Tommy Her (herxx393)
Jennifer Lieu (lieux007)
Team Zubat

Writing Assignment 1: 6 Primary Issues of Iteration 1

Introduction

As the first implementation of our code, although our code ran accordingly to the specifications required of the iteration one syllabus, our code exhibited several faults. Some of the primary issues we had in the first iteration of our code was that there were several redundancy and inefficiency issues that could have been easily minimized. Of the six commonly found issues involved in Lab 8, our code exhibited issues: *one*, duplications of testing code; *two*, repetitive calls to *make_regex()*; and *five*, multiple creations of *regex_t* pointers for each regex, although our code was stored in an array.

Issue One

Issue one characterized issues of having duplicated test cases for individual token regular expressions and the test cases involving using the *Scan* function to test whether the correct *TokenType* is returned. In our *scanner_tests.h* file between lines 56 - 386, we had redundancy issues where we had created individual test cases to both *make_regex()* and test the regular expressions for all of the *TokenTypes*. The first redundancy with this is that in *regex_tests.h* in lines 27 - 49, we were already given test cases that utilized *make_regex()* to test whether or not the correct regular expressions were being correctly created. The second redundancy involved *scanner_tests.h* in lines 416 - 642 where we were required to create test cases that would utilize our *Scan* function to determine whether it was assigning the correct *TokenType _terminal* to a given string. Another issue we had was that although we had an array that essentially stored the created *make_regex()* expressions, we were recreating it multiple times in *scanner.c* for both the *Token_Generator* function in lines 84 - 183 and the *Scan* function in lines 281 - 379. Some of the primary concerns involving these duplications is that it's inefficient and extremely redundant to call *make_regex()* an additional 40+ times to create test cases to test the token regular expressions. Since *make_regex()* is already utilized elsewhere, it would be more efficient to simply reference the array that holds the created *make_regex()* token expressions rather than recreating them each time. The solution we had decided upon to solve this issue was that we removed the 40+ token regular expression tests from lines 56 - 386 in *scanner_tests.h*, since our *Scan* tests in lines 416 - 642 already made comparisons between the correct regular expressions to an appropriated string. In addition, since there was redundancy involving multiple instantiations of the *str_regex_token_array[45]* (lines 84 - 183 and lines 281 - 379 in *scanner.cc*) which held the stored *make_regex()* expressions, we decided to remove them both from the *Token_Generator* and *Scan* functions; instead, we created an array of it under the *Scanner* class where it would be instantiated by the *Scanner* constructor and could be easily accessed by other functions and classes.

Issue Two

Issue two concerned issues of making redundant calls to *make_regex()* in our *Scan* function. With this issue, our involved calling *make_regex()* for all of the *TokenTypes* whenever the scanner moved to a new word or new character. In our code this can be seen from lines 85 - 130 and also lines 273 - 318 of *scanner.cc* where we made regex points to all of the regex expressions made with the *make_regex()* function every time *Scan* function were called. Some other concerns involving redundant use of *make_regex()* was also stated in *issue one*. The changes we made to fix *issue two* started with initiating the *regex_t* pointers in the *scanner.h* file and defining the *regex_t* pointers in the constructor for the *Scanner* class in *scanner.cc*. By moving all of the definitions to the constructor, the code will then only create the definitions for the *regex_t* pointers once when the *Scanner* class is actually constructed. The reason that it's undesirable for the *make_regex()* to be called every time that a new word or new char is scanned is that it would make an unnecessary number of calls to *make_regex()* where it could have

otherwise remained as a constant. After an initial call to *make_regex()* the regex expressions should remain constant because nothing in the code is modifying the regex expressions once it's called and recalling *make_regex()* each time is a waste of code length and processing time. By moving the *make_regex()* calls to the constructor, the *make_regex()* expressions are only made once because the scanner is only constructed once which saves code length and also processing time for an overall more efficient program.

Issue Five

Issue five involved making a *regex_t* pointer for every regex expression that was made with *make_regex()* instead of putting all of the regex expressions into an array. In our code this can be seen from lines 90 - 134 and lines 286 - 330 of *scanner.cc*. The changes we made to fix *issue five* involved making changes to a larger portion of the code that we also fixed earlier involving *issues one and two*. Instead of containing the *regex_t* pointers and *TokenTypes* associated with them in a single structure, we decided to make just one array containing the regex expressions indexed using the enumerated *TokenTypes*. The first step was we deleted the earlier initialization of the *make_regex()* from the constructor and placed them in a static array and indexed them by the enumerated *TokenTypes* to prevent potential problems with *issue four*, where if the order of *kVariableKwd* and *kEndKwd* are switched in the *scanner.h* additional problems would arise. In addition, we updated code in lines 203 (note: *Token_Generator* function was removed), 206 (note: *Token_Generator* function was removed), 407, and 410 of *scanner.cc* which had used the previous structure to match with the text using the newly created *regex_token_array*.

