Mark Dasuki 4330067
Rong Lei 4423208
CSCI 3081W
4/07/2014

Writing Assignment 1

Even though our code does compile and pass all of the tests, it does not mean that the code is efficient. The code can still definitely be improved in many ways, which includes removing excess or unnecessary codes, making the code more readable, making the code less error prone and so on. The problems we identified in our code were that we had duplication in our test and implementation code, we called makeRegex every time we called the scan function, and we created a regex_t pointer for each regex.

In our test file and scanner.cpp file we do have duplication of codes. In the implementation file, we used duplication of makeRegex and matchRegex to create regular expressions of individual terminals and match individual words of the text with the terminals. This extended our implementation by over 300 lines of codes. A way to solve this mess was to use a for loop. Since we have an array of all the return regex_t from makeRegex, we used the array as an iterator and loop through all the elements to match the individual words in the text with the terminals by use matchRegex on the individual elements. By doing this, we were able to get rid of all the duplication in our implementation file. But for our test file, we believe that having duplication of code is fine as long as each test varies in their names. Since we are testing regular expressions individually, we have no choice but to use the same codes repetitively but use different input strings to test and see if we get the expected output using the same Terminal function.

Our scan function of our Scanner class does make calls to makeRegex for all the terminals that are in the enumerated type terminal_t. This is inefficient because the pointers to those regular expressions will never change, so therefore we do not need to make the makeRegex calls whenever we need to use the scan function. A way to solve this problem is to declare call them outside of the scan function and make them as a global variable so the scan function have access to all the regular expressions that are needed to do the scanning. Another way to solve this is include an array of all the makeRegex pointers in our Scanner class. With this implemented, we only make that array when we initialize an instance of a Scanner class object. This is done by making the default constructor to create the regex array when we declare an Scanner object.

We do not have an array of all the terminals, it is redundant because the whole point of that array is to get the index of the terminal, which is also the return value of our enumerated type terminal_t. When we refer one of the terminal that exist in terminal_t, the return value is the integer that is correspond to the position of that terminal. Also, if we want to covert an integer to a terminal_t type, we can cast the integer to terminal_t type, that is very similar to get the return value of the enumerated terminals.

Changing the position of variableName and endKwd will change the priorities of these two terminals. For example, if a keyword “end” was in the text, instead of labeling its terminal to be an endKwd terminal, our code would label it as variableName, because if we changed the order, it prioritizes variableName first. This happens because during our check for terminal type (in scanner.cpp, lines 71 to line 76), our code only change the word “end” to a endKwd only if numMatchedChars is greater than maxNumMatchedChars. Since the numMatchedChars from matchRegex of variableName

and endKwd are equal, our code will end up labeling “end” as a variableName instead. A way to fix this is have another if-statement in the part of the code that checks if the word is an endKwd. If it turns out to be an endKwd, then label its terminal to be that. Another way that might work is to have variable name in position 0 of the enumerated type terminal_t and change the code so that when numMatchedChars is less than or equal to maxNumMatchedChars it will also change the current lexeme's terminal label. This way we will know that if a word that's a variableName with the numMatchedChars is equal to numMatchedChars of any of the other keyword in the terminal_t type, the code will label those as the keyword instead of variableName.

Our code does exhibit the problem of creating a regex_t pointer for each regex instead of putting them in an array. This problem is exhibited from lines 51 to 214 in scanner.cpp. They are used in our current code, as our current code would not work without these regex_t pointers. The problem with this method is that while it produces correct results, it was time consuming to create and due to the large amount of variables, it means we have to keep track of these large amounts of variables. This can cause potential errors in the code, as it is also harder to keep track of. Another problem is that since we had a large amount of if statements when matching the regexes, so those if statements are unnecessary when we can loop through the array. We can fix this problem by putting all of the regexes into an array. Then, when we go through the text, instead of matching the regex case by case by having many if statements, which is what we have, we can loop through the regular expressions when we are matching them to the text since we now have an array of regular expressions. This solution avoids creating a regex

pointer variable for each regex as we are putting them in an array and looping through that array instead.

Our code does not exhibit the problem of using integer literals in place of tokenType values. One of the problems with using integer constants to identify the keyword is that it makes the code more difficult to read, since it is essentially hard coding. Having an integer constant somewhere in the code does not tell any information about which keyword it was supposed to signify. Another potential problem that can also occur is that since the integer literals are based on the order of the tokenEnumType, if we change the order we also have to change the all of the integer literals that we use, or else we would have errors. So keeping and using integer constants to identify keywords can create potential errors in the code. We can get rid of the literal constants by using the tokenTypes values instead of the literal constants.

In the end, we were able to fix the problems that occurred in our solution. The problem of having duplication in our testing code was that is increased the amount of unnecessary code, as it extended the implementation of our code by 300 lines. Therefore, by fixing the problem, we improve the readability of our testing code. It is inefficient to have our Scanner class call makeRegex many times. Thus, by fixing this problem, we improve the efficiently of the code since we don't have to call all those matchRegex each time we use the scan function. The problem of having a regex_t pointer for each regex instead of putting them in an array, is that we had to keep track of a large amount of variables, and we had an unnecessary amount of if statements which makes the code longer and more difficult to maintain. By fixing this problem, we have code that is easier to maintain and debug. Our code has improved slightly as it is a bit more efficient, more maintainable, and is cleaner and easier to read.