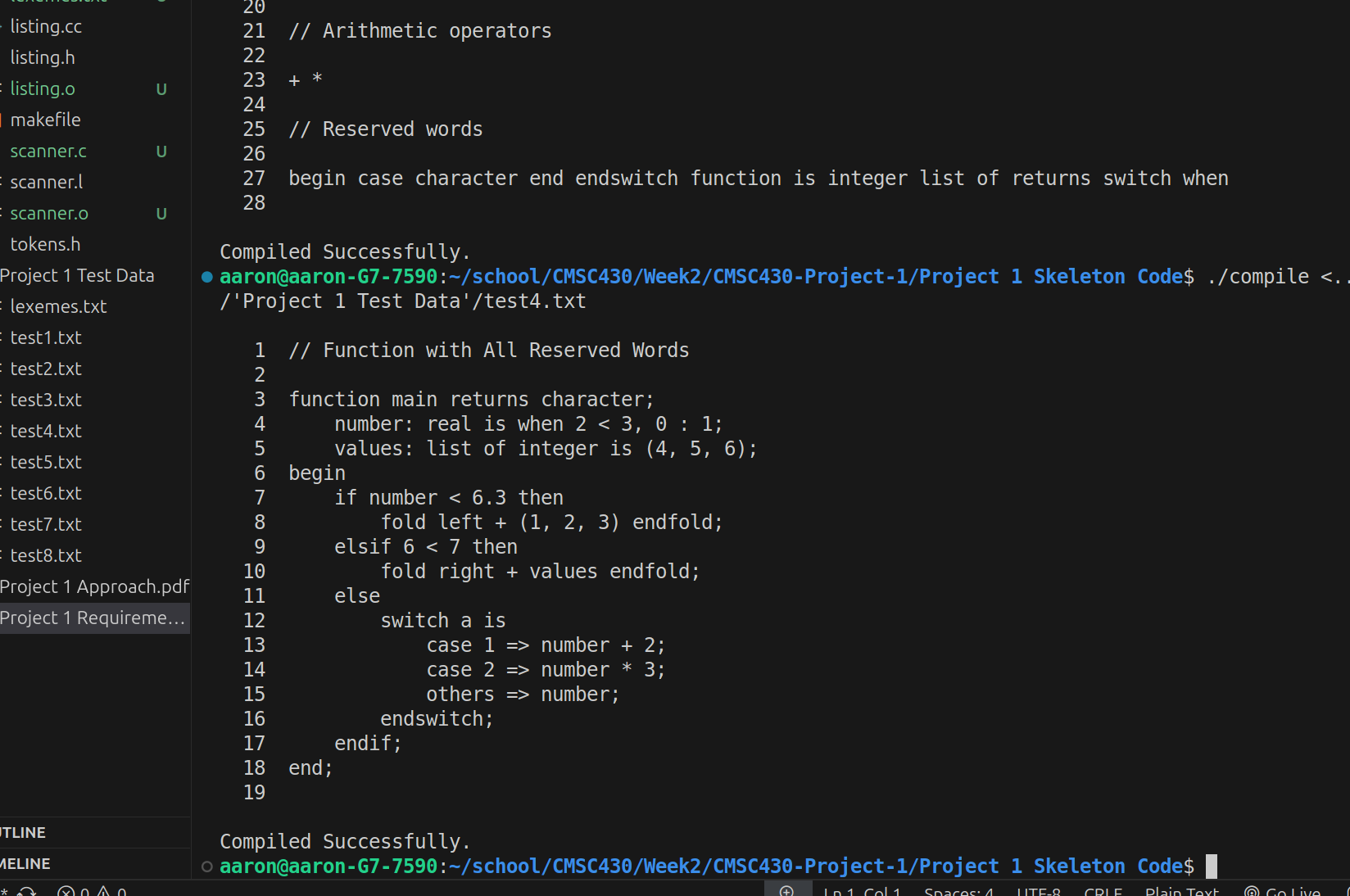
Test2.txt



