Interpreter Documentation

Spring 2019

James Pratt

SFSU

Software Development Class

Table of Contents

1 Introduction

1.1 Project Overview

This is a class project from my junior year in college for my software development class.

This project is an interpreter for the mock programming language X. What this means is it takes your program's source code, breaks it up into readable pieces, and properly executes each line. There are many different commands that are used to execute each line. These commands include reading a user's input, loading their input into the program, calculating the correct output, and printing the output. There are many other levels that go into executing these commands – for example creating a function, which is like a block of code that is needed to be executed before returning to the rest of the code. This project knows when to create these functions, and when to exit them.

There are two example programs that are written in this mock language X: one that computes the nth Fibonacci number, and one that finds the factorial of a number. When entering in a small number, the program will properly calculate the result and print it out for you. All that needs to be done to run these files is to add their title as a program argument in the configuration's menu.

1.2 Technical Overview

Going off of what was explained in the project overview, there are several areas of importance for this interpreter, and they are all focused on exercising proper encapsulation. Our main method is in Interpreter.java. Once it detects a file is to be run, it creates a program, creates a virtual machine, then executes that program within this virtual machine.

When creating the program, it needs to do a few things: parse each line of the file, then put each command into a hash map (making sure to resolve each address). This is all done within our ByteCodeLoader.java class. When resolving an address, we are simply matching up our jump codes. If we have a label attached to a GOTO, it will know that it needs to jump to the matching label after LABEL. Likewise with CALL and FALSEBRANCH. Once all addresses are resolved and our hash map is filled out, we move on to creating our virtual machine.

When creating the virtual machine, we also create two important data structures that we will be using all throughout the program: the runtime stack, and the frame pointer stack.

Once we call the virtual machine to execute, it begins reading each byte code line of the program. It is able to navigate each line through programCounter, and goes into a while loop until it reaches a byte code called HALT.

There is an abstract class called JumpCodes that extends ByteCode. This abstract class keeps track of where specific commands need to jump to when they are called. The other classes extend straight to ByteCode because not all classes need jump codes. When we come across byte codes such as GOTO, CALL, FALSEBRANCH, and LABEL, we will have the value at which they need to jump to – which had been instantiated once their init function was called.

The other classes are self-explanatory. We have LIT and POP which push and pop from the runtime stack. STORE and LOAD push and pop based on a given offset. READ and WRITE takes user input and prints the proper output from the top of the stack. BOP is our binary operator which takes the last two elements from the stack and calculates them, then pushes the value to the stack.

We will know when to execute certain functions by using the frame pointer stack. ARGS, CALL and RETURN set a specific offset to jump to, then returns to its position once it has completed the function. This offset is stored in the frame pointer stack, then discarded once completed. The frame pointer stack determines where in the runtime stack the function begins and where it ends.

With these byte codes in place, this program will be able to execute mathematically based programs by reading their source code.

1.3 Summary of Work Completed

I was given skeletons of files for Program, RunTimeStack, ByteCodeLoader and VirtualMachine. Interpreter was instructed not to be touched.

I edited ByteCodeLoader.loadCodes() so that it properly parsed each string in the file and store it in a hash map. I spent tedious amounts of work on Program.resolveAddress() to properly map each function call to its correct address. I created and implemented a dozen methods in the RunTimeStack() class that the virtual machine would pull from. I created a dozen individual byte code classes as well as an abstract class – JumpCodes – that specific codes would extend based on their unique needs. I created the methods needed to query the runtime stack and frame pointer stack from VirtualMachine() to RunTimeStack().

2 Development Environment

Version of Java used: 11.0.5

IDE: IntelliJ IDEA

3 How to Build/Import your Project

To import this project into IntelliJ, you want to download the repository from the link on the front of this documentation. Once cloned, open IntelliJ and select import project. The root of the project is one level above the interpreter directory. Click "Create project from existing resources" button. Click next until you can click Finish.

4 How to Run your Project

Now that the interpreter is loaded into IntelliJ, find main>java>edu.csc413.calculator>interpreter>Interpreter. Find the drop down menu next to the green arrow and click on "Edit Configurations..." and under program arguments type in the name for the SOURCE CODE file you would like to run (For example, factorial.x.cod). Click on the green arrow at the top right of the screen to run the code. The program should successfully run.

