Project 9

Lexical Analysis of Bantam Java Programs

Kyle Douglas, Paige Hanssen, Wyett MacDonald, Tia Zhang

Introduction

In this week's project our group's main task was to build a scanner that breaks down Bantam Java files into tokens. In order to make our scanner work properly, it was important to account for a number of specific token types, errors, and other special cases. Ultimately we were able to integrate this lexical analysis feature into our IDE.

Procedure

Prior to adding our new scanner in our IDE, we first started this week's project by drafting up our general approach. This meant looking through the given files Scanner.java, Token.java, Error.java, ErrorHandler.java, and CompilationException.java and thinking about what modifications needed to be made in order to make our lexical analysis tool work seamlessly. Almost all of these files were left untouched, and the only one that we made additions to was the scanner file. After reading through some of the Bantam Java Lab Manual, better familiarizing ourselves with bantam java and its legal tokens, we started to make cases to handle specific token types.

In trying to account for all possible cases for tokens, there were a few unforeseen problems that we ran into. One problem we had was figuring out how to deal with the end of a file(EOF) when there was an open string or block comment. Unsure as to whether or not this should return an EOF token, an error token, or both, we ultimately chose the latter. We did this by having a conditional checking if stringOpen = true or multilineCommentOpen = true or singlelineCommentOpen = true after the while loop. If so we would have it print out an error because we know a string or comment is still open. We believe this was the most elegant solution because having an open string or block comment at the end of a file IS an error, and despite the scanner not realizing it until the end of file (when you would expect an end of file token), the error should be recognized first, followed by the EOF.

When we wanted to actually integrate our scanner into our IDE from project 7, we had to make some adjustments to our old project in order to make it more elegant. This meant freeing up our toolbar by deleting our compile, run, and halt buttons so that we could replace it with our new scan button. If a saved file is selected to scan, then a new tab will open and print the tokens in the tab, and all the errors are printed in the console. Tokens were printed in the new tab by addings tokens returned by the scan() method to the Tab's CodeArea. We wanted to make sure scan() returned each token, rather than having another field to store the Tokens. We ran into an issue because we had a while loop in scan(), as well as handleScanButton() in ToolBarController, which was skipping every other token. This took a while to debug, and we owe credit to Liwei for catching the double scan().

When refactoring project 7, we deleted our previously existing CompilationController, and instead made a ToolbarController that handles the Scan button. Because that is where the Scan button is handled, we chose to create our ErrorHandler and Scanner in the ToolbarController class, to avoid passing it through from Controller. We also moved the handling of Find into ToolbarController instead of EditMenuController, because the search bar is displayed in the toolbar, so it just makes more sense.

Inelegance

In the refactoring of our code from our previous projects, we had to change the computeHighlighting method in the JavaCodeArea from private to public, which, although inelegant, allowed us to access it within the editMenuController. We chose to do this because it was necessary to use it when resetting the highlighting of text after using the find functionality. In addition to this inelegancy, we have numerous case statements for our scanner, which could perhaps be more concise and/or split up into more methods. We also had to pass in the tabPane and console to our ToolBarController, which is not elegant but was necessary in order to print the Tokens to a new tab and the errors to the console.

Work Division

Paige refactored and debugged the previous project, particularly working on completely fixing the Find function, and worked with Errors, did documentation and completed other, small tasks. Tia’s primary work was writing the pseudocode for Scanner.scan(), translating it to code, and doing most of the the debugging for it. Wyett and Paige worked together to figure out the case we hadn’t thought of in the last project that caused an error. Wyett also tested for bugs and fixed issues like EOF with unclosed string/comments, along with writing the main method in Scanner. Kyle wrote the report.