Verbal Description

Bang! is a dynamic, weakly-typed, expression-based functional scripting language that prioritizes concise syntax and is a good language if you want to practice code golfing. The entire codebase will be written in TypeScript, and requires no external libraries or frameworks. This language is loosely inspired by JavaScript's dynamic typing system, but uses "nil" instead of "void", "undefined", and "null". *Bang!* also doesn't have null reference exceptions, which is one of the more frequently seen errors in JavaScript. *Bang!* also doesn't have type errors, another commonly seen error in JavaScript. Because there are no type errors, any operator used between any two types has special functionality and will not cause errors or non-deterministic behavior.

The langue also takes inspiration from Python, another language commonly used for code golfing. However, Python writes out a lot of operators (such as "and", "not", and "or"), which takes up more characters than *Bang!*'s operators. Another important feature of this language is its many non-associative operators, like Python's greater-than (>) operator. For example, "3 > 2 > 1" in Python would evaluate to true, whereas the left-associative > operator from JavaScript would cause the same expression to evaluate to false. Developers who want to run scripts written in *Bang!* will only need a computer with a command line program and some kind of text editor to write their script in. Access to the language's documentation would also be useful.

Justification

Bang! is an appropriate project for CMSI 4072 because it applies what I've learned in both CMSI 3801 (PLang) and CMSI 3802 (Compilers). It's an extension of the original version of *Bang!* from Compilers, but with the implementation of features I couldn't figure out how to implement (due to either lack of knowledge or lack of time). Additionally, even though this is a continuing project, it's still a more difficult project than the first rudimentary compiler I implemented. Last year, because a compiler typically follows the waterfall model and I hadn't designed or implemented a compiler before, I didn't fully understand how each component would affect the later components. Some of the things I wanted to implement weren't possible either due to the chosen framework for the grammar (OhmJS) or because I didn't set up the first components correctly, and changing early components required major refactorization across the entire codebase. However, this year, I have a better understanding of what will be required for the language features I want to implement.

I designed most of the language features during junior year in CMSI 3802, and I should be able to complete the project by the end of the year. Because this is a project I've been working on for the past semester, I already have the front end of the project finished, and will only need to complete the part of the interpreter that actually calculates and produces the output. If I complete that early, I can expand on the project by implementing an optimizer, which will shorten runtime of user programs. Another potential expansion is a website dedicated to code golfing in *Bang!*. CMSI 3802 taught me the skills I need to complete this project, and having done a more rudimentary, compiled version of this language also gave me the context I'll need to create a solid foundation for this project.