Upon examining our code for iteration 1, we found that there are few improvements that could be done. Our initial implementation used global regex_t* declarations to define regular expressions similarly to the example in the problem 5 description.

Question 1:

No, make_regex is never called in our scanner_tests.h file as we tried to avoid code duplication. After we have updated our implementation and used array of type regex_t*, we created a function regex_initialize that is called from the default constructor of Scanner class. The purpose of regex_initialize function is to initialize regex_t* array to store regex_t* definitions so that there is no need to make redundant calls to make_regex. It is preferable to make scanner object a parameter to scan function. If scanner object is passed as a parameter then there is no need to instantiate a new scanner object in each call to scan functions. This approach is preferable because it allows to instantiate a scanner object once and simply pass it as a parameter to all the subsequent calls of scan function. This was not a problem in our implementation, however. In our scanner_tests.h file we instantiated scanner object once and then used it to call scan function. Since all the make_regex definitions are created in the object constructor, we could simply reuse them. Also to test the code properly we need to have consistent definitions of the regexes which means that they should only be declared once.

Question 2:

No, our scan function does not call make_regex each time when it's called. As discussed in the problem 1 solution, all calls to make_regex happen in the constructor of the scanner object. After the scanner object has been instantiated there is no need to redefine make_regex declaration as they can be used for all the subsequent calls to scan function. Each time the scan function is called, it needs to use precompiled regular expressions, but it doesn't mean that they have to be compiled each time the scan function is called.

Question 3:

No, we do not have redundant array of TokenType values. The data in such array would have been useless because it contains the same int values as the original kTokenEnumType type. All the values can be accessed directly from its definition, so that there is no need to initialize additional redundant arrays.

Question 4:

Changing order of in the definition of enum kTokenEnumType in scanner.h will not cause any adverse effects in the rest of our code. We made sure to use only the keywords from the enum definitions instead of int values, therefore the changes in order in the definition of enum kTokenEnumType do not have any adverse effects. Changes in the order only affect the int values that are bounded to the keywords.

Question 5:

In our original implementation we used global regex_t* declaration. This was not necessary. We got rid of those definitions and replaced it with array of type regex_t* which is initialized by the default constructor of the scan class. This approach allows us to save memory by avoiding global constants. Moreover, the regex definitions are initialized only when when we need them. As mentioned above, we put all the regex definitions in the array that is initialized by the default constructor of scanner object. The initial global declarations were defined starting line 18 in the scanner.cc file which are now commented out.

Question 6:

No, there are no places in our code in which integer literals are used instead of enumerated TokenType. We avoided using integer constants so that our code is not dependent on the order of declaration of keywords inside the kTokenEnumType. If TokenType values are used consistently in the code then its order does not affect the rest of the code. Hardcoding the int values, however, makes the rest of the code dependent on the order of declaration. This can cause errors if the new keywords were added to the kTokenEnumType.

Overall, our biggest improvement was moving the regex declarations to an array which is initialized by the public constructor of the scanner class. This is a valuable fix which can potentially save a lot of memory and also makes the regex declarations consistent with the rest of the code.