CSC 413 Project Documentation

Summer 2022

Abbas Mahdavi

918345420

CSC 413-02

https://github.com/csc413-SFSU-Souza/csc413-p2-AbbasMahdavi021

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 4
3 How to Build/Import your Project 4
4 How to Run your Project 4
5 Assumption Made 5
6 Implementation Discussion 5
6.1 Class Diagram 5
7 Project Reflection 7
8 Project Conclusion/Results 7

1 Introduction

1.1 Project Overview

This program is an interpreter which is used to run programs loaded from a file that contains an X language complied bytecodes. It reads the file and performs the multiple different functions based on each line of text in that file. The two .x files are programs that take an integer, which is input from the user, and return a result, based on what the program is (either Fibonacci or factorial of the entered integer. The program reads through a stack of values that are separated by functions and prints out the correct result.

1.2 Technical Overview

The program reads a source code file which is written in the mock language of X, so the program acts as an interpreter/compiler for this language, which is just a simplified version of java. The files contain various bytecodes and arguments, which is used to turn a positive integer into a result based on the file program.

This programs work by having each byte code representing a specific function. If there is a function, the program parses each byte code to figure out what their function is. The RunTimeStack class manages a stack of values and frames in order to store values that are generated by the said byte codes and their function. These values are used to calculate a result of an input.

1.3 Summary of Work Completed

Firstly, I implemented the RunTimeStack class with its different methods, which are used to manage the runtime stack. I implemented the VirtualMachine class, which uses the methods in the RunTimeStack class. There were various methods that needed to be written in the Virtual Machine. Most of these are used as way to call the methods in the RunTimeStack class. And a few more that are used to get addresses and program counter. These methods in the VirtualMachine class are used by the ByteCode classes, as they are not allowed to call methods from the RunTimeStackClass.

I then created an Abstract class called ByteCode, which has 3 main parameters of Init, execute, and dump, and this class will be extended by the 15 subclasses which I created later on. I also created an Interface class called BranchCode, which is implemented by a few of the 15 subclasses. Before creating the 15 classes however, I impletemented the ByteCodeLoader class to parse the program argument file into the ByteCode objects and their arguments, if there are any, which correspond with their class name. These objects are then stored in an ArraList which are passed to a Program object, inside the loadCodes method. I implemented the program class and the resolveAddress method. This is used to resolve the address of the ByteCode objects that are implemented in the BranchCode interface. This allows the objects to know the correct address and so to what labels to point to, when the ByteCode in the program is executed.

I then implemented the 15 bytecode (sub)classes. All of these extend the ByteCode class which has a dump parameter, allowing these classes (except DumpCode, HaltCode, and LabelCode) to use the dumping function. Dumping function, when dumpingIsOn is set to True inside the VirtualMachine class, in the executeProgram method, it dumps the program inside the console, as well as the runtime stack and its frames. I then finished the executeProgram, which takes cares of running the program and dealing with dumping, when necessary.

2 Development Environment

IDE: IntelliJ IDEA Ultimate

Version of JDK Java: 18.0

3 How to Build/Import your Project

After cloning the project properly in to our computer,

- Open IntelliJ IDEA
- Click on Import Project
- Find the folder “csc413-p2-AbbasMahdavi021” directory
- Click on this folder and open it, making sure this is the directory folder
- If JDK is not setup already, setup JDK on the top right of IntelliJ
- Then build the project by IntelliJ nav bar, Build > Build Project

