Interpreter

CSC 413 Project Documentation

Spring 2019

Stephanie Sechrist

918679078

413.02

https://github.com/csc413-02-spring2019/csc413-02-stephlsechrist.git

Table of Contents

1 Introduction

1.1 Project Overview

For this project, I was asked to implement an interpreter for a made-up language X; the interpreter processes byte codes produced from source code written in X to run the X program. The interpreter works with an implemented virtual machine to do this. The project uses stacks, HashMaps, and ArrayLists.

1.2 Technical Overview

For this project, it is easier to explain the structure from the byte codes all the way up to the Interpreter. Each byte code is a child class of ByteCode, all contained in the same package. Not all of these byte codes have the same data fields or methods, so there is a separate child class for each one. However, all byte codes have an init() function and an execute() function. Init() is called to initialize the instance of the byte code; here is where we set any private variables, such as labels, variable names, or addresses.

These byte codes are initialized in ByteCodeLoader, which is also where the source file is parsed. The file is read with a BufferedReader line-by-line. For each line, a StringTokenizer is used to tokenize the string into parts delimited by a space. The first portion is the byte code. Java reflection is used to build instances of classes corresponding to the bytecodes. In addition, if the string has more tokens, they are added to an ArrayList<String> and passed into the respective byte code's init() function, where the arguments are assigned to private variables. Back in ByteCodeLoader, the byte code just instantiated is added to an ArrayList of byte codes called program.

These are added via the Program class, which contains the private program ArrayList; in other words, the program class stores all the bytecodes instantiated from the source file. Also in Program is the resolveAddrs() function that is in charge of finding the addresses for branches (FALSEBRANCH, GOTO, CALL) to jump to and assigning that address to each branch function. These addresses are designated by the byte code LABEL. A HashMap is used to store all the labels and corresponding addresses, and this HashMap is used anytime a label is encountered in the program. If the label is found, the branch byte code gets the corresponding address saved to a private data field.

This resolveAddrs() function is passed the ArrayList<ByteCode> from ByteCodeLoader and returns the ArrayList<ByteCode> with resolved addresses. ByteCodeLoader passes this back to VirtualMachine. VirtualMachine oversees Program and executing the byte codes. The boolean isRunning is set to true util the byte code HALT turns it false; then, the VirtualMachine stops executing byte codes from program. VirtualMachine has a RunTimeStack (RTS from here on) called runStack that keeps track of values in the byte code program. It also has a Stack returnAddrs that is used when RETURN is called to know which address to return to from the function we are in. Function scopes are delineated by frames in this implementation. Within the RTS, there are frames, and each frame is its own scope.

This RTS is manipulated and defined in RunTimeStack class. Here, we have functions to manipulate the RTS. Each function prevents any manipulation if it is not allowed. For example, the user is not allowed to pop the stack if the stack or the top frame of the stack is empty. These functions are called by the VirtualMachine, which is in turn requested by the byte code to perform operations on the run time stack or change the address being executed. Once all of the byte codes are executed and HALT is called, the VirtualMachine stops executing. Control goes back to Interpreter, and the program finishes.

1.3 Summary of Work Completed

For this project, Interpreter was provided and not allowed to be altered. CodeTable was also provided, and I did not make any changes to it. Some code was provided in ByteCodeLoader, Program, RunTimeStack, and VirtualMachine, but I completed the rest. Finally, all byte codes were created by me. Below is a more detailed list of added classes and methods:

In ByteCodeLoader:
- implemented all of loadCodes()

In Program:
- implemented getSize(), which I ended up not needing but left in anyway in case of future updates
- implemented resolveAddrs()
- implemented addCode()

In RunTimeStack:
- implemented all methods except the constructor

In VirtualMachine:
- implemented executeProgram() from base code provided by assignment PDF; added if statement for when dump is on
- implemented everything else except the constructor

2 Development Environment

For this project, I worked solely in IntelliJ IDEA 2018.3.4 (Ultimate Edition) on Windows 10. The version of Java being used is 11.0.2.

