On a high level, we have our optimizer broken up into three basic parts. The first part takes in the IR file and builds out a CFG graph. This algorithm looks at the instruction and figures out whether it is the first instruction in a basic block, an unconditional jump, or a conditional jump. Based on these results, the algorithm will declare a vertex to wherever it could jump, add a vertex to another leader, or append the instruction to the current basic block. The second part is analyzing the reaching definitions for the CFG through an iterative algorithm. In this algorithm, each block keeps track of its own definitions, and the lines of code that could clobber their own definitions. We used these to track the definitions that were making it in and out of each block in the CFG. The final part of our optimizer is the part that actually modifies the IR based on the data that was gathered in the previous section by marking the critical instructions for the operation of the program and eliminating anything unnecessary.

On a low level, we decided to use java for the helper functions provided with the project to save us the time of porting all of the objects and methods provided. We had a heavy reliance on objects to track the different values needed throughout the analysis phase. We used a two-way circular tree type structure in order to save the vertices between the different basic blocks in the CFG. This also assisted in being able to run all of our processes on a CFG object and continually build more information within the objects to create a CFG that has all of the information needed for optimization within itself.

One of the challenges for the optimizer is how much data needs to be related to other bits of data in order to properly analyze. Along with this, trying to limit the number of times we have to traverse the generated CFG for data that could have already been collected. Also, making sure to account for edge cases with recursion and local variables proved to be something that was easily overlooked during the initial planning phase of structuring the optimizer. As a result of this, some of our solutions were more brute forced based on certain circumstances instead of a general rule that worked. This would also count for a deficiency that could be looked at further in order to keep processes more concise. Also, there appears to be a bug when an instruction is defining itself with itself, and we had too many instructions for the quick sort.

To build: find src -name "*.java" > sources.txt && javac -d build @sources.txt

To Optimize IR file: java -cp ./build optimization/optimizer path/to/file.ir

This optimization function will deliver the optimized IR in a top level directory called “optimized”. This directory will mimic whichever relative path/to/file your IR was originally. We ask that you prepare that by making the directories within this file before running the optimizer.

SQRT TEST CASES	0	1	2	3	4	5
Non Optimized	94	70	16	101	133	155

Optimized	77	57	11	89	110	139