Deepinder Bhuller

Student ID: 911770878

CSC 413.03 Spring 2019

https://github.com/csc415-03-spring2019/csc413-p2-rogue-7

Project 2:

Interpreter

Introduction

Project Overview

The basic idea of project two was to build and implement an interpreter for a mock programming language X. There are a number of parts of the Interpreter project that must be implemented to work successfully. The parts of the project are the bytecode folder, the VirtualMachine, RunTimeStack, ByteCodeLoader, Interpreter, CodeTable, and Program classes. The bytecode folder contains all the bytecodes that result from processing the language X files. When a file is passed into the Interpreter, the file is processed by the Interpreter and run in the Virtual Machine. The main requirements of this project were to **(1)** Implement all the ByteCode classes with the correct abstractions, **(2)** Complete the implementation of the following: ByteCodeLoader.java, Program.Java, RunTimeStack.java, and VirtualMachine.java, **(3)** Make sure all variables have the correct modifiers, **(4)** Make sure not to break encapsulation under any circumstances. The CodeTable and Interpreter classes have already been implemented.

Technical Overview

The Interpreter Project is an implementation of a mock programming language X. The Interpreter class is the entry point for this project and is where main() is located. When a file is passed into the interpreter, it is read line by line and bytecode instances are generated. This is accomplished through keeping track of the Frames and the Runtime Stack. Frames are the set of variables(arguments, local vars, and temp storage) for each function called during the execution of a program. A stack is the best way to implement this because the function calls and returns happen in LIFO order. For example consider the following execution:

```
Main() {
        A();
                B();
}
```

The corresponding runtime stack would would have Main() at the bottom, A() next, and B() at the top of the stack. First B() would be executed, then returned to A(), then A() is executed and returns back to main(), which then completes execution. Taking this concept, when applied to a sample Language X program, each function call represents another frame.

There are about 15 bytecodes, some examples are HALT, DUMP, READ, WRITE, ect. All of these codes can be abstracted to a general ByteCode class. Another abstraction can be made with some of the codes branch to other parts of the program, this abstract class is BranchCode and it extends CallCode, FalseBranchCode and GoToCode. These abstractions provide an efficient way to implement each bytecode and to help the program recognize the correct one. Each bytecode has an init function and a execute function. These two functions are essential to the function of the program because the init function defines the bytecode when the interpreter comes across it in a file, and the execute function defines the action the virtual machine takes with each corresponding bytecode. Each class of bytecode does not contain the logic to be executed, instead each bytecode requests the virtual machine to perform the action for the bytecode loaded from the code found in the .x file. Some codes such as ReadCode, WriteCode, and HaltCode’s init functions don't have any statements because nothing needs to be initialized when for their specific codes.

The next part of the project is the Runtime Stack and RunTimeStack class. The Runtime Stack records and processes the stack of frames generated by the function calls of the .x file passed into the program. The RunTimeStack uses a Stack<Integer> framePointer and ArrayList<Integer> runTimeStack. The framePointer stack records the beginning of each frame when calling functions. The runTimeStack ArrayList is used to to represent the runtime stack because all locations of the stack need to be accessed. The class RunTimeStack also maintains the active frames, wo when a function is called in the passed in .x file, a new frame is pushed to the stack. When there is a return from a function, the frame is popped off the stack. The functions that make the class work are the constructor, the dump(), peek(), pop(), newFrameAt(int offset), popFrame(), store(), load(int offset), load(int offset), push(Integer val). The dump() is very important is is responsible for dumping the current state of the RunTimeStack. When the stack is printed, it should include divisions between frames and if a frame is empty it must be shown as well. The peek() returns the top of the stack without removing it. The pop() removes an item from the top and returns it. The newFramAt(int offset) creates a new frame in the RunTimeStack class and the offset is used to denote how many slots down from the top of the stack a new frame begins. The popFrame() pops the op frame when returning from a function. Before the opo, the functions return value is at the top of the stack so the value is saved and then the top frame is pop[ed and the return is pushed back onto the stack. It is assumed the return values are at the top of the stack. The store(int offset) stores the values into variables and the function will pop the top value of the stack and replace the value at the passed in offset in the same frame, the value is then returned. Load(int offset) loads variables onto the RuntimeStack from the passed in offset within the same frame. This means that the function goes to the offset in the current frame, copies the value and pushes it to the top of the stack, no values are removed with load(). Push(Integer val) returns an Integer object and is used to load a liter value onto the RunTimeStack(ie: LIT 5 or LIT 0 will cal push with the value being 5 or the value being 0).

