Assignment #2

Interpreter

Austin Sy-Velasco

Student ID: 915863837

CSC 413-01 Summer

https://github.com/csc413-01-su18/csc413-p2-iamaustinsy.git

Introduction:

a. Project Overview

In this project, the main objective was to create an Interpreter, a program that would take a Machine level code (in this case Code X), run the code in a Virtual Machine of the compiler, and output the desired task back to the user. In one case, the code is of factorial, which means it takes a value from the user and outputs the factorial of that value back to the user.

b. Technical Overview

For this project, the main class that starts off the build is the Interpreter.java class, which is already done by the professor. Alongside this class, CodeTable.java is provided as well which is a HashMap of all the Byte Codes that we will see in the Code X source code. What I have had to implement for this project are all the Byte Codes which are an extension of the abstract super class ByteCode.java. On top of that I have implemented the ByteCodeLoader.java class that reads the source file for the various byte codes and adds them onto an Array List called "Program." The next class is the RunTimeStack.java where most of the complicated parts of the project occurred for me. To my knowledge, the code is read through and values are pushed onto the stack for future computations. And finally the last class is the VirtualMachine.java class that executes the program to provide the output back to the user.

c. Summary of Work Completed

At the start of this project, I was extremely lost as to what to do, and I can probably say this was my toughest assignment so far. My work was based off of the recommended order of work to do provided in the instructions. I was able to create all 15 ByteCode class being an extension of an abstract ByteCode.java class. Then I implemented the loadCodes() method in the ByteCodeLoader.java class. I will get back to this class towards the end. The next part I had to implement was the resolveAddress() function in the Program.java class. This I will also come back to later. RunTimeStack was pretty straight forward to implement as the functions needed are described in detail the pdf. The VirtualMachine's executeProgram() was also provided by the pdf so this class was also simple to implement I just put some getters/setters that the byte codes would need. Due to lack of knowledge and time, I failed to implement them properly and the code runs up to a point where it asks the user for a value to work with, and then it becomes stuck in an infinite loop constantly asking for a value and not outputting what I would like it to.

Development Environment

a. Version of Java Used

JDK 1.8.0_73

b. IDE Used NetBeans IDE 8.1

How to build or import in IDE

Begin by cloning the repository onto your local computer through Git. Assuming you are using NetBeans IDE, start by going to File and selecting New Project (CTRL + Shift + N). Once at the pop-up screen select Java as your "Categories" and under "Projects" select Java Project with Existing Sources. Name your Project under Project Name and select where you want to build this project in your directory. In the next step, under "Source Package Folders:" click Add Folder and go to the directory of the repo that you cloned earlier, go into the folder and select the source folder of the program. After that go to the next step and finish importing. Then to setup the machine code that you would like to run hit File > Project Properties > Run. Under main class select "interpreter.Interpreter" and under arguments put the entire directory of either "factorial.x.cod" or "fib.x.cod."

How to run your project

Once the program is imported you can go to any file and build the project to run it. Start by building the project. Use “Clean and Build Project” or Shift + F11 to build. Then run the program with “Run Project” or F6. You should see that the interpreter asks for a value from the user, and then proceeds to store it (depending on if dump is on, which as of right now it should be commented out) but then continues to ask for a value when it SHOULD be giving back a proper answer.

Assumptions made when designing and implementing your project

We are assuming that the source code provided by the professor is working properly and any problems that the program may have is due in part to the work done by myself.

Implementation Discussion

The byte codes are done based on the code table provided and the thorough explanation of their functions in the instructions. Each byte code should have an initialization function and argument function. The initialization function creates the byte code and is then pushed onto the ArrayList program. The argument function works in the cases such as “LIT 3” where the 3 is the argument. This case would attach the argument value to the bytecode to be put on the runtimestack when being executed.

