CSC 413 Project Documentation

Spring 2019

Christopher Eckhardt

ID#: 915736372

CSC413.02

https://github.com/csc413-02-spring2019/csc413-02-username006

Table of Contents

1 Introduction

1.1 Project Overview

This project was the implementation of an interpreter for the bytecode generated by a pseudo-language called 'x'. This program needs to have a list of all known bytecodes and compare that list against the bytecode instructions that are given to the program as an argument in the main method. This program contains a virtual machine that will be handling the execution of bytecodes. It controls the order and context required for the program to execute in the correct order and with the correct arguments.

1.2 Technical Overview

This interpreter operates with 4 basic components. The first is the virtual machine which executes the bytecodes in the order they are presented to it by means of the program counter. The runtime stack keeps track of all the variables being used by the program, and the frame pointers which provides a context to their meanings. The program, which is a list of the bytecodes in a predefined order for their given task. The fourth major component is the implementations of the bytecodes themselves contained in their own package.

1.3 Summary of Work Completed

The program functions as it should for the given …x.cod files provided. The dump code, when turned on, provides a visual representation of the executed bytecodes and the current state of the runtime stack.

2 Development Environment

The development environment for this project was as follows;

IDE - IntelliJ IDEA 2018.3.4 (Community Edition)

JRE - 1.8.0_152-release-1343-b26 x86_64

JVM – I used the OpenJDK 64-bit Server VM included with IntelliJ IDEA

OS – macOS Mojave, version 10.14.3 (18D109)

3 How to Build/Import your Project

As the IDE used in this class is Intellij IDEA, I will assume that will be import destination for the project files. Begin by building a "new project from existing sources", when prompted just navigate to the project source and click next. No additional building or 3rd party libraries are required.

4 How to Run your Project

the simplest way to execute the program is to navigate to the Evaluator class and locate the main method. To the left of the method signature, just before the line number should be a green "play triangle". Give that a click and choose to run the method. The program should start in the console below and prompt you for an expression.

5 Assumption Made

I had minimal assumptions going into this project. I didn't understand it enough at the beginning to make any. The only actual assumptions that I could derive were that the instructions were correct and current, and that the provided bytecode files do not contain any errors.

6 Implementation Discussion

This portion of the documentation be about how I implemented the required classes and functions of this program. I began this project by going in the recommended order of completion given in the instruction PDF. I started with the ByteCode Classes. Because it told us to wait until the last step to actually implement, I took a wild stab at trying to cover all eventualities with the parent class. I thought I was going to have to change it later but I got lucky and I was pretty close with my initial guess.

I began by implementing an abstract class called ByteCode. It needed an execute method that is passed an instance of the virtual machine. Next it needed an init function that is passed an ArrayList of the arguments as strings. They can be converted to other data types later if needed. I figured the ByteCodes would need to give information from data fields. From reading the instructions I knew different ByteCodes would have different data types as stored variables, so I Made the getArguments() function return an Object. It's only parameter is the index of that I could use to access a specific argument that code might have. I initially thought having a setArguments() method might have some value, but it turns out the only time it is called is during the resolvAddrs() method. 'toString()' is the only method I had to add or change later. It's only purpose is to pass the appropriate string back to the virtual machine while Dump() is being called.

```
6
7   /*
8   This is the parent class to all byteCodes. It contains all the methods required for the ByteCodes
9   to do their assigned tasks.
10   */
11
12  public abstract class ByteCode {
13
14      public abstract void execute(VirtualMachine virtualMachine);
15
16      public abstract void init(ArrayList<String> arguments);
17
18      public abstract Object getArgument(int index);
19
20      public abstract void setArgument(String arg);
21
22      public abstract String getString();
23
24  }
25
```

Figure 6.0.1

Next I began working on the ByteLoader class's loadCodes() method. This program has four major steps. It reads a line of bytecode and splits that line by spaces. The first index of the resulting array are always the actual bytecode instruction followed by its arguments. The first string is then checked to see if it has a valid bytecode class, then an instance of that class is created. The remaining arguments are then passed in an Arraylist of strings to that ByteCode's init() function. The ByteCode is then added to

the program Arraylist. Before the instance of a Program is returned, All symbolic addresses for the jump instructions (Goto, FalseBranch, and Call) are resolved with resolveAddrs() method.

Resolving addresses was a bit messy. I implemented a working version pretty quickly but I'm sure If I had more time I could find a cleaner solution. My working solution depends on two nested for-loops with an if statement in between. The first loop goes through the program ArrayList. Each bytecode is checked to see if it is an instance of one of the three jump codes. The second for-loop iterates through the program again looking for the matching symbolic argument and replaces the jump code's symbolic address with the destinations index number. Because this only has to occur once when the program is executed, I assumed it wouldnt have a huge effect on performance. This is shown in figure 6.0.2.

```
31  public void resolveAddrs() {
32
33      for(int i = 0; i < program.size(); i++) {
34
35          if( program.get(i) instanceof FalseBranchCode
36                  || program.get(i) instanceof CallCode
37                  || program.get(i) instanceof GotoCode) {
38
39              for(int j = 0; j < program.size(); j++) {
40
41                  if(program.get(j) instanceof LabelCode) {
42
43                      if( ( program.get(j)).getArgument( index: 0).equals( program.get(i).getArgument( index: 0) ) ) {
44                          program.get(i).setArgument(  Integer.toString( program.indexOf( program.get(j) ) ) );
45                      }
46                  }
47              }
48          }
49      }
50  }
```

