Implementation

The development of the plagiarism detector involved implementing Michael Wise’s “Greedy String Tiling” algorithm, as introduced before. In order for the algorithm work flawlessly, we have prepared our own data holder classes: a “Token” class responsible for objectifying each tokens in investigation, and a “Match” class that tracks where the duplicates were detected. The implementation of the pseudo-code of Greedy String Tiling was done in Java. Our tool’s workflow is as follows.

## Tokenizing

The Greedy String Tiling compares tokens instead of each characters. It is required that the tool is able to understand the source code (written in Java) and recognise the grammar. We used ANTLR and an external Java grammar (from Github, under BSD licence) for ANTLR to convert the source files into token strings. Initially, the given Java grammar dealt with all Java statements in one expression. However, in order to be able to classify different semantics behind the tokens, we needed to add separate method for if statements, for loops, etc. A Listener object was created for the parser to be able to tokenize according to the aforementioned rules. The tokenized string is then passed to the main method to be manipulated for plagiarism detection.

## Algorithm

The Java implementation of the Greedy String Tiling algorithm is as follows:

* Given the list of Token objects of two source code files, set A and set B, we brute-force iterate through A and B with two nested loops. The algorithm compares the tokens to see if they are identical, and to see if the two tokens are not ‘marked’ (initially set to false). If the condition succeeds, another innermost nested loop repeats this process until the condition does not hold. While it does, it counts the how far the duplicate has extended, and creates a Match object with Index of first character of duplicate from both sets, and the length of the duplicate - thus it eventually collects the set of all longest common substring.
* The algorithm has a mechanism where it keeps only the largest duplicates. The program checks whether, in each loop, the match is at least as long as the longest ones found before, whose length is stored in maxMatch variable. If the length is equal to maxMatch, another maximal match is found and added to the global set called matches. If the length is larger than maxMatch, the global set matches is cleared and newly adds the current duplicate. The maxMatch is now the length of the current duplicate.
* The global set ‘matches’ is then emptied to the result set called ‘tiles’. The matches set is reset and the above process repeats until maxMatch is lower than a fixed threshold, which we set to be 3. As the more duplicates are found, the algorithm will eventually halt when it cannot find any more duplicates that are long than length 3.

## Analysis

The similarity percentage is calculated as follows:

Coverage = sum of all lengths of duplicated codes

Similarity percentage = 2 \* Coverage/length of file A + length of file B

Higher percentages indicate more chance of plagiarism. It should be noted that given two identical code, if one was to add more contents to one of them, the similarity percentage decreases, due to the nature of the calculation.