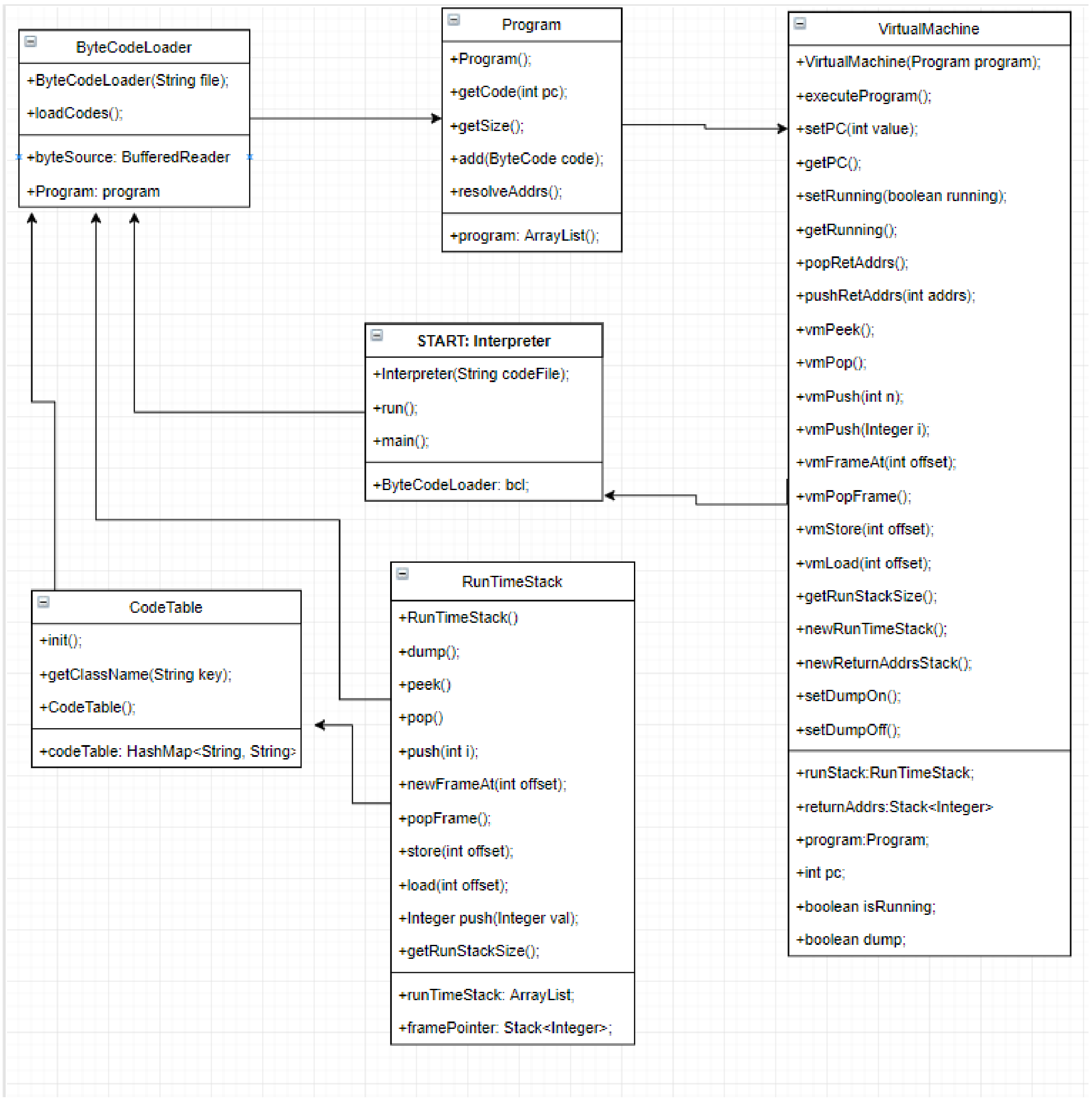
The class ByteCodeLoader takes the file used for the program and reads its content to a buffered reader. Then the class is used to call the function loadCodes() that I had to implement. The code has to tokenize the string for the function to read. Grab the class name of the current line and create an instance of that type of Byte Code. Then if the line has any additional arguments, it takes it and attaches it to the Byte Code for when it is later added onto the Program class which holds an ArrayList of ByteCodes.

The class Program is where the code resolves the target address of where certain byte codes have to go. This particular section is where I struggled a lot to code and in all honesty it is still not done. I had no clue where to go with this.

