Interpreter

CSC 413 Project Documentation

Spring 2019

Stephanie Sechrist

918679078

413.02

https://github.com/csc413-02-spring2019/csc413-02-stephlsechrist.git

Table of Contents

1 Introduction

1.1 Project Overview

For this project, I was asked to implement an interpreter for a made-up language X; the interpreter processes byte codes produced from source code written in X to run the X program. The interpreter works with an implemented virtual machine to do this. The project uses stacks, HashMaps, and ArrayLists.

1.2 Technical Overview

This program begins in the Interpreter. Additionally, there are 5 classes (VirtualMachine, RunTimeStack, Program, ByteCodeLoader, and CodeTable) and one package of bytecodes, which contains 16 classes including the abstract ByteCode. In section 6.1, I discuss the implementation in detail, but the overall flow of the program is as follows: Interpreter creates CodeTable and VirtualMachine and also has a ByteCodeLoader. It instantiates a Program and calls the ByteCodeLoader to create a Program using a HashMap in CodeTable. The Program has byte codes and is returned to Interpreter. The VirtualMachine has a Program and also creates a RunTimeStack. RunTimeStack does not have any relationship with the byte codes and is only accessed by VirtualMachine (VM from here on). The Program object created by ByteCodeLoader is controlled by VM via Interpreter; Interpreter passes the Program object to the VM when it creates the VM. vm.ExecuteProgram() is called by Interpreter, and in VM's executeProgram() is where we go through Program and execute the byte codes by performing operations on VM's runTimeStack runStack.

1.3 Summary of Work Completed

For this project, Interpreter was provided and not allowed to be altered. CodeTable was also provided, and I did not make any changes to it. Some code was provided in ByteCodeLoader, Program, RunTimeStack, and VirtualMachine, but I completed the rest. Finally, all byte codes were created by me. Below is a more detailed list of added classes and methods:

In ByteCodeLoader:
- implemented all of loadCodes()

In Program:
- implemented getSize(), which I ended up not needing but left in anyway in case of future updates
- implemented resolveAddrs()
- implemented addCode()

In RunTimeStack:
- implemented all methods except the constructor

In VirtualMachine:
- implemented executeProgram() from base code provided by assignment PDF; added if statement for when dump is on
- implemented everything else except the constructor

2 Development Environment

For this project, I worked solely in IntelliJ IDEA 2018.3.4 (Ultimate Edition) on Windows 10. The version of Java being used is 11.0.2.

3 How to Build/Import your Project

1. Use given git repository link to clone the project into desired folder.
2. Open IntelliJ IDEA and select Import Project in the Welcome window.
3. Find the folder that you cloned the git repository to and open the project folder (should be named csc413-02-stephlsechrist).
4. Highlight the csc413-02-stephlsechrist folder to be the root of the source files. Click OK.
5. Select “Create project from existing sources” and click Next.
6. Accept the defaults for naming by clicking Next.
7. Accept the defaults for default root by clicking Next.
8. Accept the default libraries (none) by clicking Next.
9. Accept the default module (should be csc413-02-stephlsechrist folder) by clicking next.
10. Select project SDK. You should be using JDK 11. In Name field, type 11. If not already found for you, find your JDK home path. Click Next.
11. Next page should say “No frameworks detected.” Click Finish.

4 How to Run your Project

There are two included source codes to run: factorial.x.cod and fib.x.cod. To run these, you must configure the Interpreter class (which has the program’s main method).

1. Right click on Interpreter and select **Run ‘Interpreter.main()’**
2. You should see in the dialog box, ***Incorrect usage, try: java interpreter.Interpreter <file>
3. Above editor, click the drop-down menu next to the Build Project icon (green hammer) that should say **Interpreter**. From this menu, select **Edit Configurations…**
4. Under Application > Configuration, enter the source code name you would like to run in **Program Arguments** box.
5. Click **Apply** and **OK**.
6. Now you can run the project again.

