CSC 413 Project Documentation

Fall 2019

Ufkun Erdin

911022312

CSC413.02

GitHub Repository Link

Table of Contents

1 Introduction 3

1.1 Project Overview 3

1.2 Technical Overview 3

1.3 Summary of Work Completed 3

2 Development Environment 3

3 How to Build/Import your Project 3

4 How to Run your Project 3

5 Assumption Made 4

6 Implementation Discussion 4

6.1 Class Diagram 4

7 Project Reflection 5

8 Project Conclusion/Results 6

1 Introduction

1.1 Project Overview

Because a machine does not speak/understand “human speak”, when we write code that is not read by the machine the same way we read it. Compilers are programs that translate people-friendly code to machine language instructions that the machine will be able to understand and execute. Some languages, such as Java or – in this case, X – are both compiled and then interpreted by two separate programs.

For this project, we had to build an Interpreter for the language of “X”. We were given several compiled bytecode files that our interpreter would read and process to illustrate how interpreted languages are processed.

1.2 Technical Overview

The first step in building the project is to create classes for each bytecode. This way, we can store information about each individual instruction (ie: the literal value to be loaded, offset for arguments, address of the call instruction, address of the branch destination, etc).

Once each bytecode object was built, the next focus was on being able to load them in to the Program object and then resolve all of the symbolic addresses: that is, wherever there were branching instructions – we would need to define what line in the program each would branch to.

After all of this was done, building the Virtual Machine (VM) and the Runtime Stack (RTS) objects were relatively straightforward. Majority of time spent on the project, at this point, was spent debugging the three test cases (factorial.x.cod, fib.x.cod, and functionArgsTest.x.cod).

For the BOP bytecode, we created an extra set of classes called “operations”, similar to how we handled the various operation classes in the calculator project. Rather than having one giant selector in the BOP class, we created a HashMap<String,Object> that allowed us to match each operation character to a separate operator class that would perform the calculation and return the result.

1.3 Summary of Work Completed

All of the code written was built by me.

2 Development Environment

This project was built in JDK 12 using IntelliJ IDEA Ultimate 2019.

3 How to Build/Import your Project

Building/importing this question is fairly straightforward. Everything required to run the project is within the project folder.

4 How to Run your Project

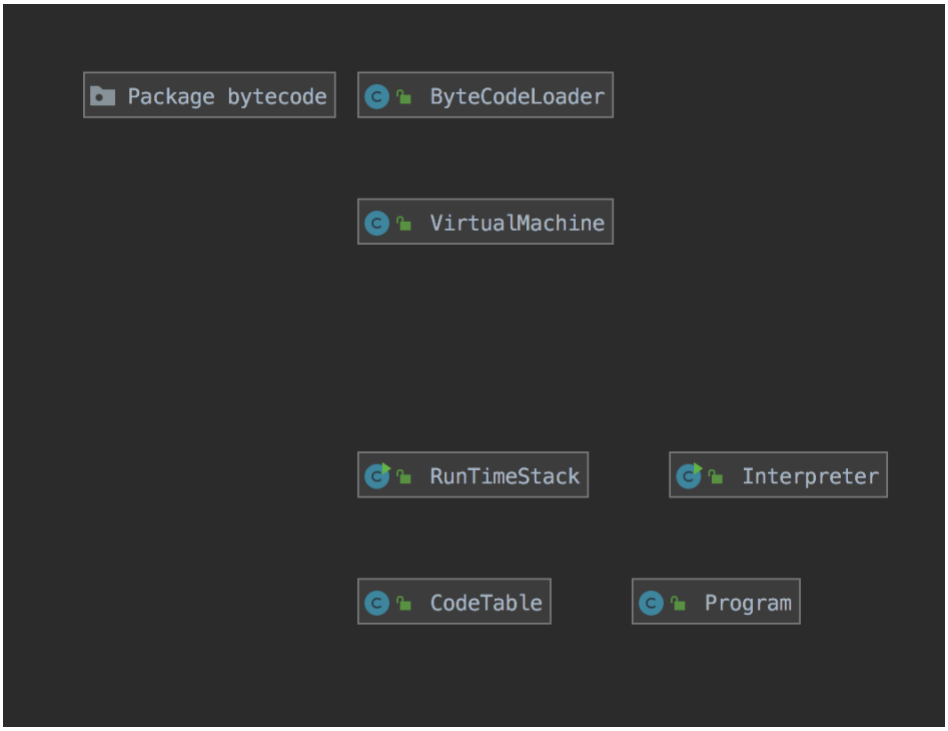
Simply create a configuration that executes interpreter.Interpreter passing an argument of one of the three *.cod* files named above.

5 Assumption Made

When designing this interpreter we had to operate under the assumption that the code given to the compiler was functional and that syntax or runtime errors would not occur. We had to assume that whatever instructions are in the bytecode will function exactly as the user intended for their code to function.