Figure 6.0.2

It took me getting pretty deep into the RunTime stack and Virtual machine to finally get somewhat of a grasp on how this whole thing works. I decided to keep all of the logic either in the runtime stack or in the bytecodes themselves. This meant that my virtual machine methods would just be intermediaries. In order for the runtime stack to be affected by a ByteCode, that ByteCode would have to ask the virtual machine to then request the RunTime stack to perform the action. This resulted in the virtual machine methods being super simple. The Frames confused me, not conceptually, but rather making sure they were accurately stored in the stack. I thought for sure the ByteCodes were using incorrect logic and causing my problems. It turned out a returned value was being pushed to the runtime stack once in a ByteCode (like it was supposed to) and then being pushed again in the actual runtime stack method. These extra values being pushed to the stack were resulting in a wrong value being printed by the WRITE code a frame too early. I spent a whole night tracing the program step by step and I eventually found the source of the problem. After this I was getting the correct output from the programs. That night of carefully tracing was probably the best thing I could have done. I have a much better understanding of how this interpreter works now.

The ByteCodes were pretty easy to implement. The most complicated part was the Bop Code. I actually used the first assignment as a template for how it could work. I created a separate package within the ByteCode package called BOP. This contains the BOP abstract class and its sub-class operators. The BOP parent class contains a HashMap of all the different operators and Is retrieved with a getBop() method called in the actual BopCode ByteCode class. This is shown below in figure 6.0.3 and

6.0.4.

```
public class BopCode extends ByteCode {

    private String operator;


    @Override
    public void execute(VirtualMachine virtualMachine) {

        BOP bop = BOP.getBOP(operator);
        int operand2 = virtualMachine.popRunTimeStack();
        int operand1 = virtualMachine.popRunTimeStack();
        virtualMachine.pushRunTimeStack( bop.execute(operand1, operand2) );

    }

    @Override
    public void init(ArrayList<String> arguments) { this.operator = arguments.get(0); }

    @Override
    public Object getArgument(int index) { return null; }
```

