Henry Chang, hechang@ucsc.edu
Andrew De Neve, adeneve@ucsc.edu
Aaron Doubek-Kraft, adoubekk@ucsc.edu
CMPS 109
11/08/2016

CMPS 109 MIS Phase 1 Report

Makefiles and building project:

- Running "make" in the root directory will build an executable of the project. The executable is invoked with "MIS_x <filename>"

Workflow:

- Initial UML design, collaborative
- Implementation:
    - Henry: Types, Out
    - Andrew: Arithmetic Ops, Get_Str_Char, UML Redesign
    - Aaron: Parser, Jumps, Report
    - Collaborative: Main, MIS
- Redesign and reimplementation using the object factory model

Design Decisions:

- Overview: The base of the MIS system is the MIS class, which contains a Parser object that is instantiated on the file passed in with the command line argument, a map of variables, a factory for instantiating operation and variable objects, and a run() method that executes lines from the passed in file one at a time. All instructions that are processed by the MIS inherit from the Keyword base class, which enforces the implementation of clone, initialize, and execute methods. The MIS receives a line from the parser in the form of a string vector, and then uses the first element in the array to clone an appropriate object, which is then initialized using the remaining elements in the vector as arguments. The MIS then calls execute on the object, which results in variable objects being stored into the map, or operation objects executing their operation and then being deleted. Any exceptions are written to an "MIS.err" file.
    - Alternative: Our original design would have involved using a switch statement in the MIS class to determine which operations to perform, so many of our objects have parameterized constructors that are not used in the final design.
- Keywords: All keywords that can be executed by the MIS must inherit from this base class that enforces clone, initialize, and execute methods by using a pure virtual method. Clone and initialize have arguments of a string vector representing the raw data to be used to instantiate the object, a reference to the variable map in the main MIS class, and a pointer to the current parser object. Variable objects and insert themselves into the variable map, and other operation objects can access the variables by name in the main program and call methods in the parser to change the position in the input stream.
    - Alternative: In our original design, this class did not exist. It was added to support the object factory initialization model in the MIS system, to improve on the switch statement

- Parser: The Parser simply goes through the file line by line, assuming correct syntax, and returns a vector of strings representing the line. The type of the operation is then interpreted by the MIS system, and the arguments are interpreted by the individual class initialize methods. The Parser also maintains a vector of labels, which mark specific locations in the input stream, and those labels can be called by name from the various Jump objects to move the Parser to that location in the input stream.
    - Alternative: the Parser could have created a larger data structure, like a 2D array, which contained all lines in the program. This could have allowed for more initial processing of the file, like checking for syntax, or evaluating certain types of statements like labels and jump before others, but also would have introduced additional complexity.
- Types: The Type class is an abstract base class for any variable objects used by the MIS. It enforces getValue and setValue methods that take void pointers, which are then casted internally to the correct type of pointer, and the value of the object is written to the variable pointed to by that pointer.
    - Alternative: This design introduces some undesirable type checking for the caller, but this is the best solution we could find to preserve the inheritance structure. We also tried using templates, but this would have negated some of the benefits of inheritance that we utilize elsewhere in the program.
- Jumps: The Jump classes are divided into a hierarchy with the JumpOperation class at the root. This class defines a final method execute() and enforces the implementation of a method jumpCondition() with a boolean return type. If the jump condition is true, then the jump object goes to the label with the given name of the parser passed in as an argument. This structure enforces that any new jump objects will actually execute a jump, and the available customization is in what logic will trigger the jump. The hierarchy is further divided by the number of arguments that are required to initialize the jump object.
    - Alternative: The hierarchy was expanded from our original design to accommodate cloning. We initially only had three jump classes, which had a boolean flag in the constructor to distinguish between Z and NZ, and an enumeration to distinguish the various comparison jumps.
- Arithmetic Operations: each arithmetic operation class implements an execute method that performs the appropriate calculation and stores the result into the variable passed in as an argument. The Assign and Get_Str_Char operations also inherit from this class, because the internal implementations of clone and execute are extremely similar to that of the other arithmetic operations. The actual operation of these classes did not change significantly during the redesign, however the initialize operations are somewhat complex due to the necessity of working with variables of different types.
- Other Operations: Label also inherit from Keyword. Label creates a label at the current position in the input stream and then inserts it into the Parser label vector.