If you have Java installed, you can run the program from the command line using one of the source codes as an argument and Interpreter as the program.

5 Assumptions Made

In this project, it is assumed that the source code provided only contains accurate byte codes listed in the code table and correct arguments in correct order. It is also assumed that the user will not try to use BOP to divide by zero, so that test case is not accounted for. We also assume that all values on the run time stack are ints. I also assume that the user will not try to crunch giant numbers, as int overflow is not required to be addressed in this assignment.

6 Implementation Discussion

6.1 Discussion

For this project, the structure and flow were already determined for us, and we had to fill in the gaps to implement what was required.

For this project, it is easier to explain the structure from the byte codes all the way up to the Interpreter.

Each byte code is a child class of ByteCode, all contained in the same package. Not all these byte codes have the same data fields or methods, so there is a separate child class for each one. However, all byte codes have an init() function and an execute() function. Init() is called when initializing the instance of the byte code; here is where we set any private data fields if any, such as labels, variable names, or addresses.

Upon Interpreter's request, these byte codes are initialized in ByteCodeLoader, which is also where the source file is parsed. The file is read with a BufferedReader line-by-line. For each line, a StringTokenizer is used to tokenize the string into parts delimited by a space. The first token is the byte code. Java reflection is used to build instances of classes corresponding to the bytecodes during runtime. In addition, if the string has more tokens, they are added to an ArrayList<String> and passed into the respective byte code's init() function, where the arguments are assigned to private data fields. Back in ByteCodeLoader, the byte code just instantiated is added to an ArrayList of byte codes called program.

These are added via the Program class, which contains the private ArrayList, program; in other words, the Program class stores all the bytecodes instantiated from the source file in program. Also in Program is the resolveAddrs() function that is in charge of finding the addresses for branches (FALSEBRANCH, GOTO, CALL) to jump to and assigning that address to each branch function. These addresses are designated by the byte code LABEL. I use a for loop to go through program and find any LABEL byte codes and look up each LabelCode instance's private data field label using getLabel(). A HashMap is used to store all the labels and corresponding addresses. In another for loop, I go through program, and if I encounter any aforementioned branch byte code, I look for its label in the HashMap. If the label is found, the branch byte code gets the corresponding address saved to private data field branchAddr via the setBranchAddr() function in each branch's class. I use a HashMap, because I know I will want to search and insert often, and the average case for these is O(1); the worst case is O(n).

This resolveAddrs() function is called by ByteCodeLoader to update branchAddr on program's branch byte codes. ByteCodeLoader passes the Program object back to VirtualMachine. VirtualMachine oversees Program and executes the byte codes. To go through the byte codes, we have int pc, which keeps track of the address to be executed. Since the byte codes were stored in order in ByteCodeLoader, the index of the byte code in the Program's ArrayList<ByteCode> is its address. While the boolean isRunning is true, a while loop executes byte codes. isRunning is set to true until the byte code HALT turns it false; then, the VirtualMachine stops executing byte codes from program. If dumpState is true (only possible if DUMP ON is in the source file), the byte code and its arguments are displayed to the console, as well as the current state of the runStack. At the end of the while loop, I increment pc.

Apart from incrementing, pc can be changed by CALL, RETURN, FALSEBRANCH, and GOTO. VM has a Stack returnAddrs that these byte codes can change. When CALL is executed, it requests the VM to get the current pc, push it to returnAddrs, and set the pc to the branchAddr associated with the byte code via VM's getPC(), pushReturnAddrs(), and setPC() functions. When RETURN is executed, it requests the VM to pop returnAddrs and set pc to the value popped via VM's popReturnAddrs() and setPC. It also requests the VM to pop the top frame of VM's runStack via popFrameRunStack().Function scopes are delineated by frames in this implementation, which will be discussed later.

