Our first iteration did exhibit some of the inefficiencies brought up in lab 8. For the first problem in lab 8, our tests for each of the token types did display a fair amount of duplicate code. Therefore, further modifications to these tests in the next iterations would be more complicated based on the redundant structure of the tests for each token type. Our iteration one code also exhibited the inefficiencies mentioned in problem two in that the scan function used the makeRegex function each time it was called. In this same vein, as the makeRegex function was being called every time, our code created pointers to these regular expressions rather than simply storing this information in an array. These problems described in lab 8 were present in our code, but all six of the problems from lab 8 lead to many inefficiencies in implementing this project.

The first problem that could appear in this project is the use of redundant or duplicated code in the code to test the individual token regular expressions. This issue was present in both scanner_tests.h from line 55 to 395 and in regex_tests.h from line 17 to 341. This practice is an issue because it creates a needlessly long-winded script and creates any further modifications to these scripts more difficult and tedious. As exemplified by the line numbers of where these issues were present in our iteration one files, each of these groups of tests for the token types span more than 300 lines. The sheer amount of code within these files makes debugging more laborious and impractical as each of the tests has the exact same structure. In each of these iterations, any changes made to these tests would require parsing through these 300 lines of code to make edits. In order to fix this problem, the general structure of these tests can be written into another function that is then called by each test. Therefore, any changes to the token type tests will be much easier to implement in further iterations.

Problem 2 involves making function calls to makeRegex for each regular expression every time the scan function is called, which can be made more efficient by doing the function calls only once. Our code exhibited this problem in lines 82-132 in our original code, right at the top of the scan function. Since the scan function is called many times during the tests, creating new regular expressions for every call of the scan function was a major inefficiency. The problem was fixed by creating an array that holds the regular expressions. The array is initialized and filled with the regular expressions in the constructor for the scanner, so it is only done once when the scanner object is created and any time a regular expression is used, it simply references the array of pre-made regular expressions.

The third problem described in lab 8 refers to the use of an array of TokenType values. This array does not offer any more information than is already provided by the enumerated type and we did not have this issue within our code for iteration one. Implementing an array like this would force the programmer to access the tokens within this array using the enumerated type's value or an integer literal. Using the enum TokenType itself to access the content of this array is useless as the token type is already being accessed within the scanner class. Implementing this array may also lead to another bad practice of hard-coding an integer literal to access a token. The issue of using integer literals while referring to the token types in this project is further explained in reference to problem six. Therefore, the use of an array of TokenType values is redundant as the TokenType already provides a means of accessing the list of tokens and may even lead to more bad coding practices.

Problem four from lab 8 pertains to the ordering of the enum TokenType within the definition of scanner.h. We did not change the order of kVariableKwd and kEndKwd within our project for the initial iteration. The order of the token types does matter as our scanner is moving through the text provided and matching the tokens of FCAL language in order for us to further analyze the functionality of the FCAL program. Regular expressions are used by the scanner in order to identify these critical pieces of the inputted program. The order of these regular expressions depends upon the token's precedence. In the case of the kVariableKwd and kEndKwd, the scanner checks for the kEndKwd and later checks for kVariableKwd. This ensures that the scanner will not mistake the variable name "end_example" for a kEndKwd token. This example will match both the kEndKwd and kVariableKwd, but the kVariableKwd has higher precedence and should be the token type that is ultimately matched.

Problem 5 involves making named regex_t pointers for each individual regular expression rather than storing them in an array. Our code exhibited this problem in lines 82-132 in our original code, at the top of the scan function. Since the token enum exists, it is very simple to implement an array that indexes each regular expression with its corresponding token enum (since each enum's value is unique each will have a unique index into the array). The solution for this problem went right along with the solution for problem 2, where we created the array during initialization of the scanner object and the array uses the token enum as the index into the array that finds each pre-made regular expression. This is more efficient because less time is spent initializing unnecessary named regex_t pointers, yet the way the array of regular expressions is accessed is still intuitive.

Problem 6 involves using integer literals (magic numbers) instead of the token enum to identify a specific token. Our code did not exhibit this problem. It is important not to use magic numbers to identify a token in our program because any time the token enum is changed, any magic numbers used could potentially be wrong. If the enum had been used instead, the enum's value would be changed as well so the code would still function as it should. This solution goes along with the solution for problem 4, since it is a way to account for any changes made to the token enum type and preventing them from breaking the code. Additionally, using the enum to identify a token is more helpful for anyone else viewing the code, since they'll be able to immediately understand what the enum does given the context it is used in rather than trying to figure out what an unlabeled integer literal is supposed to mean.

The problems that we fixed mainly involved reducing redundancies in our code to increase its efficiency. Any code that was unnecessarily being run more than once has been modified to only run as much as is needed (such as moving makeRegex calls to places in the code where the only need to be done once), and any extraneous data structures or potentially inaccurate ways to access the data our program uses have been fixed (removing arrays of enums and any magic numbers used in place of enums). The changes have caused a slightly noticeable improvement in the runtime of our program’s tests; what used to take a moment to complete now happens seemingly instantly. The overall effect of these code improvements would be much more noticeable at a larger scale, but changes like these are good practice even for smaller projects.