Nostromo
Rohith Anil anilx005
John Carter carte668

Eliminating Redundancy and Improving Efficiency of the Scanner

This document contains descriptions of changes that we made to our project files for the scanner in order to improve efficiency and to also remove redundancies present in our code. The first change we made was to modify our scanner_tests.h file so that we only created a scanner and token object once for every test, rather than once per test. In that file we also condensed over 300 lines of code that tests our creation of regular expressions down to 20 lines by simply creating a loop for those tests. The next change that we made was to move our array of regular expression strings and the creation of those regular expressions into the constructor, so that they were created only once per scanner object creation, rather than every time the scan function was run. We also had a separate function to find the terminal types that used a switch statement to find the index of each TokenType but we were able to eliminate that by simply using the index of each token found during the scan. The last thing that we needed to change was to use an enumerated TokenType to check bounds in 2 loops instead of an integer constant. All of these changes helped improve the efficiency of our code, to eliminate some redundancies, and it also improved the readability of our code.

The first thing we wanted to verify is that we did not use the make_regex function in our scanner_tests.h file to test the proper creation of our regular expressions. We found that we do not use the make_regex function but rather we use a call to our scan function from a scanner object to test the proper creation of regular expressions. The reason why this is important is so that we don't make two calls to make_regex in case any of our test functions utilized our scanner object in other ways which would have resulted in duplicate calls to make_regex. One thing that we did need to change in our scanner_tests.h was the efficiency of our code when testing the proper creation of regular expressions and when testing the functions in our scanner_class.cc and token_class.cc files. There were some tests that were creating separate scanner objects and token objects(lines 40-113, scanner_tests.h) but we improved the overall efficiency by creating one scanner object and one token object to be used for all of the tests(lines 19-26, scanner_tests.h). We also improved the efficiency and readability of our code in scanner_tests.h by condensing over 300 lines of code(lines 141-462, scanner_tests.h) that tested all of our regular expressions down to 20 lines by creating a loop that tested each of our regular expressions(lines 144-168, scanner_tests.h), and a simple function that did the actual testing for each expression(lines 126-131, scanner_tests.h). We also added a token object to the argument of that function at line 126 so that it used only one instantiation of a token object rather than every time the function was called in the loop. These changes were needed to improve the efficiency of our code by needing only one instantiation of a scanner and token object for every test and also by condensing over 300 lines of code down to 20, which certainly improves efficiency and also readability.

The next change that we needed to make was to move the creation of our regular expressions from the scan function(lines 41-65, scanner_class.cc) in scanner_class.cc to the constructor. This was important because our regular expressions were being created by make_regex every time the scan function was called rather than a single time per creation of a scanner object. We first

moved our array of regular expression strings to the constructor(lines 32-43, scanner_class.cc). Then we moved the for loop that created the regular expressions along with the creations of the white-space, comments, and line-comments regular expressions to the constructor(lines 45-54, scanner_class.cc). These changes were important not only to eliminate the redundancy of continued calls to make_regex but to also improve the execution time of our program.

The third problem with our program that we needed to resolve was to eliminate our function find_terminal(lines 137-273, scanner_class.cc) which used a switch statement to find the enumerated TokenType that corresponded to the token that was found when scanning the text. This function's purpose was to set the terminal in our token object to an actual enumerated type, but we found that it was actually useless because we could just use the original index that corresponded to the token found in our regular expression array. We then just typecast the index that was found to TokenType, and that was sufficient. This was an important change because we didn't need the extra call to our function nor the extra space to find our TokenType value which improved execution time and readability.

The fourth potential problem that we needed to check for was that the order of the enumerated TokenTypes was not important to the overall function of our program. We verified quickly that it was not a problem because our scan function in scanner_class.cc always used the token found in the text that had the maximum number of characters matched. This was important to verify so that the proper token was found when there were potential conflicts with other possible token values. For example, it will match the token for the "end" keyword rather than treating it as a variable keyword. Since our scan function checks for the maximum number of characters matched when scanning the text, changing the positions of any of our enumerated TokenTypes would not have an adverse effect on the function of our program.

The fifth potential problem that we needed to check for was that we didn't create a separate regex_t pointer for each token regular expression. We verified that this was not a problem since we used a regex_t pointer array to store all of our regular expressions for our tokens. The reason that this was important was simply for efficiency with regards to space, the ability to check for regular expression matches by looping through our array rather than having to check each regular expression individually, and eliminating the need for more than one regex_t pointer.

The last problem that we needed to fix was to eliminate the use of integer constants in our for and while loops(lines 66, 93, scanner_class.cc). To check for the bounds of both loops we changed both integer constants to the kEndOfFile enumerated TokenType value since that value would always be found immediately after any regular token values. This was important to change so that our program would properly scale if we were to add any new TokenTypes. If we were to add any new TokenTypes, the kEndOfFile TokenType would always remain immediately after all of our regular token values, including any new additions, thus eliminating the problem of having to change any loop bounds.

The six potential problem areas that were outlined were important to address because they had a large impact on the overall efficiency and readability of our code. The four major changes that we made have improved the efficiency of our scanner program by eliminating excess code and eliminating redundancies. A major impact was reducing the execution time of our program and

also improving the readability of our program. Creating one scanner and token object for all of our tests certainly helped, along with putting a large number of our tests into a single loop, rather than numerous lines of unneeded code for each individual test. Moving where we created our regular expressions from the scan function to the constructor of scanner_class.cc also improved the execution time by not having to create the regular expressions per scan but only per scanner object created. The execution time and readability of our program was also improved when we eliminated the function to find the TokenType corresponding to each token found and instead simply used the index of each token found from our regular expression array to set the token terminal. The final change we made was to utilize an enumerated TokenType for our for and while loop bounds rather than an integer constant. Although this had little effect on execution it helped scalability by allowing potential additions and subtractions of TokenTypes without the need to change other parts of our code. Because of the changes that we have made, and also by verifying that other problem areas were in good order, we can have much more confidence in the efficiency and readability of our code.