Another part of the project is the Program class. This class is responsible for storing all the bytecodes read from the .x source file. The bytecodes are stored in an ArrayList<ByteCode>. This ensures only bytecodes and its subclasses can be added to this ArrayList. There is also a HashMap<String, Integer> that is in the constructor for the class. The class has functions getCode(int pc), program() constructor, getSize(), addCode(ByteCode code), getProgram(), and resolveAddrs(). The getCode(int pc) function returns the ByteCode at a given index, and the resolveAddrs() function resolves all symbolic address in the program.

The CodeTable class is used by the ByteCodeLoader class and stores a HashMap which allows us to have a mapping between bytecodes as they appear in in the source code file and their classes in the interpreter.bytecode package. An example is (“HALT, “HaltCode”), where HALT is the bytecode in the file and HaltCode is the class that defines it. These are populated through an init() method. The CodeTable is used by the ByteCodeLoader class's method loadCode() to get an instance of each bytecode with CodeTable's hashmap.

The next piece of the project is the VirtualMachine class. This class is used to execute the given program. This class is the controller for the program and all operations go through it. It contains a RunTimeStack runStack object, int pc, Stack returnAddrs, Boolean isRunning, and Program program. The RunTimeStack runStack object is the RunTimeStack. The int pc is a program counter that keeps track of the current bytecode being executed. The Stack returnAddrs is used to store the return address for each called function, except main. The boolean isRunning is used to determine whether the VM should be executing bytecodes. The Program program object is a reference to where all the bytecodes are stored. The VirtualMachine has many functions that I implemented. The bytecode classes request the VirtualMachine to execute their function. This means the ByteCode class completes the action by requesting the VirtualMachine to execute the bytecodes function through the VirtualMachine's methods. The returnAddrs stack stores the bytecode index(PC) that the virtual machine should execute when the current function exits. Each time a function is entered the PC is pushed onto the returnAddrs stack. When a function exits the PC is restored to the value that is popped from the top of the returnAddrs stack. One important function of the VirtualMachine is the executeProgram().

The Interpreter class is the entry point for the project and is unchanged. This class performs all initialization, loads the bytecodes from a file and runs the virtual machine.

Work Completed

First I created all the classes that defined the bytecodes. Each bytecode was implemented as a class that extends an abstract class ByteCode. First I had to read through the documentation provided to understand what each code did. Each class would extend ByteCode. At first I had all classes of bytecodes extending ByteCode. I realized another abstraction could be made by having another abstract class called BranchCode that would extend ByteCode, making it a subclass that extends the codes that perform branching operations. First I focused on implementing the init() functions for each bytecode. From the documentation I gathered that they must be initialized with passing in an ArrayList<String> args. Each subclass init method is implemented based on the description provided in the documentation. The BranchCode subclass implemented a few more methods than the ByteCodes. The ByteCodes also all had a method called execute(). This method was responsible for requesting the virtual machine to execute the function of the bytecode read from the file. Each execute() works through the virtual machine.