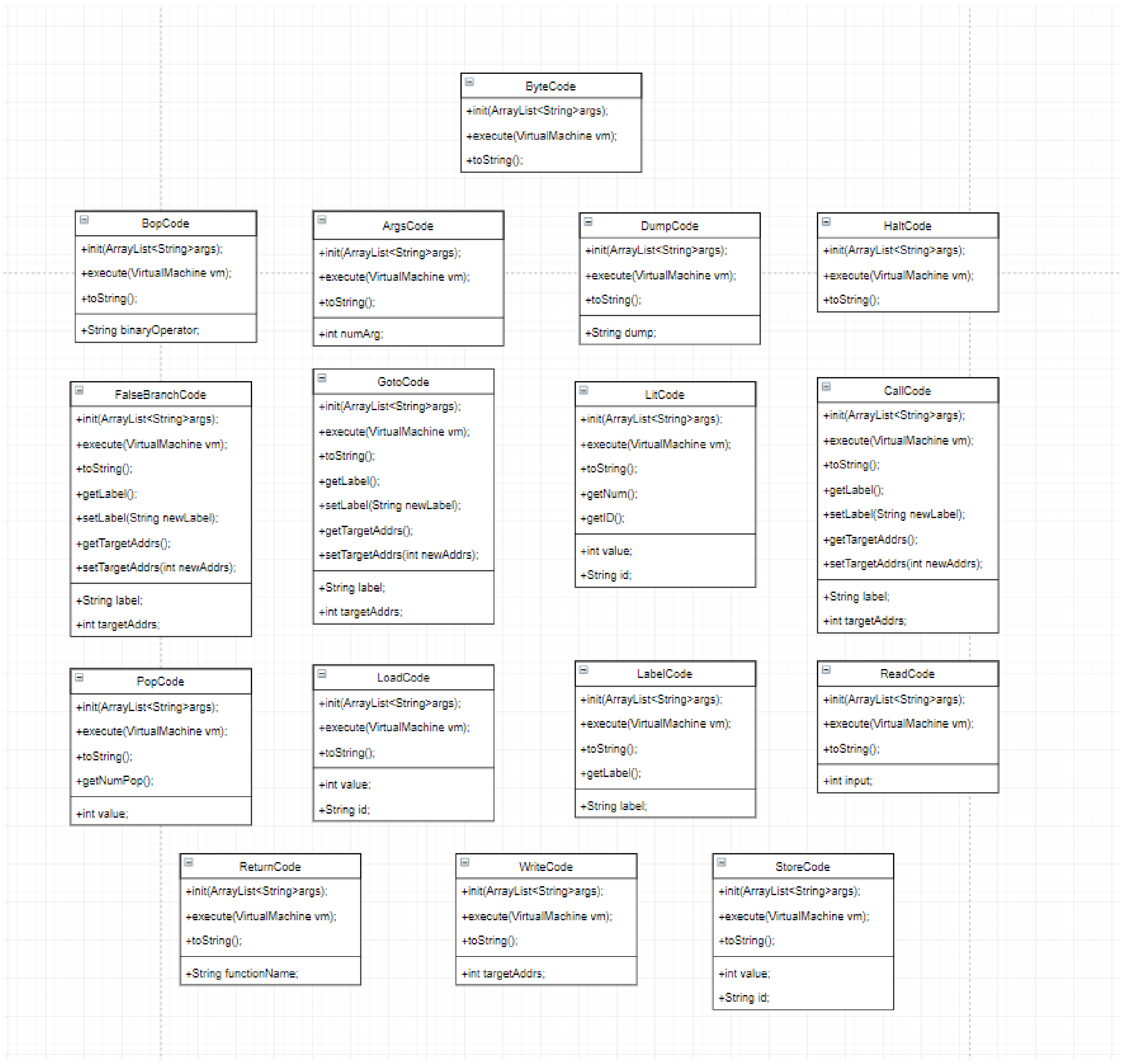
The RunTimeStack class was fairly easy to implement as it is described in detail in the instructions. This stack is where a lot of the logic occurs for the program. The priority in this class is the dump function as it outputs the answer in a formatted form for the user to see the current state of the runtimestack.

The VirtualMachine class is where the program executes the program to be used. It iterates through every bytecode on the program array list and executes them. The rest of the functions in this class are getters/setters that certain byte codes will need for their functions.

Implementation UML



NOTE: This showcases the basic hierarchy of the classes used for the project (ByteCodes not included in graph yet).


ByteCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
BopCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+String binaryOperator;
ArgsCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+int numArg;
DumpCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+String dump;
HaltCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
FalseBranchCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+getLabel();
+setLabel(String newLabel);
+getTargetAddrs();
+setTargetAddrs(int newAddrs);
+String label;
+int targetAddrs;
GotoCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+getLabel();
+setLabel(String newLabel);
+getTargetAddrs();
+setTargetAddrs(int newAddrs);
+String label;
+int targetAddrs;
LitCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+getNum();
+getID();
+int value;
+String id;
CallCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+getLabel();
+setLabel(String newLabel);
+getTargetAddrs();
+setTargetAddrs(int newAddrs);
+String label;
+int targetAddrs;
PopCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+getNumPop();
+int value;
LoadCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+int value;
+String id;
LabelCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+getLabel();
+String label;
ReadCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+int input;
ReturnCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+String functionName;
WriteCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+int targetAddrs;
StoreCode
+init(ArrayList<String>args);
+execute(VirtualMachine vm);
+toString();
+int value;
+String id;

NOTE: These Byte Codes are an extension of CodeTable that are used for the rest of the code.

Project Reflection:

From the beginning I knew that this would be one of the toughest assignments that I would have to complete in all years I have taken at SFSU. Even though I understand how the Algorithm for the program works and the flow of it all, I still failed to get it to complete the task of the sample code. If I could have asked a few more questions, if I could have read a little bit more, if I could have had more time, I believe I could have gotten this to work. Alongside this program I had to work on another project in CSC 415 that split my time between the two. I was able to implement the Byte Codes and the ByteCodeLoader class. Where I believe my issue for the program lie, is in the Program.java, with some partial problems in RunTimeStack.java and VirtualMachine.java. My program launches and runs until it gets to a READ code. Once it asks for a value, it infinitely continues to ask for values to push onto the run time stack. As someone mentioned on slack, the problem could be due to the fact that the PC (program counter) doesn't proceed past the READ and CALL code. Once the code CALLS to a target address, it moves the PC back to that code and runs again. But once it completes its task it has no way of returning back to its original position. I thought of resolving this by modifying the VM itself to bypass this, but as stated already, that would have been the wrong way to do this program.

Project Conclusion and Results

I am incredibly disappointed in failing to complete this program. If I find the time in the future, I might come back to complete it just for the sake of it. The code technically does compile, but it fails to complete the program. My biggest struggle was understanding the algorithm whilst working alongside another coding project.