CSC 413 Project Documentation

Spring 2020

Duy Nguyen

Student ID: 917446249

CSC 413.03

GitHub Repository Link

https://github.com/csc413-03-spring2020/csc413-p2-duynguyen2

Table of Contents

1 Introduction

1.1 Project Overview

This is an Interpreter program, which takes this programming language X, and interprets the language and runs a program using it. This language X is just a simple one that uses only a few commands to run a program, very similar to contemporary ones. The Interpreter will go into files in the X language and interpret what its instructions. It will read files with type x.cod at the end of their names. The interpreter only supports certain commands. The two files that can be tested are factorial.x.cod and fib.x.cod. The factorial file takes a numerical input and computes the factorial, with a few limitations into what it can compute. The fib file takes a numerical input and computes the Fibonacci sequence, which is a series of numbers where the following number is the sum of previous 2.

1.2 Technical Overview

The Interpreter program takes mock language X, it is like a simpler form of Java, and it processes byte codes created from source code files with the extension x. It runs takes and runs it alongside a virtual machine. The byte codes are put into run time stack (RTS) and there is a frame pointer stack (FPS) to hold and maintain the activation records.

The Interpreter Class is where the programs get run. It performs all initializations, loads the ByteCodes from the source file and runs the virtual machine to follow the files instructions.

The RunTimeStack class is what makes RTS and FPS. It has a constructor method to create both, where the RTS is an ArrayList of Integers and the FPS is a stack of Integers. It has a dump method, which will print the frame. It also has the usual stack methods, pop, push, peek, and also methods to pop and add to the FPS.

The CodeTable class has a HashMap to map keys for the ByteCode class names. It has a method getClassName, which will take a string as an argument and return the corresponding class name from the HashMap.

The Program class stores all the ByteCodes read from the source file into an ArrayList and it also handles address resolution. It has a constructor method to allocate memory and a getSize method to get the size of the ArrayList of a program object. There is a getCode method which takes an integer and returns the ByteCode from the ArrayList that is at that index. The addCode will take a ByteCode object and add it to the ArrayList. The resolveAddress method will take a program object and uses a HashMap to map labels to the line numbers that they occur. It will loop through the ArrayList for Label ByteCodes to add to the HashMap. Then it loops through to look for FalseBranch, Goto or Call ByteCodes, and if they are found then it will get the line number and set a destination address to it.

The ByteCodeLoader class is used to initialize the ByteCodes using their init methods and adds them to the programs ArrayList. It functionality is in the loadCodes method, which uses a BufferedReader to go through line by line and make an instance of the appropriate ByteCode.

The VirtualMachine class is what controls the program. It feeds request to the RTS when certain operations are needed but not in the respective scope. The VirtualMachine has a constructor method that takes a program object, and an executeProgram method which will run the program given. This is

also the only place that dumping occurs. It also has multiple methods to help complete tasks that cannot be accessed by other classes, such as popping the RTS.

The ByteCodes the interpreter supports are the Args ByteCode, Bop ByteCode, Call ByteCode, Dump ByteCode, FalseBranch ByteCode, Goto ByteCode, Halt ByteCode, Label ByteCode, Lit ByteCode, Load ByteCode, Pop ByteCode, Read ByteCode, Return ByteCode, Store ByteCode and Write ByteCode. Each of the ByteCodes are subclasses of an abstract class, ByteCode. They all must have the methods init, execute and toString. The init takes an ArrayList of strings and initializes the arguments. The execute takes a VirtualMachine as an argument and execute the function of the ByteCode called. The toString is to just format and returns a string that shows what is happening and what ByteCode is being called.

The Args ByteCode sets up how many arguments the function has. It will always be executed before the Call ByteCode, and takes only one argument, which is the number of arguments for the following function call. This number is used to figure out how many values from the top of the RTS will be a part of the activation frame of the next function call. It will find where the frame begins and push that index into the FPS.

The Bop ByteCode implements binary operations, such as basic mathematical operators and logical operators. It removes 2 values from the RTS, completes the operation asked using them and pushes the result back into the top of the stack.

The Call ByteCode is how the VirtualMachine moves from each location in the program to execute the code sections we call Function. When encountering this ByteCode, it jumps to the corresponding label in the program and is responsible for where control goes after a function is completed.

The Dump ByteCode turns dumping ON or OFF, where dumping only occurs when it is ON. Dumping is just printing information, such as what is in the RTS. This ByteCode is not to be dumped, however.

The FalseBranch ByteCode executes conditional jumps. It only takes a Label as an argument and remove the value of the RTS and check if that value is 0. If so, jump to the label given, otherwise move to the next ByteCode.

The Goto ByteCode takes one Label as an argument and is used to jump to it unconditionally, unlike FalseBranch. This means, regardless of the state of the program, a jump will occur to a given label.

The Halt ByteCode will stop the VirtualMachine object from running.

The Label ByteCode has no function and only marks the location in the program where other ByteCodes can jump to.

The Lit ByteCode pushes the literal values to the RTS. It can take one or two arguments and only pushes one value to the RTS.

The Load ByteCode moves values from an offset in the current frame to the top of the stack. This offset works from the beginning of the frame.

The Pop ByteCode takes an integer value and pops the same number of objects in the VirtualMachine object's RTS.

The Read ByteCode takes user input from the keyboard and the only inputs allowed are integer values.

The Return ByteCode helps return from functions and puts the correct return values in the correct position in the RTS. This is used for convention for returning values and handling arguments. Functions are required to return values in the correct spot in the RTS before returning said value.

The Write ByteCode displays the information to the console, and the only thing it can display is the top of the RTS and nothing else.