5 Assumption Made

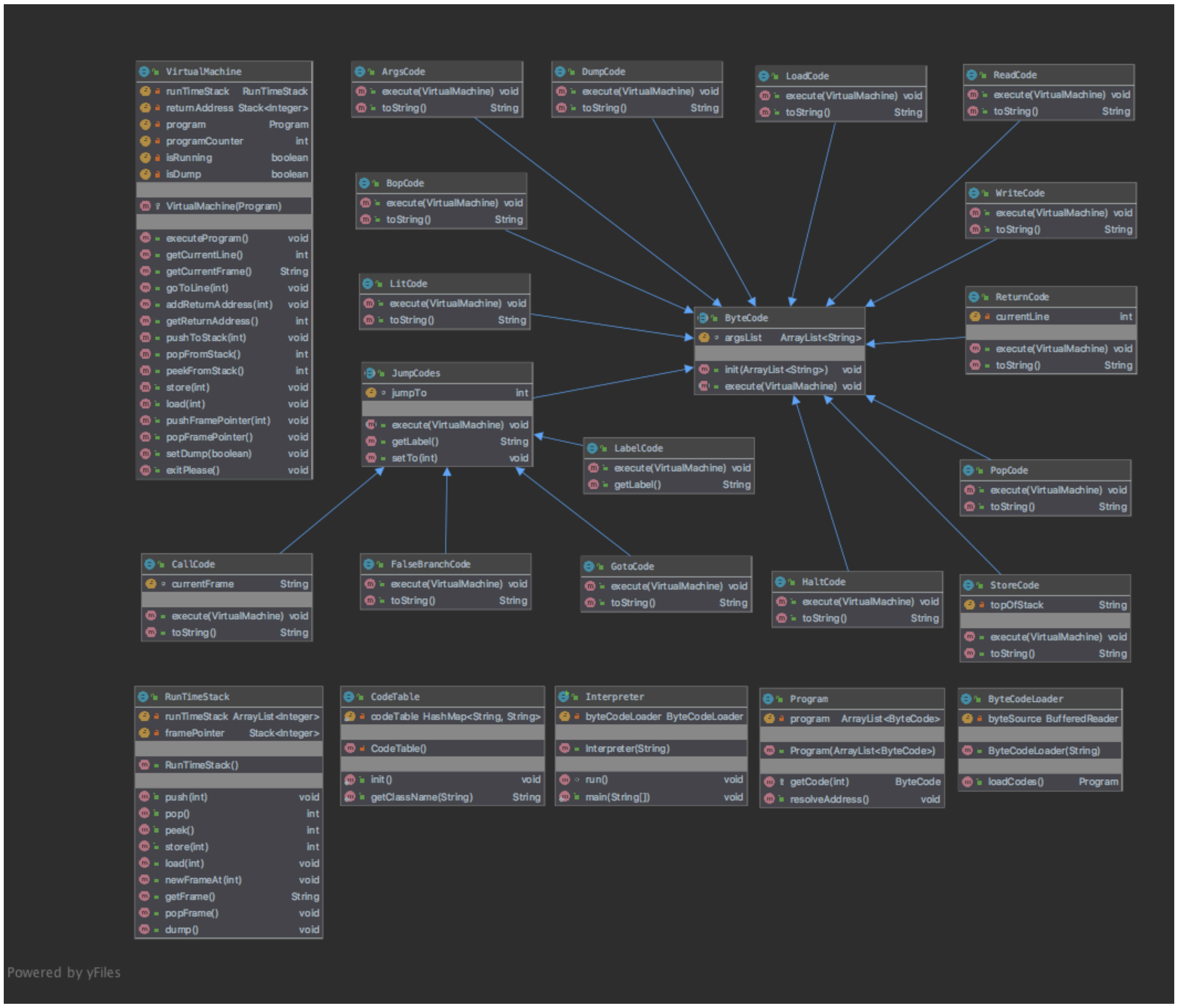
When taking on this project I did what instructor Souza requested and read the PDF several times. The most confusing part was figuring out java reflection, then figuring out the bytecode loader class. I followed the directions on the PDF suggesting where to start, and things just started coming together little by little. I knew this was going to take a long time to figure out, so I started right when we got assigned. I just didn't entirely know what I was doing until a week before it was due.

6 Implementation Discussion

As for the structure of the assignment, I wanted my byte codes to be concise and organized. They have clear variable names and comments when need be.

For the UML Diagram I have Label, FalseBranch, Call and Goto as children of JumpCodes, which is the child of ByteCode. The rest of the ByteCodes extend ByteCode.

6.1 Class Diagram



7 Project Conclusion/Results

Results of this project should return the correct Fibonacci number / factorial. If you would like to step through the dumping process, simply turn dump on at the beginning by entering DUMP ON at the beginning of each program's source code.