4 How to Run your Project

After importing the project correctly

- Navigate to IntelliJ nav bar, Run > Edit configuration
- This should show Application >/ Interpreter on the left side of the menu
- Inside this menu, the JDK must be set (perf. 18), and to the right of it, MainClass should have “interpreter. Interpreter” in it.
- We can now decide what x language file/program to run, as there are multiple.
- We
- In the program arguments enter either “factorial.x.cod” , “fib.x.cod”, or “functionsArgsTest.cod”, depending on which one you want to run first.
- Click Apply then Ok.
- We must come back to this menu, and change the Program arguments, to another language x files, ending in .cod, mentioned above, to run them.
- After entering the desired .cod file name in the Program Arguments section, we then navigate to InterlliJ nav bar, Run > Run Interperter. This can also be done in top right of IntelliJ next to the drop down menu, by clicking the green run button.
- “factorial.x.cod” , “fib.x.cod”, will ask for a positive integer, and print a integer result.
- “functionsArgsTest.cod” is a test and dumps and prints huge result.
- “factorial.x.cod” , “fib.x.cod” can also use dumping function, but this needs to be turned on in the VirtualMachine class > executureProgram method, and switching dumpingIsON to true, then running the program. This will again take an integer, but will also dump much more info, similar to the “functionsArgsTest.cod” program argument.

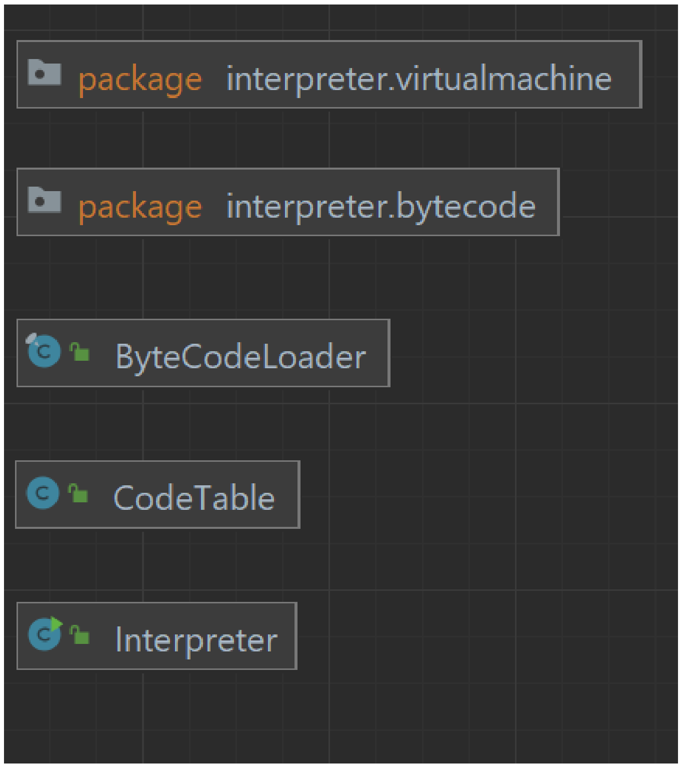
5 Assumption Made

- The .cod files do not contain any errors.
- Interpreter Class must not be changed.
- CodeTable class work properly.
- Popping past any frame boundary is not allowed in RunTimeStack
- RunTimeStack will be popped when empty.
- ByteCode classes, in the execute parameter, are allowed to call methods in VirtualMachine, that call methods in the RunTimeStack class.
- Values in the ByteCode classes must remain private.
- ByteCode classes must have init and execute function, and must a dump function, but these can be empty when they are not needed
- ByteCode classes cannot call methods from the RunTimeStack, such as the dump method
- ByteCode classes need an abstract class as well as an interface
- Each value from the framePointer stack corresposnds to an index in the RunTimeStack and these indices indicate the start of each frame in the RunTimeStack

6 Implementation Discussion

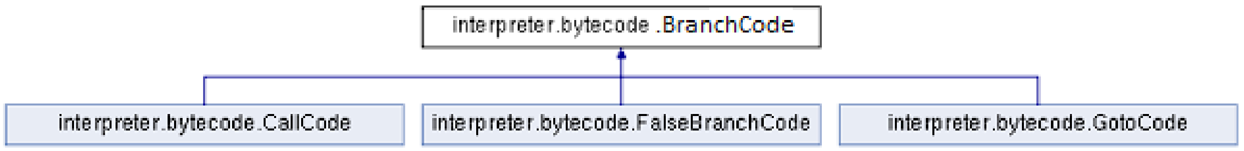
Each byte code subclass will have its own file and all byte code classes will be stored in the same directory including the interface that distinguishes byte code classes which need address resolution. ByteCodeLoader.java will read the program argument file and create byte code objects based on the file line by line. It will only look in the bytecode directory to identify unique byte code objects.