3 How to Build/Import your Project

1. Use given git repository link to clone the project into desired folder.
2. Open IntelliJ IDEA and select Import Project in the Welcome window.
3. Find the folder that you cloned the git repository to and open the project folder (should be named csc413-02-stephlsechrist).
4. Highlight the csc413-02-stephlsechrist folder to be the root of the source files. Click OK.
5. Select “Create project from existing sources” and click Next.
6. Accept the defaults for naming by clicking Next.
7. Accept the defaults for default root by clicking Next.
8. Accept the default libraries (none) by clicking Next.
9. Accept the default module (should be csc413-02-stephlsechrist folder) by clicking next.
10. Select project SDK. You should be using JDK 11. In Name field, type 11. If not already found for you, find your JDK home path. Click Next.
11. Next page should say “No frameworks detected.” Click Finish.

4 How to Run your Project

There are two included source codes to run: factorial.x.cod and fib.x.cod. To run these, you have to configure the Interpreter class (which has the program’s main method).

1. Right click on Interpreter and select **Run ‘Interpreter.main()’**

2. You should see in the dialog box, ***Incorrect usage, try: java interpreter.Interpreter <file>
3. Above editor, click the drop down menu next to the Build Project icon (green hammer) that should say **Interpreter**. From this menu, select **Edit Configurations…**
4. Under Application > Configuration, enter the source code name you would like to run in **Program Arguments** box.
5. Click **Apply** and **OK**.
6. Now you can run the project again.

If you have Java installed, you can run the program from the command line using one of the source codes as an argument and Interpreter as the program.

5 Assumptions Made

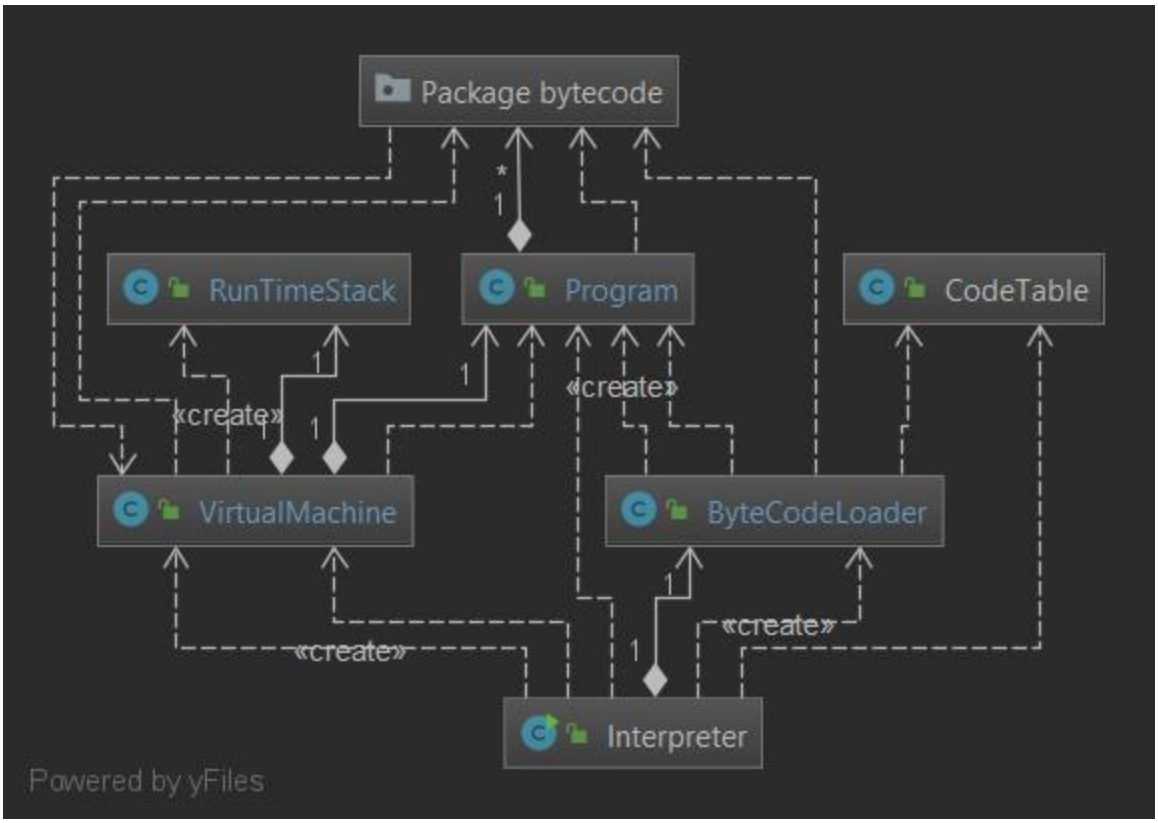
In this project, it is assumed that the source code provided only contains accurate byte codes listed in the code table and correct arguments in correct order. It is also assumed that the user will not try to use BOP to divide by zero, so that test case is not accounted for. We also assume that all values on the run time stack are ints. I also assume that the user will not try to crunch giant numbers, as int overflow is not required to be addressed in this assignment.

