The overall design of my optimizer was inspired by Lecture 4 Slide 4. There are 6 main steps that my optimizer implements: creating a Control-Flow Graph (CFG) for each of the functions in the program, calculating the GEN/KILL sets for each node/block within the CFG, calculating the IN/OUT sets for each node/block within the CFG with the "fixed-point" condition, running the "mark" part of the "mark-sweep" algorithm, then running the "sweep" part of the algorithm, and then finally reconstructing the function to only keep those which have been marked (to keep). I chose this approach because it seemed the most straightforward; having covered it in class, and worked with it, and being provided with the direct algorithm, it seemed to make the most sense for the project.

As for design decisions that I made, one of the most important choices was to make each instruction its own node/block. This seemed the most intuitive because if I wanted to remove an instruction, I could just remove the node itself from the "graph" rather than dealing with large blocks of code and removing an instruction from within. In addition, I decided to create two extra classes, IRcfg and IRNode. IRNode contained all the information necessary for a node/block as well as some helper attributes such as "defined_var" and "used_vars" which I used to assist in other parts of the optimization. IRcfg creates the CFG itself, and the rest is pretty specific detail which I will leave to the code and comments rather than reexplaining everything unnecessarily here. However, note that I used HashMaps in IRcfg constructor. This is because it makes it easier to find nodes, especially when connecting them. It is important to note that for these HashMaps, I decided to use the "irLineNumber" of each Instruction as a unique identifier for the nodes.

Some challenges that I came across included figuring out what information was important to me, figuring out how to retrieve that information, figuring out what information was important

to have (i.e. in the creation of IRNode and IRcfg classes). To address these challenges, I diagrammed the relationship between each class, for example, for IRProgram (which is at the top of Demo.java main function), it contains a list of IRFunctions, which amongst many contains a list of IRInstructions, and so on and so forth. I also diagrammed out the CFG by hand to understand the desired flow of the example program itself to check if my CFG construction was on the right path. But the most helpful of all was print statements. I added many after I completed each part of the optimizer, which helped me debug, for example, missing instructions that should have been marked as “critical” but weren't or instructions that were removed that shouldn’t have been, amongst many other things I checked with these print statements. Overall though, I felt that this project was more or less straight forward so I personally didn’t come across too many unforeseen or nonsensical challenges/issues. As for known bugs or deficiencies, I cannot spot any that exist at the time of submission.

To build the project, simply navigate to the top-level project directory and run the build script as so: **./build.sh**. to run the optimizer on some input IR file, execute the run.sh script as follows: **./run.sh <path/to/input.ir> <path/to/output_file_created.ir>**. To clarify, the provided argument is the IR program that the optimizer will execute on. As for the public test cases, when testing locally, all test cases passed with self-generated .out files matching the ones provided for each respective input.