CSC 413 Project Documentation

Summer 2018

Christopher Rosana

917947321

Table of Contents

1 Introduction

1.1 Project Overview

This program reads a file and performs specific functions based on the text that is on each line in the file. The user is asked to type in an integer value at some point running the program and the program will read that value into a stack. The program manages a stack of values that are separated by functions and works its way through each of them in order to calculate a result.

1.2 Technical Overview

This program will be able to read a source code file that was written in the mock language X. The file will contain byte codes and arguments for various byte codes. The program interprets the file and asks for a nonnegative integer input from the user then calculates the result based on the input.

In order for the program to work, each byte code represents a specific function, if it has one, and the program will parse each byte code to identify how they will function. The program manages a stack and frames to store values that are generated by the various byte codes. These values are then further manipulated to calculate the result.

1.3 Summary of Work Completed

Created an abstract class called ByteCode and created fifteen subclasses that extends ByteCode. Created an interface class called AddressLabel which will be implemented by a select few of the ByteCode subclasses. Implemented the ByteCodeLoader class to parse the program argument file into ByteCode objects and their arguments, if they have any, that correspond to their class name. These objects are stored into an ArrayList that is passed to a Program object. Implemented the Program class to resolve the addresses of ByteCode objects that implement the AddressLabel interface. Doing this will tell these objects what labels to point to when the ByteCode in the program executes. Implemented the RunTimeStack class with various methods, similar to those of Java's Stack class, that will assist in managing the runtime stack. These methods will be used by the VirtualMachine class which will be used by the ByteCode classes. Implemented the VirtualMachine class with various methods that the ByteCodes will use in order to manage the runtime stack. Most of these methods serve as wrappers for the RunTimeStack class methods in order to preserve encapsulation. The ByteCode classes are not allowed to call methods from the RunTimeStack class directly.

Each ByteCode class except for Dump is capable of dumping itself to the console if the dump flag in the VirtualMachine object is set to true. If the program is being dumped, the runtime stack and its frames will also be dumped.

2 Development Environment

Java Version: JDK 11.0.2

IDE: IntelliJ IDEA Ultimate 2018.3.5

3 How to Build/Import your Project

1. Open IntelliJ IDEA
2. Click import project
3. Navigate to the "csc413-p2-blai30" directory and click OK

4. Make sure “Create project from existing sources” is selected and click NEXT
5. Proceed to click NEXT until import is finished
6. Build the project using IntelliJ IDEA IDE by navigating to Build > Build Project from the menu bar

4 How to Run your Project

1. While the project is open in IntelliJ, navigate to Run > Edit Configurations… from the menu bar
2. Click the “+” button in the top left corner of the Run/Debug Configurations window and select Application
3. In the “Name:” text field, type in “factorial”
4. In the “Main class:” text field, type in “interpreter.Interpreter”
5. In the “Program arguments:” text field, type in “factorial.x.cod”
6. Repeat steps 3-5 with “fib” for the name and “fib.x.cod” for the program arguments
7. Click APPLY then OK
8. In the top right corner of the IntelliJ IDEA window, click the dropdown menu and select either “fib” or “factorial”
9. Click the green play button next to the dropdown menu (or press Shift+F10) to run the program with the selected arguments