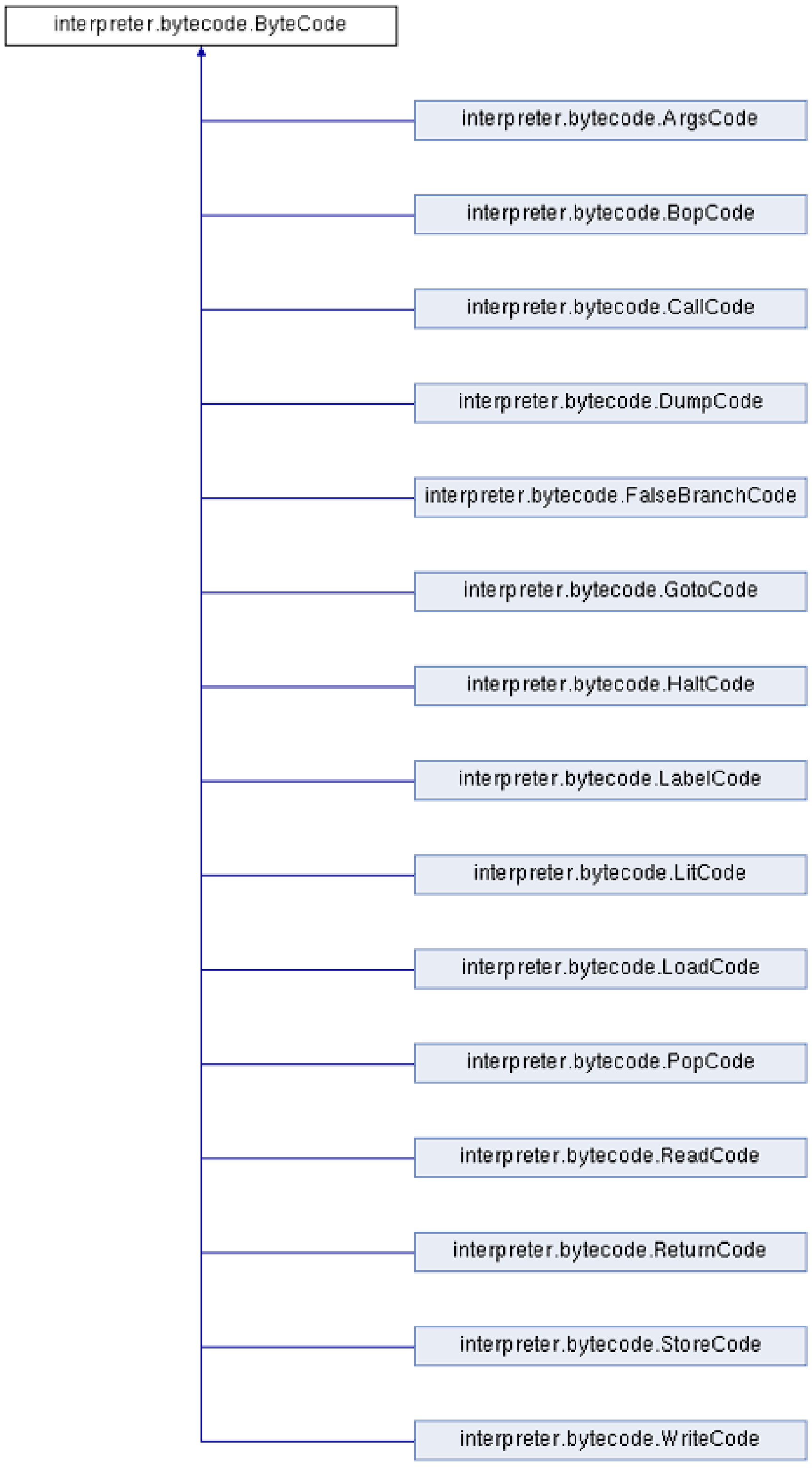
6.1 Class Diagram



```
package interpreter.virtualmachine
package interpreter.bytecode
ByteCodeLoader
CodeTable
Interpreter
```

```
interpreter/
├── bytecode/
│   ├── BranchCode.java
│   ├── ArgsCode.java
│   ├── BopCode.java
│   ├── ByteCode.java
│   ├── CallCode.java
│   ├── DumpCode.java
│   ├── FalseBranchCode.java
│   ├── GotoCode.java
│   ├── HaltCode.java
│   ├── LabelCode.java
│   ├── LitCode.java
│   ├── LoadCode.java
│   ├── PopCode.java
│   ├── ReadCode.java
│   ├── ReturnCode.java
│   ├── StoreCode.java
│   └── WriteCode.java
├── ByteCodeLoader.java
├── CodeTable.java
├── Interpreter.java
├── Program.java
├── RunTimeStack.java
└── VirtualMachine.java
```