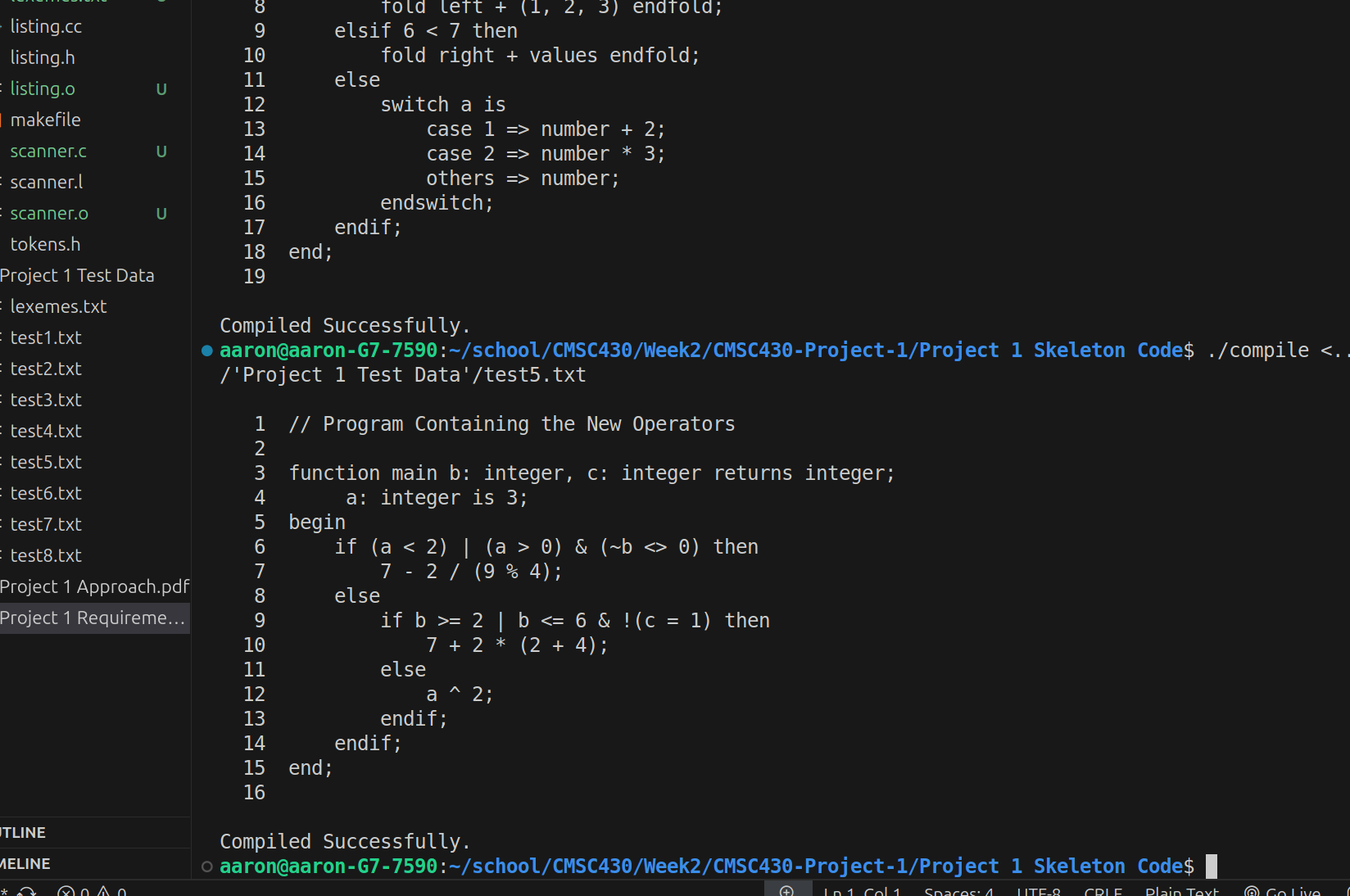
Test3.txt