VM's runStack is controlled, manipulated, and defined in RunTimeStack class (RTS from here on). Here, we have functions to manipulate the RTS. As described previously, to preserve encapsulation, the only class able to access RTS is VM, which is why there is a function in VM corresponding to each stack

function that the byte code might need. In RTS, each function prevents any manipulation if it is not allowed. For example, the user is not allowed to pop the stack if the stack or the top frame of the stack is empty. These functions are called by the VirtualMachine, which is in turn requested by the byte code to perform operations on the run time stack or change the address being executed. Once all the byte codes are executed and HALT is called, the VirtualMachine stops executing. Control goes back to Interpreter, and the program finishes.

VirtualMachine has an RTS called runStack that keeps track of values in the byte code program. Within the RTS, there are frames, and each frame is its own scope. The locations of frames are stored in RTS in an Integer Stack called framePointer. framePointer keeps track of the indices of the runTimeStack under which a frame occurs. Frames are added when ARGS n is called, n being the offset from the top of the frame. For example, if ARGS 1 is called and the size of the runTimeStack is 8, framePointer will have 7 at the top. Because these frame boundaries represent the scope of a function, if the frame is empty, the RTS does not allow any requests from VM that would involve popping values past the boundary. Byte codes that would possibly do this are BOP, FALSEBRANCH, LOAD, POP, STORE, and WRITE. When RETURN is executed, it asks the VM to pop the top frame of runStack via RTS. This is when we exit a function scope and return the value associated with the scope back to the calling function. popFrame() in RTS pops the top value, saves it in a temporary variable, pops all the other values until the frame is empty, pops the frame boundary index off framePointer, and pushes the saved value onto the RTS in the now top frame. I also have a function in RTS called peekFrame() that returns a string of the values in the top frame, which is used by CALL when dumping is on.

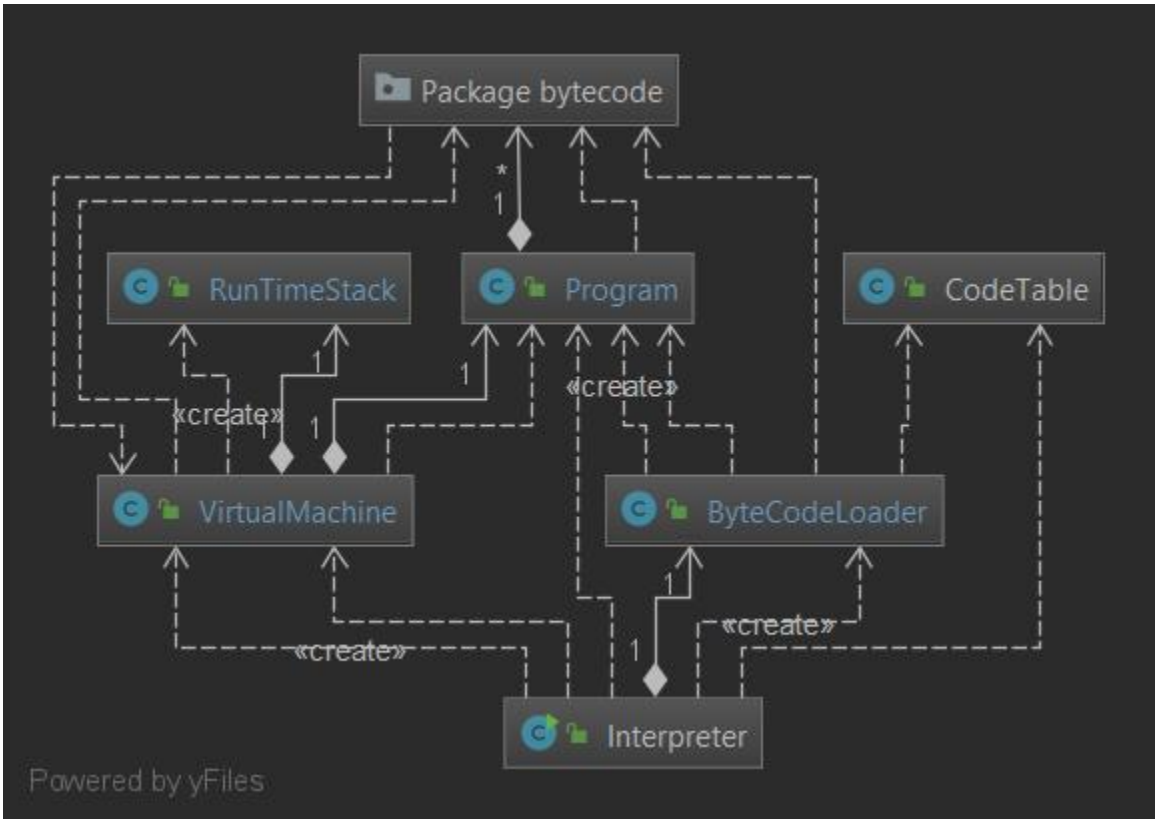
One last aspect to focus on is the dumping program state. As touched upon earlier, there is a boolean in the VM called dumpState. It is initialized to false and can only be changed using the byte code DUMP. In the source code, DUMP ON or DUMP OFF may appear. When ON, DumpCode requests the VM to switch the dumpState boolean to true via VM's setDumpState() function. Now, after executing a byte code, the VM will print the byte code and its arguments (labels, values, variables) to the command line. Each byte code has its own printBC() function, because they do not all have the same private data fields; furthermore, since the data fields are private, they may not be accessed by VM unless through getters. However, it is only necessary for LABEL and the branch byte codes to have a getter for the label. All the byte codes are printed as they appear in the source code except for CALL. For CALL, it was required to print the function being called without < or > brackets and the arguments being passed to the function. For example, in the source code, one would see CALL factorial<<2>>; in dumping output, one would see CALL factorial factorial(3). In this example, CALL is passing 3 into the function. To display the arguments being passed, RTS's peekFrame() is used as previously described.