interpreter.bytecode.BranchCode
interpreter.bytecode.CallCode
interpreter.bytecode.FalseBranchCode
interpreter.bytecode.GotoCode


interpreter.bytecode.ByteCode
interpreter.bytecode.ArgsCode
interpreter.bytecode.BopCode
interpreter.bytecode.CallCode
interpreter.bytecode.DumpCode
interpreter.bytecode.FalseBranchCode
interpreter.bytecode.GotoCode
interpreter.bytecode.HaltCode
interpreter.bytecode.LabelCode
interpreter.bytecode.LitCode
interpreter.bytecode.LoadCode
interpreter.bytecode.PopCode
interpreter.bytecode.ReadCode
interpreter.bytecode.ReturnCode
interpreter.bytecode.StoreCode
interpreter.bytecode.WriteCode

7 Project Reflection

The 35-page pdf was very overwhelming, and it was scary to take all the information in. Reading through it 3-4 times still do not resolve any confusion. There were lots of requirements and restrictions that were hard to keep track of, and forced me to come back to the pdf just to realize that “oh yeah, I can’t do that”. RunTimeStack was a great place to start, but it’s first method, dump, was probably the harded part of this project. I had to think so far ahead of how it would work in the project as a whole. I had a huge dump method with lots of hard coding, but a classmate mentioned something about runtimestack taking care of the brackets and what not. And I was able to simply my dump method a bunch. My classmate and I had to google how subList can help us here, and we were able to figure it out, and had a similar dump method.

The provided help from the professor, through video lectures, helped out a lot. Especially with the loadCodes in the ByteCodeLoader. That was the first time where I got an actual clear idea of the project structure. The loadCodes shown in the video however gave a lot of trouble when implementing and after debugging, and modifying the program constructure, everything came together nicely.

The 15 subclasses were also overwhelming to approach, but it became simple over time. Figuring out how to dump was the hardest part of this project. Being able to turn on and turn off this function and was a hassle. There was a point where my project would work perfectly fine when running fib.x.cod and would give a correct result, But turning dumping on showed how much I was missing, and that I was working my way around the proper way of reaching the result.

Debugging with the dump on was extremely fun, and it helped me learn a lot more about debugging in general. The project helped me think outside the box and learn to program more conceptually. And made me do some coding on a side of program that I had not touched before.

One thing that would have made this program easier would be a clear explanation of dumping, the availability of factorial.x.dump, and perhaps some more tool such as simple tests that can be done to ensure each step of this massive project is working well before moving on to the next.

8 Project Conclusion/Results

The program successfully implemented an interpreter for the mock language of “X”. The .cod files that can be used as tests to see if the program is complete and working, show the expected results. The program will crash if the input is big, but can run decently well, for example, factorial of 10 should get perfect results.

The project was completed on time for an Extra-Credit, and works perfectly fine, with out breaking any rules such as encapsulation, and maintaining all the requirements.