Issue Three

Our code did not exhibit problems of *issue three* because we did not create a redundant enumerated *TokenType* array. The reason this is an issue is because there is already a declaration of the array in *scanner.h* as enum *kTokenEnumType* and it's unnecessary to create an additional copy of it when you can simply reference to the array. The main purpose of the array is for identifying the *TokenType terminal_* of the characters that are scanned from file. However, in some cases this can be easily bypassed by using the original enumerated *TokenTypes* and using the *TokenType*s as indices for the regular expressions and indexing directly into the *TokenType* array.

Issue Four

We also did not have any direct problems associated with *issue four* because by fixing *issues two* and *five*, we were able to prevent issues that would occur by switching the order of the enumerated *TokenTypes*. Essentially if switching the order of the enumerated *TokenTypes* (i.e, *kVariableKwd* and *kEndKwd*) affected the code it, would mean that the code is likely utilizing some form of magic numbers which is often considered ill practice with coding. In place of using magic numbers, we used the enumerated *TokenTypes* as the indices for the regular expressions which prevents issues from arising when the orders of the enumerated *TokenTypes* are switched.

Issue Six

There were no problems with *issue six* because in finding the *TokenTypes* of the words or characters, we used the enumerated *TokenTypes* as indices instead of magic numbers, as stated previously in *issue four*. This is a common issue because magic numbers are generally advised against because if the code is modified even slightly it could change the results of the code if a number references to the wrong *TokenType*.

Conclusion

Since our code had various problems that involved *issues one, two and five*, we were able to make updates to the code that solved many of these issues. The first issue we had was the redundancy and duplication of our test cases that were written in *scanner_tests.h* involving the individual regular expressions for each of the *TokenTypes*; since there were tests already created in *regex_tests.h* that tested *make_regex()*, it was unnecessary for us to make additional calls to *make_regex()* when testing each of the individual regular expressions. In addition, since we had created a function called *token_type_tester()* that tested the *Scan* function to the correct corresponding *TokenType terminal_*, it was unnecessary to make additional calls to *make_regex()* since there was already a call to *make_regex()* in the *Scan* function that created each of the token regular expressions. By removing these redundancies for the tests, we were able to minimize the processing times since our *token_type_tester* already tests for each of the individual regular expressions and *make_regex()*. Value-wise, this update allowed for slightly faster processing times since it removed 40+ redundant test cases that were originally being evaluated in *iteration 1* and *2*. By fixing *issue two* problems, we were able to a solve a number of problems with redundancy and efficiency in our program. We had removed the multiple instantiations we had for *str_regex_token_array[45]* that were called in *Token_Generator* and the *Scan* function, since the array would make over 40+ calls to *make_regex()* any time either of the functions were used, which was extremely inefficient and extremely redundant when it could just be called once for each of the types and referenced. Instead, we created a separate array that would be instantiated in the *Scanner* constructor, which made calls to *make_regex()* only once for each token; by doing this, it made it possible for the *Token_Generator*, *Scan*, and *token_type_tester* to easily reference the array without redundantly calling *make_regex()* to create the regular expressions. By doing this, we saved quite a bit of processing time, efficiency, code length, in addition to creating a better organization overall for the program. The final problem that was resolved was *issue five*, where there were too many named *regex_t* pointers created for each of the 40+ *TokenTypes*. Although we did have an array that held the various *regex_t* pointers, it also held the regular expression and *TokenTypes*. How we solved the issue was by creating a new array, *regex_token_array*,that exclusively held the regular expressions which we indexed by the enumerated *TokenTypes*. By making these changes it became more efficient to reference each of the various *TokenTypes* when needed, rather than pointing to each of the individual *regex_t* pointers associated with the tokens. As with *issues three, four* and *six*, we did not have any of theses issues reflected in our code, however, with the changes that we had made to our code, we wrote it in a manner to prevent any of these underlying issues from occurring. Thus, with all the updates we made to our program based on these six components, our program is now more efficient, shorter in length, more legible and organized overall.