William Cao (pastor13), Jared Le (jaredtle)
CSE 403 - Project 1
2018-03-28
Intelligent Code Merge using Abstract Trees

In a multi-developer environment, version control software becomes extremely important to the productivity of the group. However, popular packages, such as Git, use a line-based approach when detecting changes/resolving conflicts. Conflicts are identified and developers notified based solely on whether different changes were made to the same lines, regardless of the content of those changes. Thus, some of these merges conflicts are not actually “true” conflicts. For instance, one developer may have simply indented a line of code while another renamed a variable on the same line. These changes are capable of naturally coexisting, and may be applied together. These merge conflicts can not only be time-consuming and tedious, but also very expensive, as development time must be set aside to allow the developers to identify and resolve these trivial conflicts manually.

Those who would benefit the most by this tool would be those who work in a multi-developer environment. This tool improves software quality by automating the merges that are trivial, which allows the developers to dedicate that saved time to improving other aspects of their code, such as bug fixes.

The high-level approach for this utilizes abstract syntax trees (ASTs). The software would parse the base code into an abstract syntax tree based on the rules of the language itself, and do the same for the incoming revision code. Afterwards, these two trees will be compared for conflicts. If there are no outstanding conflicts between these two trees, they can safely be merged together. This is different from the line-based approach, which only compares changes between each break line, instead of determining changes using the structure of the code.

By implementing this method, the version control software can automate a significant number of merge conflicts, which in turn helps developers focus on the conflicts that are significant to the code base. This project should be able to deal with refactoring of code (including moves and renaming), insertions, and the deletion of code. It should be able to deal with the many combinations of the listed changes, such as an insertion and rearrangement within the same block of code. One limitation to this approach is that it relies heavily on the syntax of the code. For instance, a commit that contains invalid syntax would be extremely difficult for the tool to parse, as the underlying AST would be unable to parse the code with any reliability. Additionally, comments and multiline expressions may lose their formatting unless the project takes care to preserve that formatting across moves and other changes to the code. Furthermore, any code that cannot be parsed by the parser will be wrapped in its own type of object.

The single greatest challenge in developing the product on schedule is figuring out how to parse the code into the abstract syntax tree. Not all languages have an easily understood

BNF table, and complex languages will require a large amount of translation in order to properly express the language. Additionally, complex language features, such as the anonymous functions introduced in Java 8, will increase the complexity of the abstract syntax tree. Parsing code into the abstract syntax tree may also prove to be challenging due to the myriad of styles that developers code in. Ideally, the developers would interpret this as a sign that perhaps they should be following a more rigid style guide. Because this is unlikely, the project can take several measures to ensure correct translation. Static analysis might be performed to ensure that the code is being translated properly, and previous academic work on encoding and decoding BNF text into trees can be consulted.