Phase 2 Project Report

make all << compiles program, and writes to files
make output << opens output file
make error << opens error file
make clean << removes executable and made files
make memCheck << valgrinds the program

In order to compile the correct .mis file, you must name it "Program.mis"
This does not run any client and server models

Design

TCP or UDP?
We chose to implement TCP because it gave us a lot of safety and ease of use that UDP would not be able to guarantee us. As we developed the client server model we understood that users could create MIS programs that are bigger than the max UDP packet size. So in order to correctly use UDP we would have to implement our own packet ordering functions, packet loss checks, and flow control, which is basically TCP. It would have been much harder to implement UDP for these reasons.

The VAR class: This family of classes changed the most from phase 1. In phase one we over complicated things with the way we organized data members, member functions, and over use of templates. In phase two we simplified the template use to be restricted to only the parent VAR class, which had a template data member to match the desired types of the variables. We then split up the previous template class, into three subclasses specific subclasses of VAR; NUMERIC, REAL, and CHAR. We also added new member functions to aid in the ease of setting, and getting values, as well as locking variables for multi-threading.

As a result of these changes, small corrections were made to all the arithmetic functions. Since we now have separate maps for each variable type, a search over all these maps must be conducted in order to determine if the variable exists and use/change its value. This is the same procedure we use anytime we have to access a variable. In hindsight, this is a very redundant task that could be encapsulated in a function.

Multi-threading
Our multi-threading implementation involved running a slightly altered version of our main run process for each thread. While parsing the file on the initial read, we identified all line pairs

which indicated the beginning and end of a thread. Then, when our main program encounters a THREAD_BEGIN command, it instantiates a thread, which runs its own "mini-main" and jumps the main to the line immediately following THREAD_END. For locking variables, we implemented a design that enabled variable lock acquisition and release within a thread. Although this choice is time-intensive, we decided to go this route since many of our previous design decisions (especially maps) are memory-intensive. The aquireLock function performs a busy wait which guarantees atomic locking of the variable in the thread. Our variable classes were designed with a private boolean data member indicating the lock status of the variable. Functions to view and set this data member are available publically.

Multithreaded Server/Client:

The multi-threading is the very hard part for our group. In the beginning, we were trying to imply a one-to-one UDP client/Server first. It works at the beginning. We can successfully transfer a txt file. However, we realize that if there always can be a file that is larger than the buffer size. There will be an error. So we switch to TCP. We successfully created a TCP client/server that can transfer a file. We were putting our mis main inside of a void function in the server to make it easier to call. Furthermore, we were sending line by line in the client. After that, client will receive line by line and save those in the map according to their line number in the map in the server. However, we were unable to complete an implementation for the server to handle multiple clients at a time. Also we manipulate with private/protected the members of TCP Socket into public so we could find our errors of codes easier.

The design of JUMP/LABEL classes:

Comparing to phase1, the design of jumping changes a little while label does not have any changes. Checking if the jumping conditions are met are inside of the jumping classes now in phase2. In phase1, the checking was in the helper functions instead of the class bodies, which makes the helper functions less condense and easier to be read. We were not able to implement jumping capability from inside to outside threads or the reverse. Jump entirely within main or entirely within a thread worked for Phase 1. For Phase 2, we completed the design of the jump classes which included the new updates to the variable classes. However, we ran out of time and were unable to test these revised classes within our new framework so we have excluded this capability from Phase 2.

Problems with the program

Throughout the course of this project, every step brought new insight into the design flaws of previous portions. Since there was no time between phases to redesign our MIS engine, we had to work with the existing code, even after recognizing flaws in our design, in order to keep on track to implement as much of the assigned functionality as possible. In retrospect, we would

like to have implemented a standalone parse method and an actual MIS engine object (instances of which could be created for each thread).