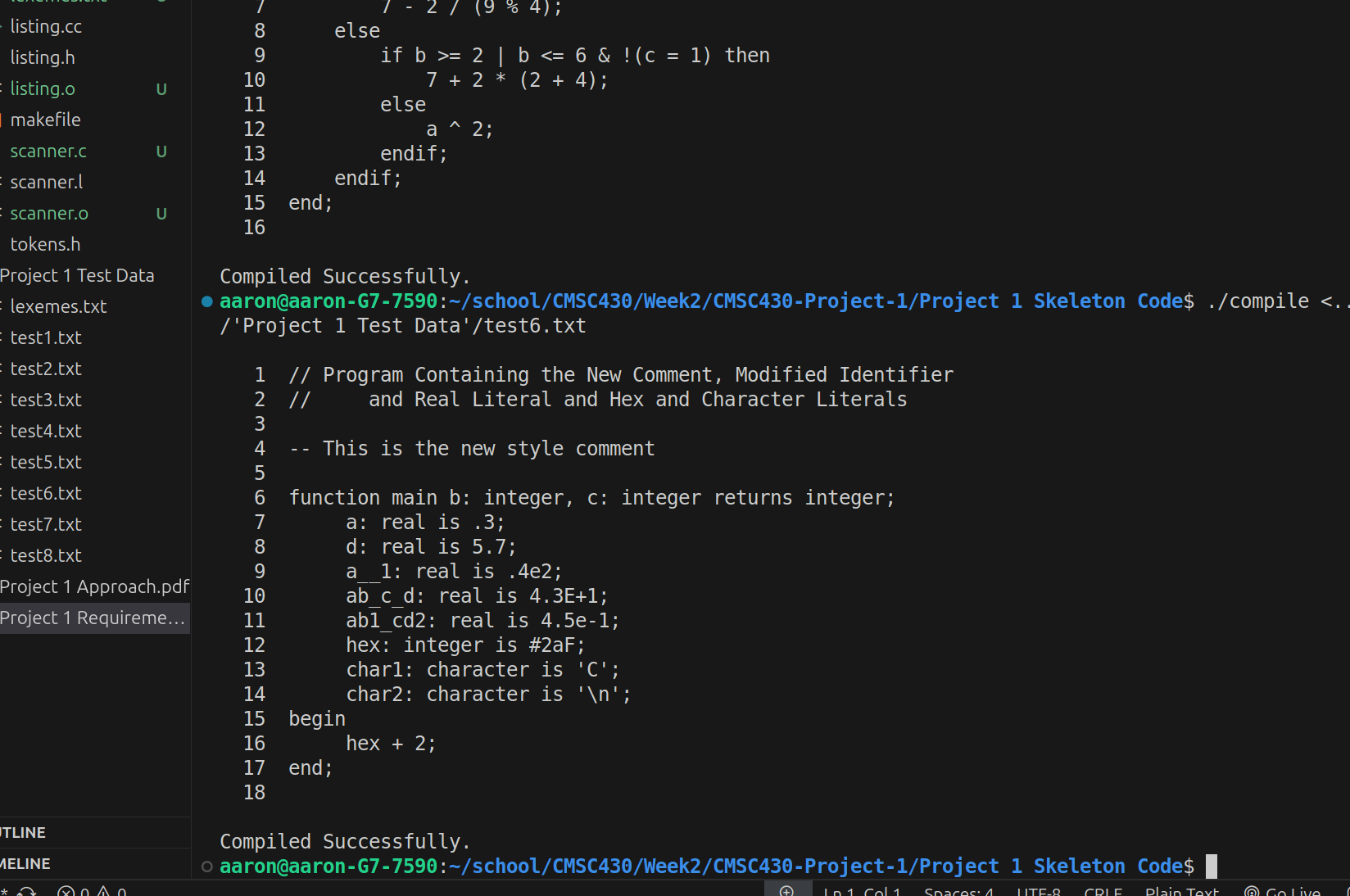
Test4.txt



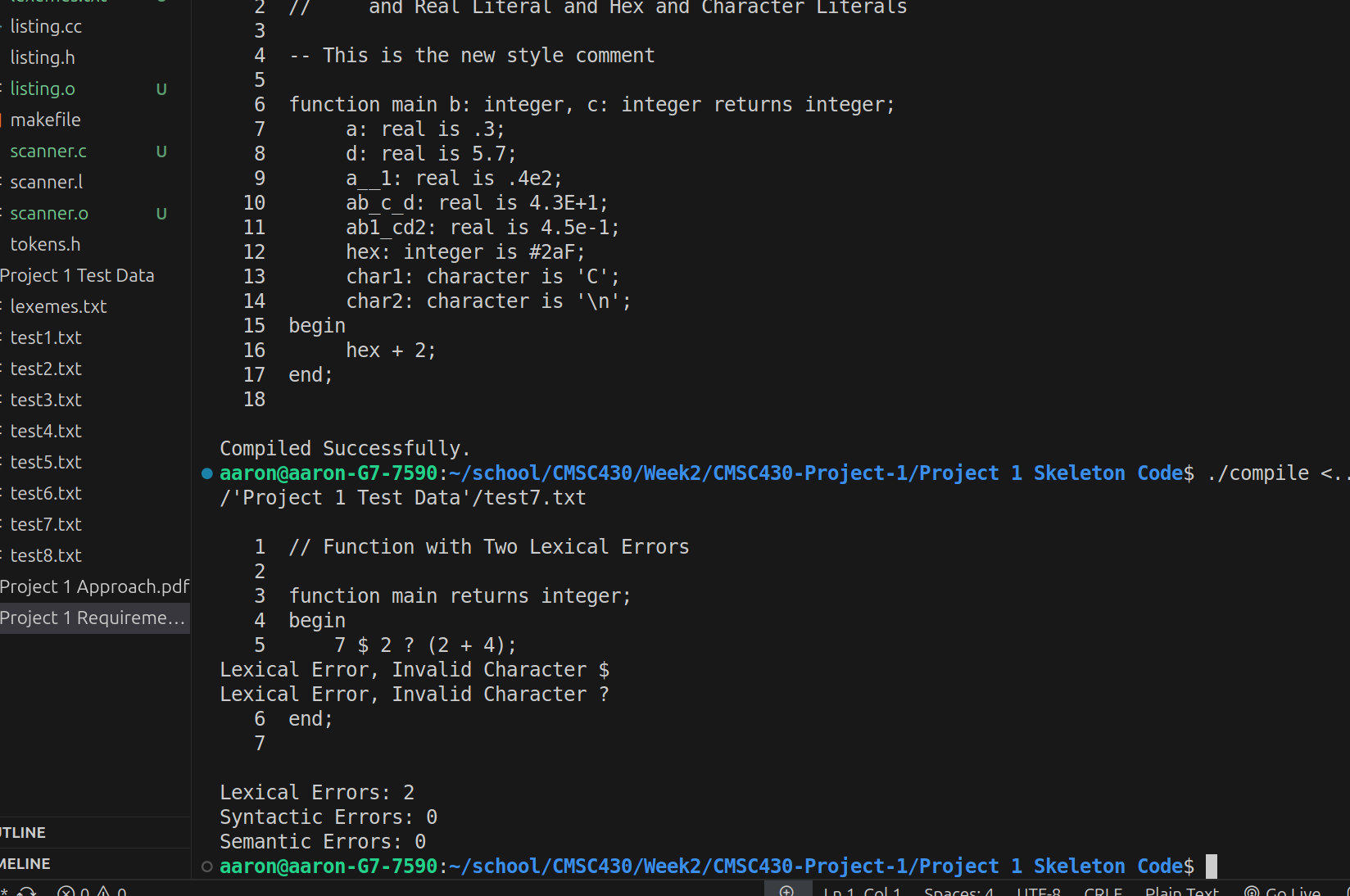