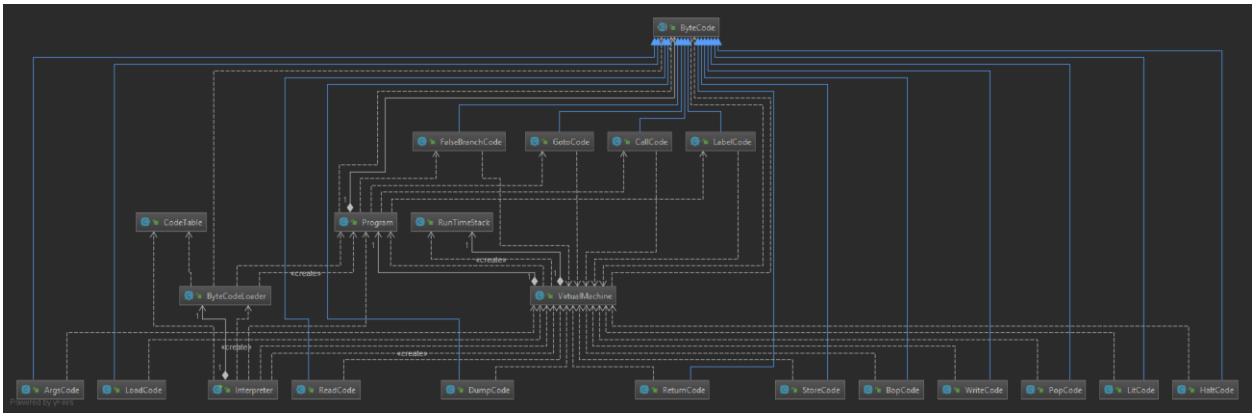
After each byte code is displayed, the current runStack is displayed by the VM by requesting RTS's dump() function. Here, runTimeStack is cloned into an ArrayList clonedStack. Then I have another ArrayList called holdArray that will be populated with ArrayLists that represent each frame in the stack; it is initialized to the same size as framePointer. In this way, the stack can be printed to the command line with frame boundaries intact. Frames are represented by square brackets. For example, [] [7, 7, 6] shows a runTimeStack of size 3, with frame pointers at indices 0 and 1. To fill up holdArray, a nested for loop is used. The outer for loop gets holdArray ready to fill the index matching the top of framePointer; the inner for loop adds values in order to the ArrayList in holdArray at the mentioned index. The nested for loop continues this for as many frames exist. Finally, a for loop prints each ArrayList in holdArray

using toString(). toString() presents the ArrayLists in the manner required by the assignment, with square brackets and commas.