Next, I implemented the ByteCodeLoader class. The two important methods here were loadCodes(). The loadCodes() reads one line of source code at a time. For each line it breaks it into tokens and then grabs the correct class name for for the given ByteCode from the CodeTable class. Then, an instance of the bytecode class name is returned from the code table. Any additional arguments are parsed in for the bytecode and then sent to the new ByteCode instance through an init function. First I created a few string values to keep track of the “key” and “value” to come from the tokens. I used an ArrayList to keep hold of the tokens from the lines read. I used a try{} and catch{} to catch any exceptions that could be thrown by the method, particularly ClassNotFoundException for using the Class.forName() and InstantiationException for using the .newInstance(). I struggled with this method, specifically with the exceptions and the logic. I used a string tokenizer to read the lines, then check if there was another token and added the token to the ArrayList.

Next I implemented the Program class. In this class I implemented the Constructor with an ArrayList<ByteCode> and a HashMap<String, Integer>. I implemented a protected ByteCode getCode(int pc) {} method. This method returned *this*.program.get(pc). Next I implemented the public void addCode(ByteCode code) {} method. This method was responsible to identify if a code was a LabelCode and then store the LableCode in a hashmap. Next I implemented the resolveAddrs() method. This function goes through the program and resolves all addresses. It is done by resolving all branch codes to Integers which are stored as addresses that are branched to.

Next I implemented the RunTimeStack class. This class was defined by an ArrayList<Integer> runTimeStack and Stack<Integer> framePointer. These two data structures kept track of the frames and the function calls. I implemented many functions to get this working. One function was the dump(). This method was implemented with an Iterator. It prints the RunTimeStack one frame at a time. If the Iterator hasNext(), then the nextFrame is the cast of class(Integer) of iterator.next(), else the nextFrame is the runTimeStack.size(). Next, the method prints the current frames. I implemented a peek() that returns the runTimeStack's size minus 1. I implemented a pop() method that defines an int toBePopped to the runTimeStack.size()-1. I also implemented a madToPop() that returns the size used by a function in the RunTimeStack. There are other methods such as push(int toBePushed), store(int offset), load(int offset) that were also implemented to make the RunTImeStack class work correctly.

Next I implemented the VirtualMachine class. This class also contained many functions. This class contained an important method called executeProgram(). This method was responsible for getting each bytecode from the program and execute them according the each bytecode execute function. This functions checks if there is a dump in the program and if there is, it prints out the stack. Next I implemented the functions that complete the VirtualMachine. All of the functions were similar to the functions in the RunTimeStack. These functions all call a runtimestack object, runStack in their function bodies. This means the VirtualMachine acts on the RunTimeStack and not on any bytecode alone.

After all of this, I went back to all my bytecode classes that I had implemented and filled out the remaining execute functions that needed the VirtualMachine class to work. This meant making all the execute functions specific to the classes they were defined in.

Development Environment

Version of Java Used

java version "1.8.0_181"

Java(TM) SE Runtime Environment (build 1.8.0_181-b13)

Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)

IDE Used

IntelliJ IDEA 2018.3.5 (Ultimate Edition)

Build #IU-183.5912.21, built on February 26, 2019

Licensed to Deepinder Bhuller

Subscription is active until February 5, 2020

For educational use only.