5 Assumption Made

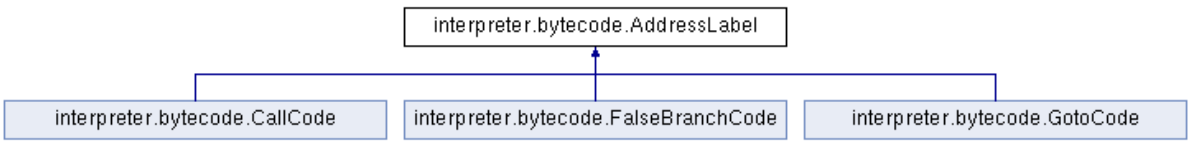
- Overflow and underflow do not have to be managed
- Byte code classes with address resolution need some kind of abstraction or interface
- The given program argument files fib.x.cod and factorial.x.cod contain no errors
- Interpreter.java should not be modified
- CodeTable.java works as is and do not need to be modified
- Program.java may have additional constructors or methods
- The RunTimeStack should not be allowed to pop past any frame boundary
- The RunTimeStack should not be allowed to pop if it is empty
- VirtualMachine.java will contain methods to be used by the execute method in ByteCode objects that simply call methods in RunTimeClass.java
- All values stored in byte code classes cannot be public
- All byte code classes have an init method and an execute method
- Some byte code classes may have empty init or execute methods
- Dumping will print information after the execution of each byte code except DUMP
- Byte code classes cannot call the dump method from RunTimeStack.java
- Each value inside the framePointer stack represents an index in RunTimeStack and these indices indicate the start of each frame in RunTimeStack
- ARGS 0 will create a new empty frame at the end of RunTimeStack