6.2 Class Diagram

The class diagram shown below is to show how the class relationships exist. The Package bytecode is expanded in the second diagram. It is hard to see the details, so please look at UML.png that is in the documentation folder alongside this PDF. As shown in the UML the following are child classes of ByteCode: HaltCode, PopCode, FalseBranchCode, GotoCode, StoreCode, LoadCode, LitCode, ArgsCode, CallCode, ReturnCode, BopCode, ReadCode, WriteCode, LabelCode, and DumpCode. These classes also have a dependency relationship with VirtualMachine, indicating changes to the byte code classes may cause changes to the VirtualMachine. There is also a dependency relationship between Program and the byte codes LabelCode, FalseBranchCode, GotoCode, and CallCode because of the resolveAddrs() method; any changes to the Program class may change the listed byte codes. Program has ByteCode as part of it, so we see a composition association relationship, in addition to a dependency. Interpreter has a dependency relationship with CodeTable, ByteCodeLoader, Program, RunTimeStack, and VirtualMachine (also, Interpreter creates VirtualMachine). Furthermore, Interpreter creates a ByteCodeLoader and a VirtualMachine. Interpreter also has a composition association relationship with ByteCodeLoader; a ByteCodeLoader is a part of an Interpreter, and if the Interpreter object is destroyed, so is the ByteCodeLoader. ByteCodeLoader has a dependency relationship with CodeTable and Program. VirtualMachine has a dependency with RunTimeStack and creates it; additionally, runTimeStack is a part of VirtualMachine, as indicated by the composition association relationship. Virtual Machine also has a dependency and composition association relationship with Program.





7 Project Reflection

This project was slow-going at first, but once I understood all the moving parts, it was pretty smooth. It took a while to get through the PDF and really understand how each object interacted with each other, as well as what was in control of what. The VirtualMachine class was a little tedious to write, because I basically wrote a function corresponding to every runTimeStack function; however, all the other functions were fun! This project really came to life once I understood how each piece fit together. It was a great lesson in encapsulation, and I think I implemented a design that did not break encapsulation. Overall, the project took over my life these past two weeks, and I am ready to move on from it, but again, it was a great learning experience. I think this is the first program I have written with this many different data structures attached, so I really feel like my knowledge was tested.

8 Project Conclusion/Results

I am a little disappointed that I could not figure out how to dump the run time stack with proper frame boundaries without getting help, but my collaborator had a great algorithm that I could not find a way to improve on. If I had more time, I would address this first off, because using two ArrayLists seems inefficient, especially because one ArrayList is made up of ArrayLists. There must be a way to do it with just a print line, but I was not able to get it to work. I am also sure I could probably find a more efficient way to deal with the byte codes. There was talk in the PDF and on slack about finding what byte codes had things in common and using that to prevent duplicate code, but I ended up executing each byte code in its own class. I would need to think a little more about how to cut down on my lines of code. Perhaps I could add a method in RTS that checks the status of the stack anytime changes are requested, rather than doing a check in the other methods of RTS.

To summarize, if I were to update this project in the future, I would focus on streamlining. This project was designed, I believe, to force us to really think about object-oriented design in terms of encapsulation and efficiency. I think I had a good grasp on how to implement this project, and I am happy with my results. The program works as expected, and I believe most of what I did was not inefficient and did not break encapsulation. There is some tight coupling going on, especially between the VM and the RTS classes, but I feel in this case it was unavoidable. They had to work together.