JRE: 1.8.0_152-release-1343-b28 amd64

JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

Windows 10 10.0

How To Build & Run

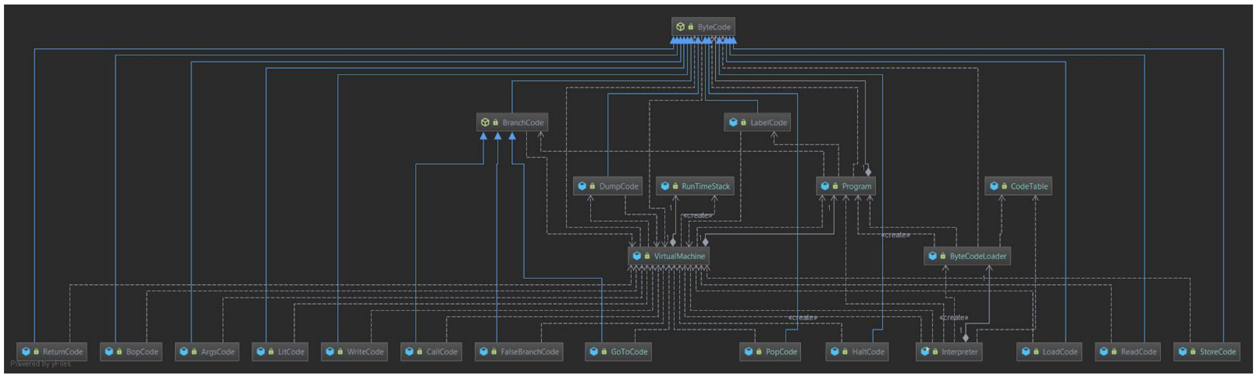
The way to import and build my project is to pull the project from github. Upon importing using git pull <project url>, in the IDE, you must set up the test configurations. To do this, click the dropdown menu → Edit Configuration → Program Arguments. From here change the main class to interpreter.Interpreter to set the main method to be run. Under

Program Arguments you would specify the .x.cod file that should be passed into the Interpreter project. The program can be compile in the command line and by clicking the play button in the IDE.

Assumptions

Assumptions in this project are as follows. We assume all given bytecode source program(.cod) files are generated correctly and contain no errors. We also assume that the Interpreter.java file and CodeTable.java file are given to us without any errors.

Implementation Discussion



The above diagram is a UML of the hierarchy of the Interpreter project. This shows that the root is the Interpreter Project, one level below is the VirtualMachine and ByteCodeLoader classes. Below that is the RunTimeStack, Program, and CodeTable classes. The final level of the hierarchy tree is the .bytecode package which contains all the ByteCode and BranchCode classes. The diagram shows all the bytecodes extending the ByteCode and BranchCode classes.

When implementing this project I had to keep encapsulation, inheritance and abstraction in mind. I had to build the project in a way that all bytecodes worked through the VirtualMachine and didn’t execute codes through the individual bytecode classes. Inheritance was important because at first i didn't have bytecodes implementing BranchCode, just ByteCode. This made it easier to generalize a group of bytecodes. Overall the hierarchy and structure of this program was straightforward because it was easy to seperate the ByteCodes with an abstraction.

Project Reflection

Overall I did not enjoy this project. From the beginning I faced issues importing the project and getting it to build correctly in my IDE. At first, I didn’t have any code completion or syntax checking and it made this project extra difficult. Aside from these issues, the Documentation was very difficult to digest and get through. Many details are hidden in the documentation and aren't obvious.

I wasn’t very familiar with how a RunTimeStack worked or how a VirtualMachine functioned and I am still not sure. I have a basic understanding of how these things function and an even more basic understanding of how they work together to parse and interpret the file’s passed in. For my tests performed on the fib.x.cod and factorial.x.cod, I am not sure I have the correct output or even have the file being passed into my program correctly. I did not enjoy this project because I found creating the ByteCodes and implementing all the classes and functions very tedious. Although this project was difficult and I don't think i did well on it, I learned a lot and was able to get a somewhat working project. I feel like if more time was provided I could have taken more time to keep debugging the project and testing the many edge cases that come up.

Conclusion

The result of my project is not what I expected. I don't think my dump() prints the stack correctly and I also think there are errors in my VirtualMachine and ByteCodeLoader. The output I get for the fib.x.cod configuration when I run it is being prompted to enter an input. Upon entering the input, the function returns a value 3. When I run the configuration for factorial.x.cod, the same prompt appears to enter an input, this time after hitting enter, the value returned is always 1. I copy and pasted the bytecode file from the documentation into a .x.cod file and ran it resulting in many errors. I think I need to test for edge cases to get this program to work and read the file correctly. I tried my best on this project and gave it my best shot. I want to do more research and study more on building interpreters and compilers and attempt it again in the future.