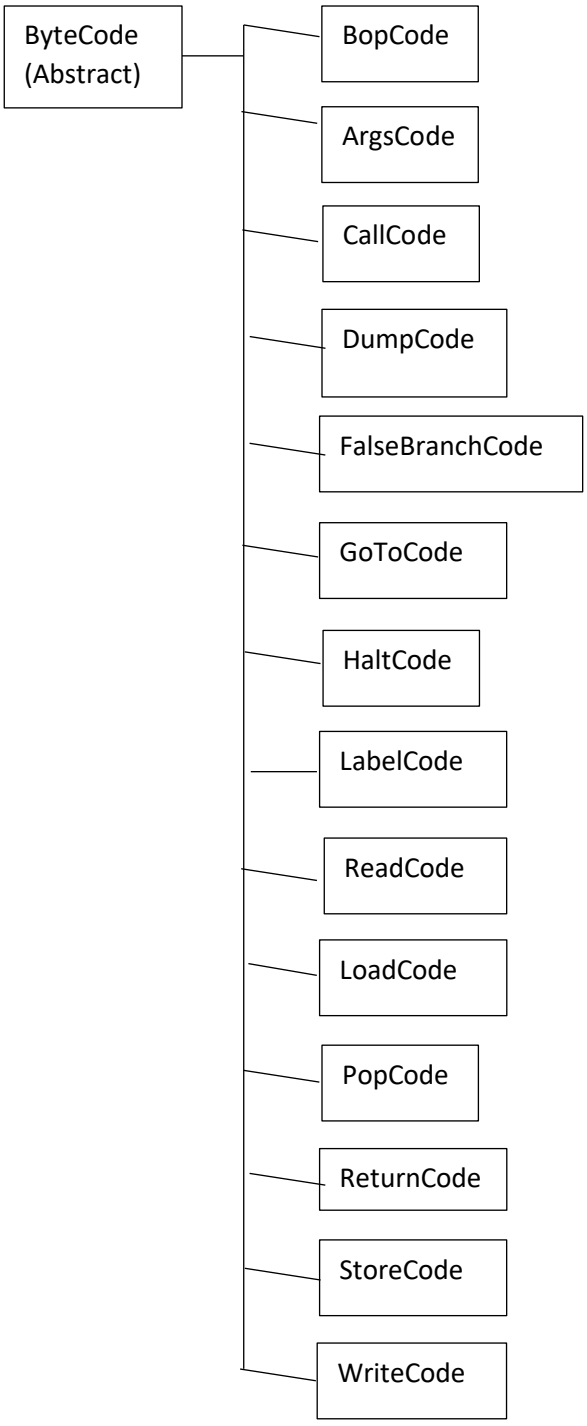
Figure 6.0.3

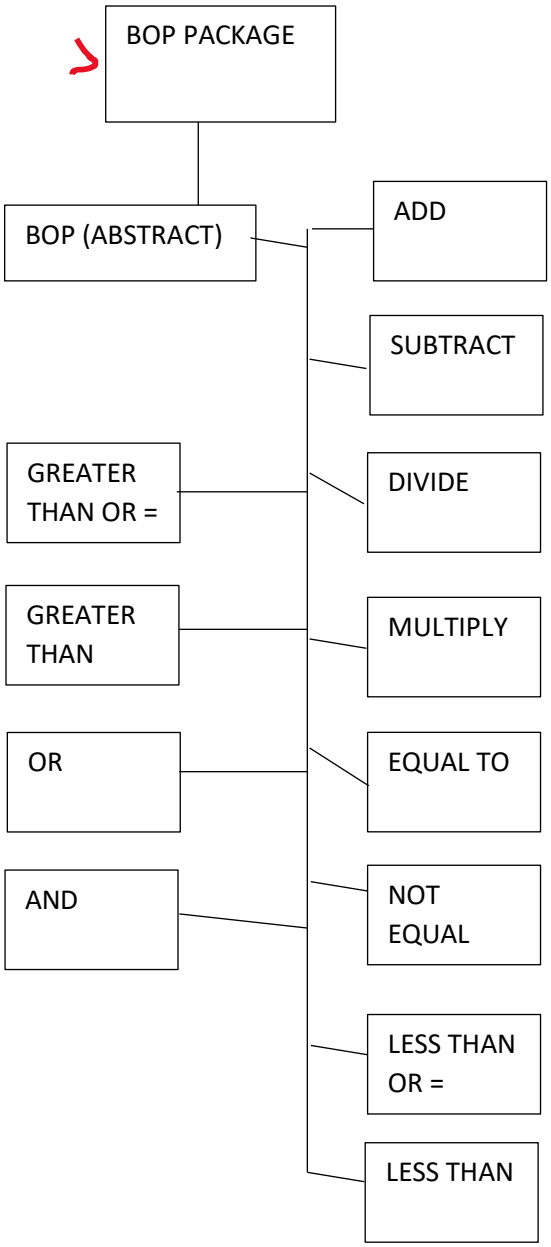
```
public abstract class BOP {

    private static HashMap<String, BOP> opTable;

    static {
        opTable =  new HashMap<>();
        opTable.put( "+", new addOp() );
        opTable.put( "-", new subOp() );
        opTable.put( "*", new multOp() );
        opTable.put( "/", new divOp() );
        opTable.put( "<", new ltOp() );
        opTable.put( ">", new gtOp() );
        opTable.put( ">=", new gteOp() );
        opTable.put( "<=", new lteOp() );
        opTable.put( "==", new equivalentOp() );
        opTable.put( "!=", new notEquivalentOp() );
        opTable.put( "|", new orOp() );
        opTable.put( "&", new andOp() );
    }

    public static BOP getBOP(String arg) { return opTable.get(arg); }
```

Figure 6.0.4

Last but not least is the dump method. This thing was the trickiest part of the whole project for me. Not so much getting it to work, I could manage that just fine, but the logic and code that I used was ugly. I will talk more about the dump method in the project reflection. I have a lot of self-criticism about this part.

6.1 Class Diagram





7 Project Reflection

This project was probably the most complex assignment I've been given here at SF state and after having done it, I can appreciate why it's still given. I learned a heck of a lot from it. The most important lesson I learned was not to dive into coding right away. I started implementing the ByteCodes before I even finished reading the assignment fully and that caused a lot of problems. Another was time management. If I had spent two-thirds of my time planning and one-third actually writing my code I probably would have ended up with a more polished product. A great example of my rush to code was in the dump method.

The first thing that I have to say about my dump method was that I hate it. Not the idea of the dump method, but my implementation of it. I really wish I had spent more time thinking it through. As of right now It contains two Arraylists which are clones of the runtime stack and framePointer stack. This means each time the program counter increments two new Arraylists are created and populated. This is horrifying from an efficiency stand point. I will leave an update at the bottom of this section if I am able to come up with a better solution between now and the due date.

Despite all of the headache I really learned a lot from this project. The moment when I realized that the interpreter working was pretty special. I had been working on the same bug for two days and was very frustrated. I made some small changes and boom! It ran, I jumped out of my chair and did a lap around the room.

UPDATE: I was able to implement dump with only one cloned stack (Frame pointer stack). It's not a huge improvement but it is slightly better than what I had.

8 Project Conclusion/Results

Looking back on this project I am both very proud that I got it to work, but also disappointed that I couldn't fix all of my mistakes before the due date. There are a lot of checks in my runtime stack class that I should have included, and some methods (dump) are very messy and in efficient. I also wish I had placed a check inside of the pop method in my runtime stack that would make sure it doesn't pop past a frame boundary. However, It was a great learning experience, I love doing big and complicated projects like this and can't wait for the next.