Test5.txt



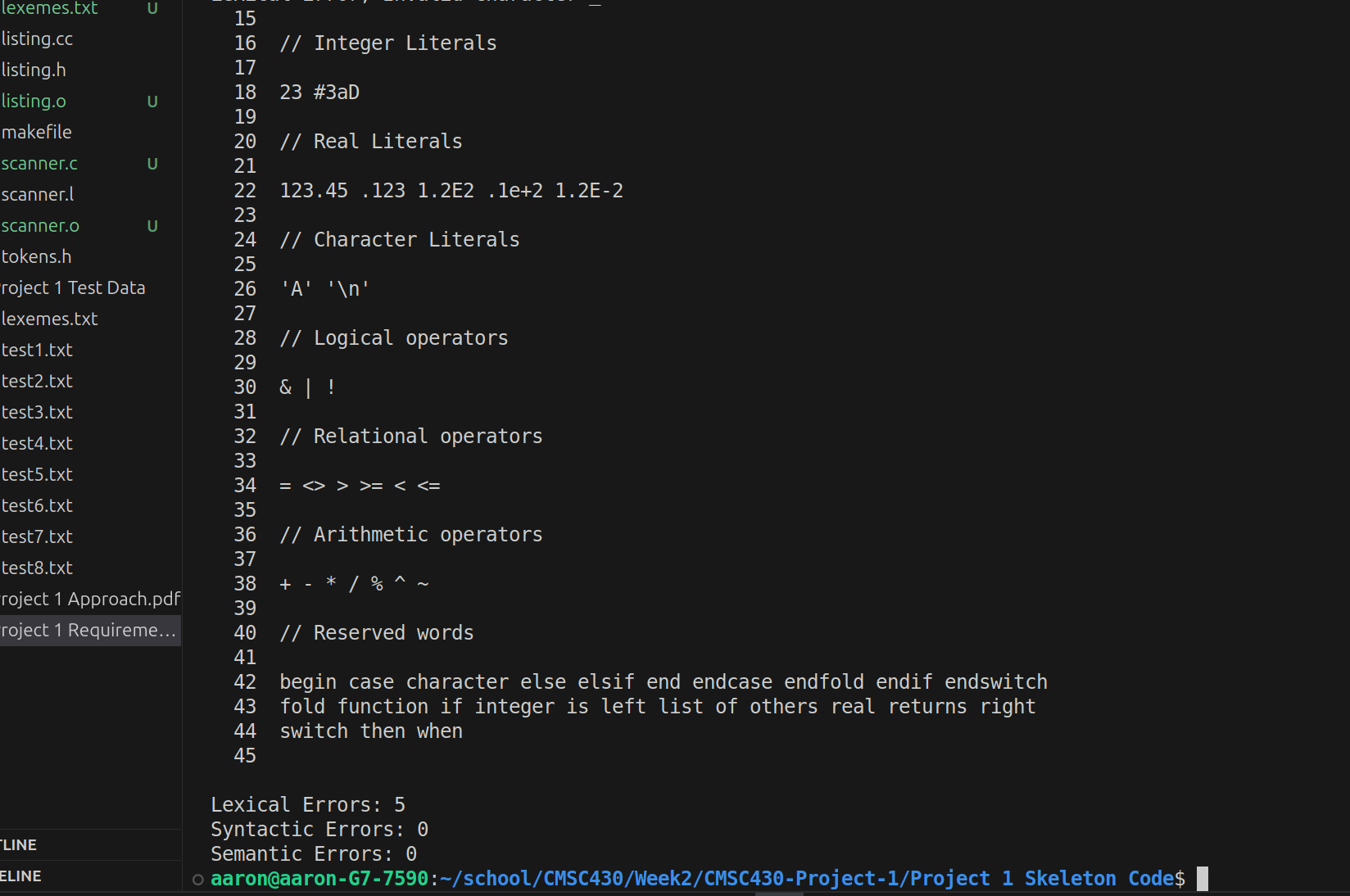