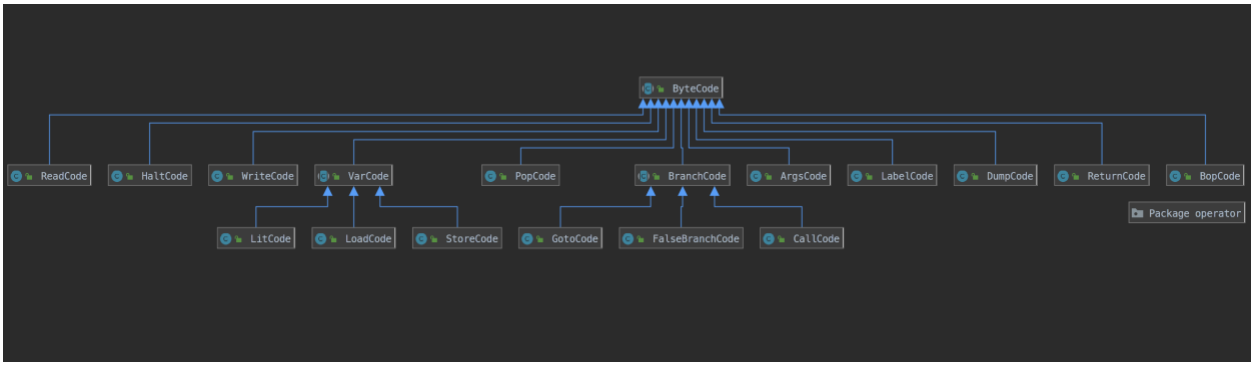
6 Implementation Discussion

6.1 Class Diagram



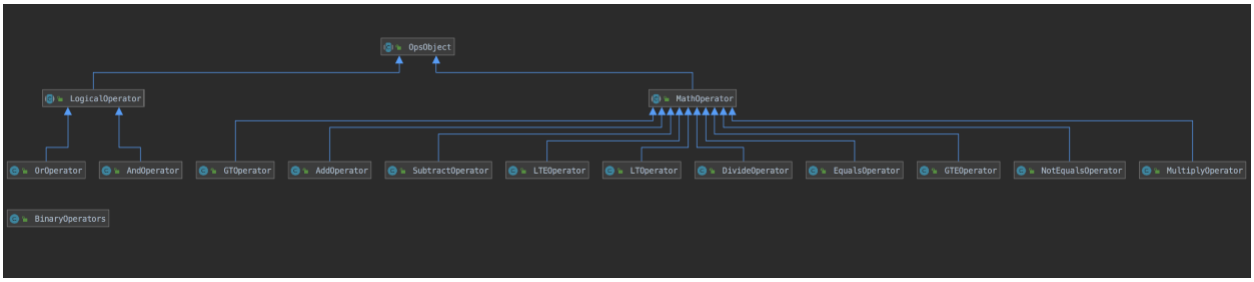
Starting from the highest level, we had the main interpreter package. Within this package, the classes are shown below. **Interpreter** was responsible for being the primary driving force/main class of the entire program. **ByteCodeLoader** would read through the given output file and, using **CodeTable** would determine which bytecode object (package bytecode shown below) would need to be created for that particular bytecode line. Once it created the **ByteCode** object, it would load it onto our program stack in the **Program** class. Once all of the bytecodes were loaded, **ByteCodeLoader** returned the Program object with the program stack back to Interpreter which then called the **VirtualMachine** and passed along the program to be executed.

The **VirtualMachine** acted as an interface, or mediator, between the bytecodes and the **RunTimeStack**, which held the results of all of the operations from the respective bytecodes.



The **ByteCode** class was divided in to several sub-classes. **VarCode** and **BranchCode**. The **VarCode** split may have not been fully necessary, but it was mostly there as a way to illustrate that Lit, Load, and Store all behaved very similarly. The **BranchCode** was the most important subclass definition here, however, because it was heavily used in preparing the Program data structure for the symbolic address resolution after all the bytecodes had been added. Having this class definition greatly reduced the number of if statements and also reduced the number of iterations that had to be done for the address resolution (discussed further in the conclusion).

Also of note, in an effort to reduce the number of selector statements and make the project more dynamic, I created another package called operators. These held all of the various binary operations that the program would need to perform (mathematical and logical). Pictured below is the operations package.



The **BinaryOperators** class was modeled similar to the Operators class from the calculator project. It has a static block of code that initializes each operator object and maps them to their respective tokens in the HashMap. The **BopCode** object used the **BinaryOperators** class as an entry way to easily pass along which operation to perform for each binary operator token. This also made the case of handling logical operations such as and/or much easier than its initial form as part of a giant switch statement inside **BopCode**.

7 Project Reflection

This project was very enjoyable to work on. It was a great exercise in instancing and learning how to manipulate objects (as is the whole purpose of OOP) to create a dynamic and expandable program. I found myself going back through code I had already written about every couple days and reviewing it, making sure it was efficient as could be and that everything had a wide-to-narrow sort of hierarchy.

8 Project Conclusion/Results

Overall, I am quite pleased with how the project came out. The one area that I would have liked to allot more time for, however, was doing a lot more stress testing to try and break the ByteCodeLoader to have unique actions for each of the various exception cases rather than having them just exit the program. I also would have liked to spend more time trying to come up with a more efficient solution for the symbolic address resolution. In the initial build, I would only save the label's lines and symbolic addresses, then I would parse through the entire program and resolve the symbolic jump addresses for all of the BranchCodes. Right now, it goes through fewer iterations because there are the two hashmaps that work in conjunction with one another so that it only goes to the relevant lines in the program data structure. However, I'm not sure if this method is any more or less work in terms of total number of iterations, as it is parsing through the labels hashmap to match the string key. Additionally, I am creating two additional data structures which (though inconsequential in this case) does still take up memory.