Introduction:
should briefly describe the general need in your code for some improvements and indicate what problems were identified in your solution

There are 2 major problems that could make the compiler to run slow and have negative effected the code. First, there was a few blocks of code for lexeme has been duplicated and created a large file. Second, the tokenEnumType list was using to call each lexeme. It could cause the code to crash if there was a switching order from this list.

Question 1:
Is there any duplication in the testing code you've written to test individual token regular expressions and the code used by scan?
Consider the example of your instructors initial design for testing as an example of testing code that could be improved. If he had made the scanner object a parameter to his testing code then the call to makeRegex that is used by the testing code but not by scan could have been removed.

The current implementation for testing the constructor for every single lexeme and that lexeme's corresponding regular expression does exhibit code duplication. Each block of code creates a token using a proper regular expression for a given lexeme, tests the token has the correct terminal and the regular expression does not throw an exception. Then, it creates a token using an improper expression and tests the regular expression to ensure it throws an exception when given an improper expression.

Currently, there are separate blocks of code for each lexeme which are duplicated for a total of 782 lines of code in the scanner_tests.h. This code duplication creates a larger file. It also makes debugging and modifying the code more difficult. For example, there are now 782 lines where a programmer can hide a run time error or a syntax error, and code needs to be modified for every lexeme to make a modification to the testing procedure described above.

A helper procedure will be created to solved this problem. This helper procedure abstracts the method for testing individual token regular expressions by creating a generalized procedure that can be called for each lexeme. This procedure takes less lines of code and can be modified in one location.

Question 2:
Does your scan function make calls to makeRegex every time it is called?
This is clearly inefficient. Where should this initialization work be done?

Previous to the completion of iteration1, this codes has been noted and moved to the seperate file, syntaxDefinitions.h. By careful inclusion of this header file, makeRegex expression is only called once for each regular expression.

The reasons why the makeRegex method calls were moved to syntaxDefinitions.h included: code efficiency, file size, ease of modification and ease of debugging. The code was more efficient beacause makeRegex expression is only called only once for each regular expression. Similar to question 1, removing duplicated code made the program shorter, easier to debug and easier to modify.

Question 3:
Do you have a redundant array of tokenType values?
The beginning of this might look like the following:

```
tokenEnumType emum_values[ ] = {
   //Keywords
```

nameKwd, platformKwd, initialKwd, stateKwd,
gotoKwd, whenKwd, performingKwd, exitKwd,

...

The data in this array is useless. Why is that?

The code base does not contain a redundant array of tokenType values.

The data in this array is useless because it is not necessary to create a mapping from the integer to an enumeration of the token types. This creates extra codes and it is more difficult to reason with integer values than with an enumerated token type.

Question 4:
Would changing the order of nameKwd and exitKwd in the definition of enum tokenEnumType in scanner.h have any adverse effect on the rest of your code? If so, what changes should be made to prevent this from happening?

Changing the order of nameKwd and exitKwd in the definition of enum tokenEnumType in scanner.h will cause the switch to associate the nameKwd case to the 'exit' lexeme, and to associate the exitKwd case to the 'name' lexeme. This would the program to throw an exception and fail to pass scanner_tests.

Seperating the ordering of the token type enum from the implementation of this scanner allowed to add or remove lexeme and change the order of the existing lexeme without negatively effecting the code.

To solve this problem, code for creating an array of TokenBuilder objects, which added to syntaxDefintion.h. TokenBuilder was a new type that hold all the common functionality in creating a new token.

Question 5:
Do you create a named regex_t pointer for each regex instead of only putting them in an array?
The definition below is an example of this.

```
regex_t *nameKwd_regex = makeRegex("^name");
```

Are these necessary? Can you get rid of these?

The code base does create a pointer for each regex instead of putting them in an array.

It is important because accessing an array is faster than accessing local variables, and the program can iterate through the array but it can not iterate through local variables.

To fix this problem, syntaxDefinition.h will include an array of TokenBuilder that hold the attribute of regex_t pointer. Changes made to the ordering of the enumeration will not negatively effect the implementation of the regex_t pointer array because the enumerated value for a lexeme is used as the index into the array instead of a concrete value.

Question 6:
Are there any places in which enumerated tokenType values should be used instead of integer literals? For example, do you ever use the integer constant 8 to

identify the integer keyword in some way? If so, can you get rid of these?

The code base does use integer literals when an enumerated tokeType is more appropriate. A switch with individual cases are used to correlate each tokenType to the implementation details of that token in both the token constructor and the scan function.

The same reasoning that applies to question 4 also applies to question 6. The same reasoning that applies to question 1 also applies to question 6. This is a large source of code duplication.

TokenBuilder is added in an array of objects, which how the implementation detail of both checking for a valid match and setting the property need to build that corresponding token.

Conclusion:
Must summarize the work you've done and discuss how valuable each fix is (or, equivalently, how problematic each concern was before it was fixed). Explain, overall how much, if any your code has improved.

The code has been much improve in term of efficiency and design pattern. Lexeme has been duplicated and created a large file. This problem was solved by created a helper function to generalized for each lexeme. Changing the order of nameKwd and exitKwd in the definition of enum tokenEnumType in scanner.h would have negative effect on the rest of the code. The TokenBuilder was added in an array of object to solve this problem. It checks for a valid match and sets property to build that token. Overall, the code looks more compact and easier to modify since changing the order of enum tokenEnumType list will not cause any problem.