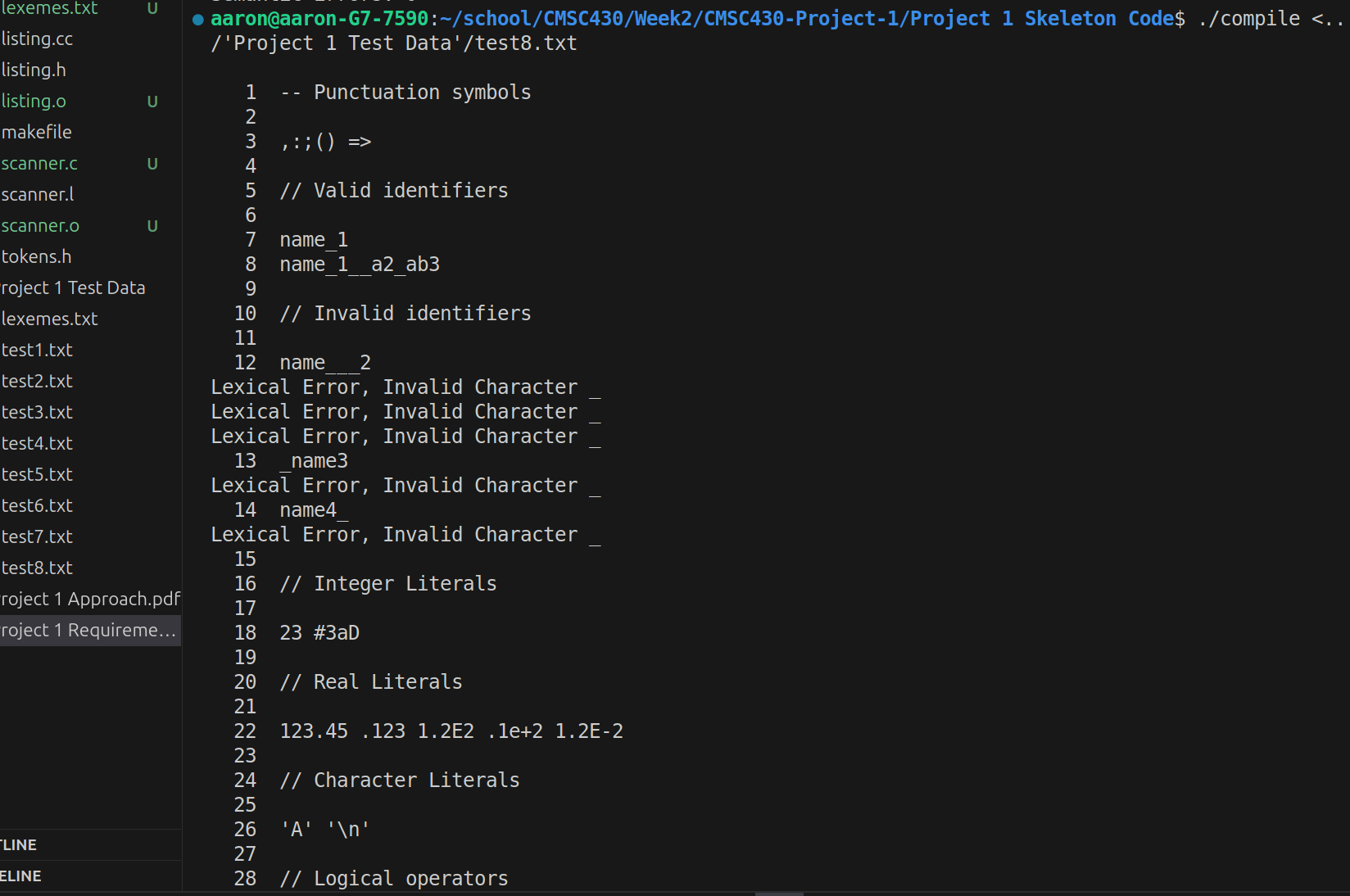
Test6.txt



