Phase 1 Report:

HOW TO COMPILE AND RUN

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”

Design

Main: We chose to create maps for our variables and calls to function helper classes. As main reads in the opp code (e.g. VAR, ADD,etc.), it accesses a map of string keys and function pointer values. There function pointers point to our helper functions, one of which exists for each type of operation. It is these helper functions which parse the string appropriately, create the appropriate operation object, have the object run its process function, and finally delete the operation object. We could improve on this design by having a static parse function which lives in main and parses the stringstream into an appropriate vector of string parameters and an opp code. This way each helper function would not be parsing and we would reduce redundancy.

The Var Class: This class hierarchy has a base class that is designed to have two derived classes. These classes are split into a STRING and templateVar class. As the names suggest the STRING class is used to create a VAR of type string, while the templateVar class is a template class designed to create either the NUMERIC, REAL, or CHAR variable types based off of the key words.

I must say that this design does not have many bright sides. The design was hurried, and created with little confidence in c++ coding ability. There are a lot of hurdles to go over for simple data retrieval and assignment. One of them being that the ASSIGN opcode, just does not work. With this in mind, the design for phase two for the VAR class will change drastically a better quality of life when working with the class.

OPP Codes: For the operations implementation we chose a class hierarchy where the base class was Function with only virtual methods. Arithmetic classes (Add, Mult, Sub, and Div) inherit from Math which inherits from Function base class. The Math class differs from Function in that it has an additional function convert() which takes in a string and converts to the appropriate parameter. All arithmetic classes have a string vector filled with parameters, a result string, a validator() function which checks parameter types and the number of parameters, and a process() function which conducts the appropriate mathematical operation and sets the value of the first variable to the result of this operation. In retrospect, it would have been better to have Add and Mult inherit from a class where the validator() implementation would be, since these two arithmetic operations have the same parameter restrictions. This is not true for Sub and Div, since the Div validator() has to check the additional condition that the last parameter cannot be zero (as this would cause us to divide by zero.)

Label Classes: The JMP and Label classes are individual from other classes. First, we scan through the whole file to just for creating labels. The helper functions will in charge of initializing labels, which will take a stringstream as input, parse it, and construct a label class. First, it will check if there are any duplicate labels. If there are not any, label classes are successfully constructed. Each label class contains a string as its name, and a int as its line number. Also, name and line number will be stored in a map for later uses.

JMP Classes: After we created the label classes. The JMP classes will be seen as a normal execution. There will be a global variable called counter. Each time a operation is executed, the counter will plus one. “JMP” will call out the helper function to parse the stringstream, and construct a JMP class. First to check if the label exists. Then, base on the classes to check if the variables are exist. If it exists, then to check if that’s the numeric. Finally, the classes will check the conditions, which are Z/NZ/LT/GT/GTE/LTE. If the conditions are met, the global counter will change to labels line number, and changes the execution process. If the conditions are not met, then the global counter will just plus one, and the process of execution goes on.