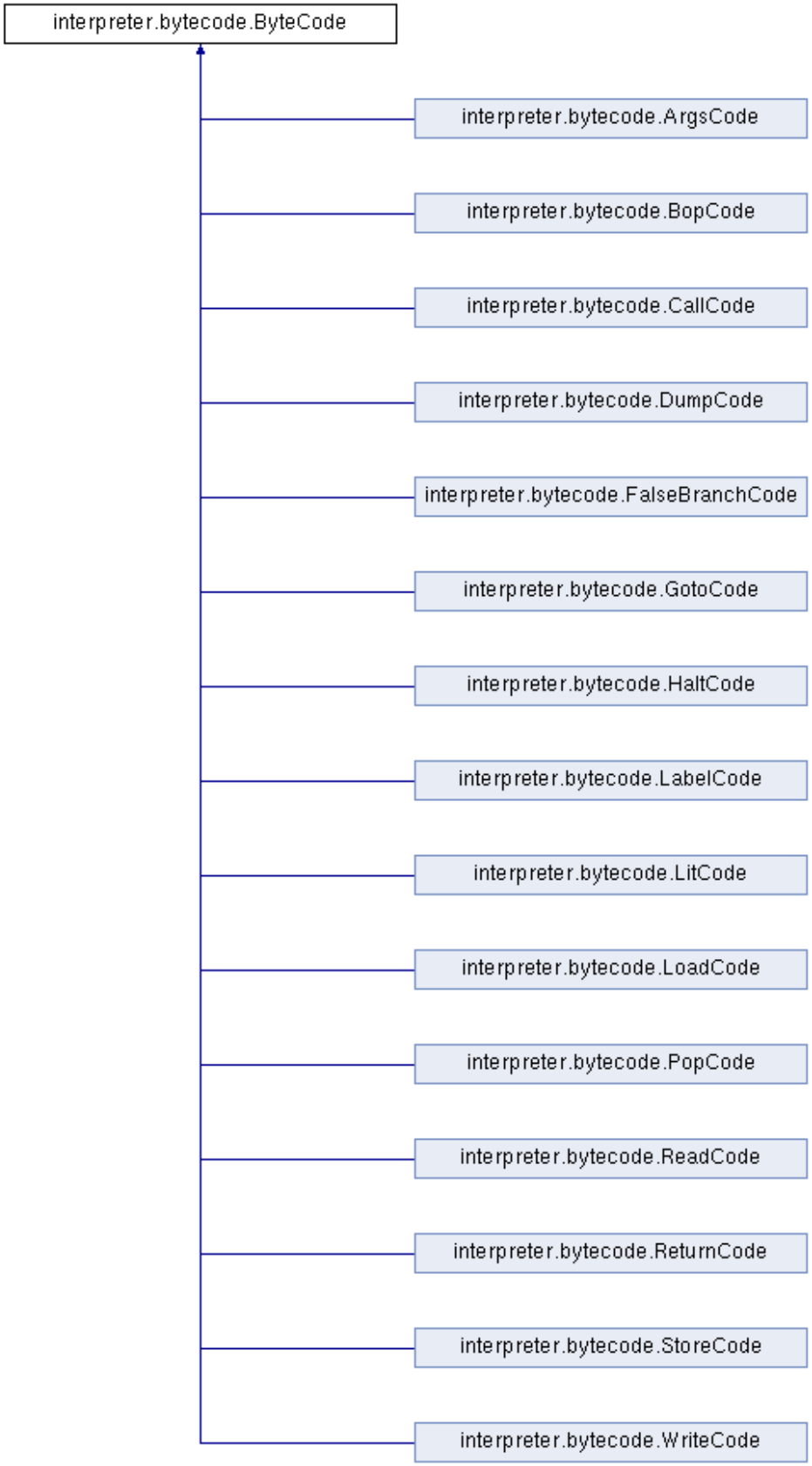
6 Implementation Discussion

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

```
interpreter/
├── bytecode/
│   ├── AddressLabel.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.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

When I was reading the pdf for this assignment, I was overwhelmed by the amount of information I needed to digest. There were a lot of restrictions explained in the pdf but they were all over the place. I was even unsure if byte code classes were allowed to have a dump method of their own for the VirtualMachine to call during execution if the dump flag was turned on because I had read that the byte code classes cannot call dump. I assumed that meant byte code classes cannot call the dump method that was inside of RunTimeStack.java, but they are allowed to have their own dump method by the same name to dump themselves.

When I was programming the loadCodes method in ByteCodeLoader, I initially had created a public method in Program.java called addByteCode that adds the new byte code object to the program ArrayList from inside loadCodes. I decided to get rid of that approach and created a new constructor in Program.java that takes an ArrayList of byte code objects as a parameter which will initialize the program object with an ArrayList of initialized byte code objects. I got this idea from a student named Jonathan Julian so I am writing this to give him credit. He had explained to me that this was much more efficient than the approach had before.

I spent most of my time working in RunTimeStack.java because I wanted to make sure that my RunTimeStack and framePointer stack were being managed correctly during runtime. Specifically the popFrame and store methods. The popFrame method would sometimes not pop every value in the RunTimeStack before pushing the top value back on top even though my program calculates the correct output. The store method would sometimes be trying to store a value to an index that is greater than the size of the RunTimeStack. The main issue with these methods boiled down to off-by-one errors that had to do with the offset or runTimeStack.size not corresponding to the correct index.

My dump function in RunTimeStack.java was pretty simple. I did the project thinking each value inside the framePointer stack represents an index in RunTimeStack and these indices indicate the start of each frame in RunTimeStack. For dumping, I simply printed from one frame pointer to the next, and then from the last frame pointer to the end of RunTimeStack. Using the subList() method, the parameters go from a start index (inclusive) to an end index (exclusive) which made it easier to not have to deal with off by one errors and duplicate values printed.

I liked this project because it helped me learn a lot about programming both technically and conceptually. This project also encouraged a lot of discussion because the pdf was pretty beefy and there were a lot of things the I did not fully understand in the pdf. Also because the professor told us to ask a lot of questions so that we do not feel regret after the project is due.

8 Project Conclusion/Results

Despite the frustration that came out of debugging the program, I definitely learned a lot from this project. I did wish more tools were provided to help us debug and see why we are not getting the correct values and such. I also wished there was a page in the pdf that lists every restriction and recommendation for the assignment since looking through the pdf for the different restrictions left me wondering if I missed any.

Both of my program argument files fib.x.cod and factorial.x.cod return the correct value when I run it. For example when running factorial.x.cod, I would input “5” and the calculated result would come out to

be “120”. For fib.x.cod, I input “9” and receive “34”. With these results I can be confident that my program works good as expected because the two files fib.x.cod and factorial.x.cod should contain no errors. I also noticed that factorial.x.cod has a POP 3 line towards the end but by the time my program gets there, the RunTimeStack would have less than 3 values to be popped. I dealt with this by not letting the RunTimeStack pop any more values if it is empty.