Test7.txt



Test8.txt



**Lessons Learned & Improvements**

One of the primary lessons I took from this project was a fundamental understanding of Flex and how it processes input text by matching rules in descending order of priority. This experience also clarified the way regular expressions are combined with C++ code. I also certainly improved my comprehension of enumeration and tokens, appreciating how each token type must be carefully returned so that further compilation stages can handle them properly. By experimenting with enumerations for different kinds of errors, I realized how crucial it is to keep track of error categories (lexical, syntactic, semantic) and link them to corresponding messages and counters.

I also became much more comfortable with constructing and modifying regular expressions to handle various edge cases. Through repeated trial and error, I recognized that either negative lookahead or simpler pattern-plus-code approaches can forbid certain sequences of underscores or invalid characters, but are sometimes not compatible with Flex, so alternate approaches may be needed. I discovered the importance of testing very small changes to patterns right away so that issues do not accumulate and force backtracking. In the future, I would consider modularizing the scanner further by grouping numeric, identifier, and other token-related patterns more distinctly, and I might also integrate automated tests or specific output checks that validate the output against more varied known correct results. A more thorough integration with a Yacc/Bison grammar would be another improvement (and is most certainly coming according to the Syllabus), ensuring the tokens flow directly into higher-level parsing. Overall, I have gained a stronger foundation in lexical analysis, which will serve me well in any future compiler projects that rely on Flex or similar tools.

**Conclusion**

By stepping through these sections, it should be clear how the project was approached incrementally, how I tested each aspect of the scanner, and the insights I gained regarding both Flex usage and error handling strategies. The combination of code changes and frequent iterative testing helped maintain a stable codebase while expanding functionality and reliability, particularly in regard to regular expressions and enumeration-driven token returns.