Project 2: Interpreter

Spring 2019

Francis Santiago

Student ID: 917860819

CSC 413.02

GitHub Repository Link: https://github.com/csc413-02-spring2019/csc413-02-fpsntiago.git

Table of Contents

[1 Introduction 3](#_Toc522827688)

[1.1 Project Overview 3](#_Toc522827689)

[1.2 Technical Overview 3](#_Toc522827690)

[1.3 Summary of Work Completed 3](#_Toc522827691)

[2 Development Environment 3](#_Toc522827692)

[3 How to Build/Import your Project 3](#_Toc522827693)

[4 How to Run your Project 3](#_Toc522827694)

[5 Assumption Made 4](#_Toc522827695)

[6 Implementation Discussion 4](#_Toc522827696)

[6.1 Class Diagram 4](#_Toc522827697)

[7 Project Reflection 5](#_Toc522827698)

[8 Project Conclusion/Results 6](#_Toc522827699)

# Introduction

## Project Overview

This project is called the interpreter. We are tasked with completing the code that is provided to us. This program is designed to process the fake “X” language. The interpreter processes the “assembly” instructions of two files, fib.x.cod and factorial.x.cod. These files are example programs of the Fibonacci nth number and finding the factorial of a number. The program reads each program and interprets each instruction in the “X” language. This program is essentially a translator that converts instructions from a made-up language to execute commands that are available to standard programming languages.

## Technical Overview

This program goes through two files, fib.x.cod and factorial.x.cod. By loading the “Assembly” instructions of each file, the program is then executed depending on the certain instructions by putting each bytecode instruction/argument into an array list. We define each instruction by the contents of the x.cod file. After loading the commands, we isolate the class arguments and store them into a Hashmap depending on the index. The bytecodes are executed with the Virtual Machine, after they are pushed onto the stack then solved. Overall, the interpreter program takes the input from either Fib.x or Factorial.x and interprets them into instructions in the “X” language.

## Summary of Work Completed

# Development Environment

* IntelliJ or Eclipse
* JAVA SE Development Kit 11.0.2

# How to Build/Import your Project

* To import via IntelliJ, VCS -> Checkout from version control -> Git, then copy Github link
* After the project is cloned, follow the prompts that detect the source files and any additions.
* After this project should be ready to be built.
* On Eclipse, import the project as Maven

# How to Run your Project

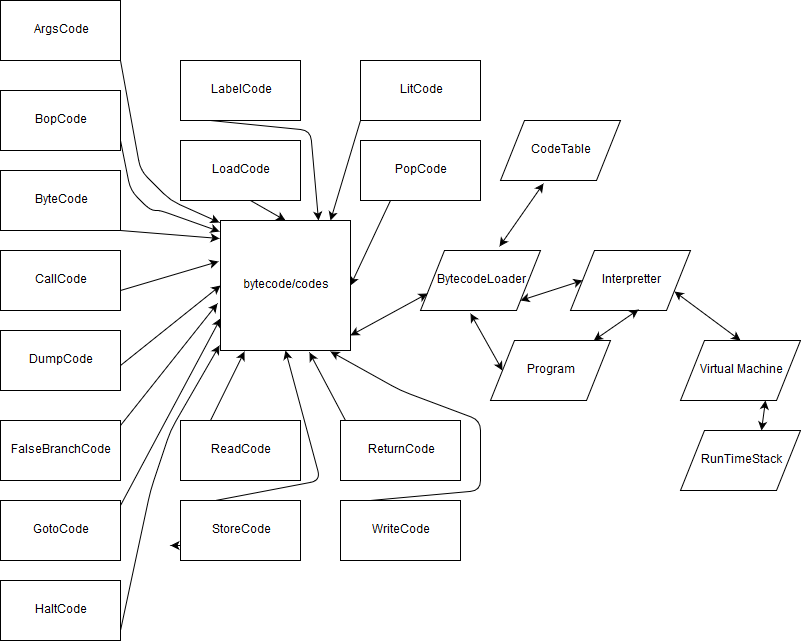
* After building the project, files that can be run have a green arrow next to them.
* Run Interpreter.java by right clicking it, make sure that the file to be read is selected.
* On Eclipse, run as Maven clean and install 🡪 run interpretter

