CS401 Optimization Week 2 report .
I spent this week reading chapter 9 of the "Dragon" Compilers book., as well as the Monadic Intermediary Language Documentation. I found the coverage of the data flow analysis from this much easier to follow than in Mogensen's book. I was able to take the notion of the (semi)-lattice from the book, and refine my existing code to better reflect the lattice structure, I noted that in the book the top node represents that there is no data (undefined) and that the bottom represents that the argument is not a constant, which is the inverse of my understanding of how we had discussed the structure when you had first introduced me to it. In the Monadic Intermediary Language documentation, I noted that it states that when a variable is set by a Bind "there is no way to change or otherwise assign a new value for an identier v that has been introduced in this way." which initially did not make sense, but once I realized that the assignments such as i $\rightarrow$ i+1, only occur when i is introduced as a block formal.

I was able to write a working version of the constant propagation algorithm, building off of the code that I wrote last week. The new code takes the array of information about a blocks formals which is a [formals.length][maximum tuple size] array that either is an array of constant values that are args to the block, an Atom.UNDEF if there are no known constants used as arguments, or an Atom.NAC if there are more than the maximum tuple size possible constants, or if the block modifies that argument. With each constant that is known, a new block is created that has an Arg $\rightarrow$ Constant inserted at the beginning, and the rest of the code of the original block. Each caller to the current block that has the same constant argument is replaced with a call to the new block. This implementation had the effect of completely unrolling both loops in arrs.mini, but it turned out to be too aggressive either on its own, or in combination with the local optimizations because for the case of sample.mini, there is no stopping point and the amount of code grows until the a stack overflow occurs.
I made additional attempts to control the amount of growth, by keeping count of how many times each block has been specialized and stopping the generation of new blocks, but this attempt has thus far failed, the exact cause is not clear, but I believe it is because of the prefix inlining combined with the way that I was counting the specializations.

For the next week I am caught between considering my current work a finished experiment that failed, and starting fresh with the experience this has given, as well as the additional knowledge from the readings, alternatively it may be worthwhile to continue working to correct the incorrect behavior of this implementation. I would like to hear your thoughts on either approach, and whether it would be better to have a more abstract framework that applies the IN and OUT definition calculations and can be used for both constant propagation and common subexpression elimination.