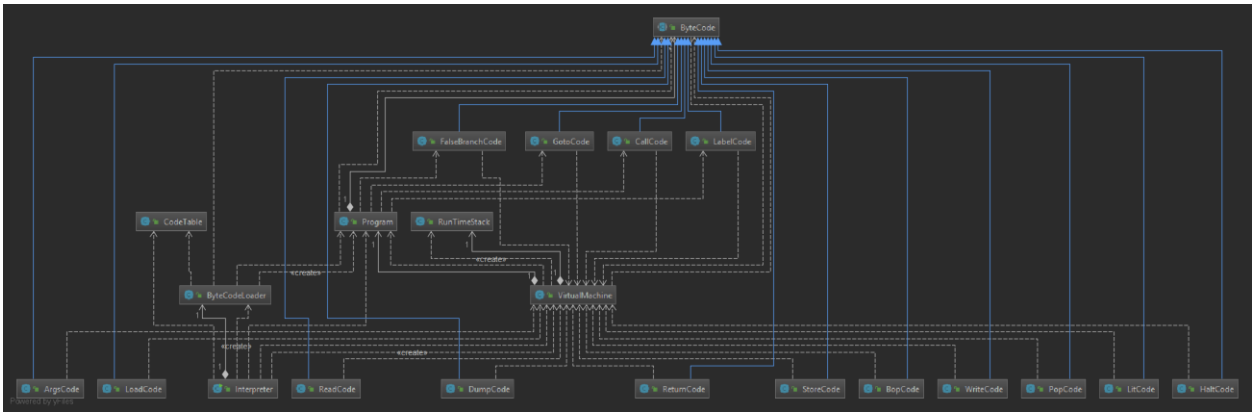
6 Implementation Discussion

Ran out of time to expand on implementation discussion. See Technical Overview section for some explanation.

6.1 Class Diagram

The class diagram shown below is to show how the class relationships exist. The Package bytecode is expanded in the second diagram. It is hard to see the details, so please look at UML.png that is in the documentation folder alongside this PDF.





7 Project Reflection

This project was slow-going at first, but once I understood all the moving parts, it was pretty smooth. It took a while to get through the PDF and really understand how each object interacted with each other, as well as what was in control of what. The VirtualMachine class was a little tedious to write, because I basically wrote a function corresponding to every runTimeStack function; however, all the other functions were fun! This project really came to life once I understood how each piece fit together. It was a great lesson in encapsulation, and I think I implemented a design that did not break encapsulation. Overall, the project took me a long time to complete, and I am ready to move on from it, but again, it was a great learning experience. I think this is the first program I have written with this many different data structures attached, so I really feel like my knowledge was tested.

8 Project Conclusion/Results

This project was designed, I believe, to force us to really think about object-oriented design in terms of encapsulation and efficiency. I think I had a good grasp on how to implement this project, and I am happy with my results. The program works as expected, and I believe I used as efficient methods as I could. If I were to update this project in the future, I am not sure what I would change. I am a little disappointed that I could not figure out how to dump the run time stack with proper frame boundaries without getting help, but my collaborator had a great algorithm that I could not find a way to improve on. I am also sure I could probably find a more efficient way to deal with the byte codes. There was talk in the PDF and on slack about finding what byte codes had things in common and using that to prevent duplicate code, but I ended up executing each byte code in its own class. I would need to think a little more about how to cut down on my lines of code. Finally, I wish I had more time! I am done with the project and happy with the results, but I am not happy with what I am submitting for documentation!