# Assumption Made

The first assumption is that we can assume that both files that need to be interpreted need no modification. We then can proceed to determine what classes need to be created or completed. We also assume that the virtual machine is what is used to execute all the files. Another assumption is that we need to create each bytecode by reading the PDF and by looking at CodeTable.java.

# Implementation Discussion

## Class Diagram



The first step of this project was to complete the ByteCodeLoader class so that we can build towards accessing the ByteCodes. The purpose of this class is to Load the bytecodes to the program class by creating an instance specific to each ByteCode. ByteCodeloader is need for CodeTable.java, which is a hashmap of the bytecodes. We are also tasked with implemented program.java, this class contains methods that add instances of bytecode to the program class and is used to resolve the addresses of each bytecode. By using resolveaddrs, we compare the branchcodes Call, Goto, and FalseBranch. By looping this comparison, we call the instances of the branchcodes, and set the address used by other classes. We also need to implement RunTimeStack.java and VirtualMachine.java. RunTimeStack.java is used to manage the stack, this file is where pop, peek, push, store, load, reset, and creating new frames are used. RunTimeStack.java is also where we monitor active frames. The Virtual Machine is used to execute codes and works closely with Interpreter.java.

|  |  |
| --- | --- |
| ArgsCode.java | Used to set new frames in the interpreter |
| Bop.java | Used for arithmetic operations. Similar to an ALU. |
| ByteCode.java | Abstract class for bytecodes |
| CallCode.java | Switches control to other functions |
| DumpCode.java | Dump instructions or do not dump instructions |
| FalseBranchCode.java | Pop the top of the stack, check if false then branch to the correct label. |
| GotoCode.java | Allows jumping to a different label. |
| HaltCode.java | Stop program |
| LabelCode.java | Setup branch Targets |
| LitCode.java | Checks two arguments, loads value to the stack then initializes the variable. This action reserves space on the RunTimeStack. |
| LoadCode.java | Pushes value of the offset from the start of the frame to the top of the stack. |
| PopCode.java | Pops value of the RunTimeStack |
| ReadCode.java | Askes user to input a number that is placed at the top of the RunTimeStack |
| ReturnCode.java | Returns from current function |
| StoreCode.java | Pop the top of the stack. The value of the offset is stored at the beginning of the frame |
| WriteCode.java | Stores value of the top of the stack and writes the top value of the stack as an output. |

# Project Reflection

For this project, I had to read the PDF multiple times just to understand what this project was asking me to do. In the past I took a systems programming class that required us to also write a similar type of project that was used to take in assembly code and then execute it based on the given commands. So, for this project I was able to use my experience from that project as a starting point. However, this was probably one of the most confusing projects I have ever encountered. I collaborated with other students in the class to understand how we would have to approach each program. Through a lot of research, I was able to break down how each of parts we needed to complete were supposed to work. For this project I had to look up more java examples to properly write the code. I think there is a lot of code that can be written more efficiently. Reading the PDF as many times as you can is essential when approaching this project. I was having trouble with IntelliJ during this project, so I was switching between it and Eclipse. I think this was because I was working on two PC’s that shared files via cloud. It ended up messing up my commits and pushes so I just uploaded the files via the Github desktop application.

# Project Conclusion/Results

Overall, I was able to get something that builds. I think that this project is 95% close to working, I just ran out of time. The error that I keep getting, is an index error that occurs when trying to read the file through the interpreter. I was able to get some output but the more code that was added, I think the code broke. After comparing my results with others, I can see that what was submitted was on the right track.