1.3 Summary of Work Completed

The only classes that didn't need to be touched is the Interpreter and CodeTable classes. I added all the needed methods for all of the classes and created a ByteCode abstract class to create all of the subclasses. There are 15 subclasses with the one ByteCode abstract class. All of those subclasses are the separate ByteCodes and all of them have the init, execute and toString function. Some have additional methods, partially for encapsulation, and more for better functionality. Most of the work followed the recommendations and hints given from the PDF, and some were preferential choice.

2 Development Environment

The IDE used to program this project is IntelliJ using JDK 12.

3 How to Build/Import your Project

This project is made with IntelliJ, so it is recommended to import the project into the same IDE since I have not tested it with any other IDE. To import the project, click the green button that says "Clone or Download" on the GitHub repo's home page. Then take the HTTPS or SSH link, but if you do not really understand SSH then use the HTTPS link. Copy the link and open your OS's terminal or Git Bash. Then in the terminal type "git clone [url copied]" and it will clone the repo. Alternatively, the Github desktop app can be used to pull. It is necessary to pull everything from the repository, otherwise it will not work as intended and the file structure needs changing.

Then open IntelliJ, or another IDE, and select Import Project and select the calculator folder as the source. Check the "Create project from existing sources" and go next. The default fields don't need to be touched so you can hit next. Make sure there are two items checked then hit next again. There should now be a list of JARs for the unit tests. Then there should be a list of folders selected and then a JDK should be selected. If not, then install JDK 12 but if it still not there then hit the plus button to show it, otherwise continue. There shouldn't be any framework and the project should be imported properly. All the folders are there and can be selected. To build it, you can run them by right-clicking the files and clicking the Run option or selecting them and hitting the play button.

4 How to Run your Project

To run the project, you need to right-click the Interpreter class and select "Edit Interpreter.main()…" option. A menu should pop up, and in the "Program Arguments" field, type in the x.cod you would like to run, ex. factorial.x.cod, fib.x.cod, etc. Afterwards, you can hit Apply and OK and you can right-click the file and hit Run or the play button for the Interpreter class.

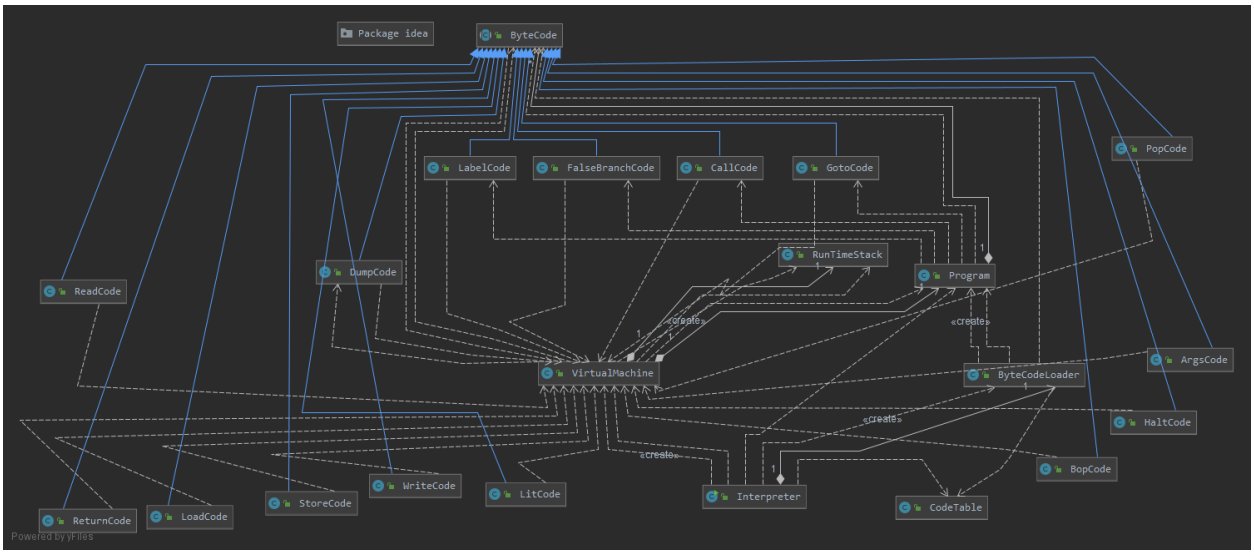
5 Assumption Made

- Divide by 0 is not tested

- Dumping happens after resolving addresses
- Only Integer calculations are done
- Neither of the given cod files should throw exceptions, but it should be able to handle exceptions

6 Implementation Discussion

6.1 Class Diagram



7 Project Reflection

The Interpreter definitely is harder than the Expression Evaluator. It was very helpful to go to lecture and hearing everyone's questions about frame pointer stacks, the run time stack and more. I initially wanted to use the Operator class from the Expression Evaluator for BopCode but as I was doing it, it seemed like a lot more work, so I just decided to use a switch case instead. It would be a lot cleaner, but it seemed like a long walk for a short glass of water. The whole project was a lot of typing. The ByteCodes took longer than I thought to make them all, not only because it's repetitively creating a bunch of them but also understanding what each one does and how dumping should be handled. The concept of ByteCodes was really difficult to understand for a bit with some of the names and making sure they do the right thing and has the right toString format. Once dumping worked, it made it a lot easier to figure out where the issues were, but it also took some time to get dumping to work. But this helped me realized how important it is to break down the big problem into smaller ones and solving them bit by bit. This took a lot of planning and a lot of trial and error to completely succeed.

8 Project Conclusion/Results

The interpreter is definitely a few steps higher in difficulty in comparison to the Expression Evaluator. It was a good way to realize that breaking down the problem is important and that big problems cannot be easily solved by directly trying to tackle them. Understanding the concepts before coding was imperative to being able to do the project, looking at examples and tracing the x.cod files is very helpful. It currently seems fully functional and follows the examples given on Slack.