CSC 413 Project Documentation

Fall 2018

RUSSELL WONG

916507662

413.01

https://github.com/csc413-02-fa18/csc413-p2-russ3llwong

Table of Contents

1 Introduction

1.1 Project Overview

This project is a simple interpreter for the “mock language X”. In this context, the mock language is kind of like a simplified version of Java language, and it will only have byte codes and their arguments if any. However, this simple interpreter only interprets 15 types of byte codes.

The interpreter will process these byte codes that will be created from source code files with the extension x. The interpreter and the Virtual Machine work together to process a program/file written in the language X mentioned. The Virtual Machine is like the main control room which does most of the work to process and execute the byte codes while managing a runtime stack class.

The project will be used to run two recursive programs written in files with extension *x.cod*. These programs are a recursive version of computing the nth Fibonacci number and recursively finding the factorial of a number.

1.2 Technical Overview

As mentioned earlier, this project is a simple interpreter for the “mock language X”. The *main()* function is in the *Interpreter* class. The *Interpreter* class constructed will first use the *ByteCodeLoader* class to load the byte codes in a source code file, with the help of the *CodeTable* class, which is needed to identify the individual byte codes. These byte code objects, after processed, will then be stored in a *Program* object.

After that, a *Virtual Machine* object will be created to process the *program* file, which contains the byte codes from the source code file. The Virtual Machine will keep track of the file’s index as it executes the byte codes. It will manipulate the runtime stack and frame pointer stack by utilizing the functions in the *runTimeStack* class. The VM itself also has a set of functions to be called upon by individual byte code subclasses to allow them to make changes to the runtime stack without directly accessing the *runTimeStack* class, which breaks encapsulation. Also, if “dump” is turned on, the runtime stack with its frames and the bytecode executed will be printed onto the output console right after its execution.

1.3 Summary of Work Completed

I started by creating the abstract superclass *Byte Code*, followed by all of its subclasses which are the individual byte code classes.

After that, I proceeded to implement the *ByteCodeLoader* class, specifically its *loadCodes()* function. I used the BufferredReader to read the file and a tokenizer to tokenize its content into String values. I also made a ArrayList of String to store the byte codes' arguments if there are any. These byte codes are then added into the *program* object, followed by a call of the *resolveAddrs()* function.

Next, the *Program* class's *resolveAddrs()* function was implemented to resolve branch codes' target addresses. I also created an *add()* function in the class to allow *ByteCodeLoader* to load the byte code instances into the program. Before being added, these objects are checked if they are instances of the *LabelCode* class. If yes, these objects' arguments, which are labels, are then stored into a hashMap that is created in the *Program* class to store their respective labels and addresses (indices in the file).

Then, I implemented the *RunTimeStack* class by implementing its list of functions like *peek(), pop(), newFrameAt()* etc to allow the VM to make changes to the RunTime Stack and FramePointer Stack. One important function that was implemented was the *dump()* function, which prints the runTimeStack with the correct frames after executing a byte code.

Lastly, I implemented the *VirtualMachine* and *ByteCode* subclasses altogether. Since the subclasses of *byteCode* interact with the VM class, it was easier to implement them simultaneously, rather than implementing one after the other. I created various functions in the VM class to allow the subclasses to make changes to the runtime stack without accessing the runtime stack directly. This avoids breaking encapsulation, though at the cost of some efficiency. After all the implementation, I manually set dump to "ON" in order to debug and test my code, making sure the program compiles without errors and produces the desired results.

2 Development Environment

Version of Java Used: Java 10.0.2

IDE Used: IntelliJ IDEA 2018.2.3. (Ultimate Edition)

3 How to Build/Import your Project

In order to build/import the project:

1. Ensure that you have an IDE and an updated version of Java, preferably version 8 or above.
2. Go to the GitHub repo consisting this project by clicking on the link on the first page of this document.
3. Clone the project using any command line program (i.e. GitBash) by entering *git clone "link"*.
4. In your IDE, go to "File", then "New", and select "Project from Existing Sources...", now browse to the folder that was cloned earlier and click "OK".
5. After that, just follow the instructions in your IDE and click "Next" when prompted until it is imported successfully.

4 How to Run your Project

In order to run the project:

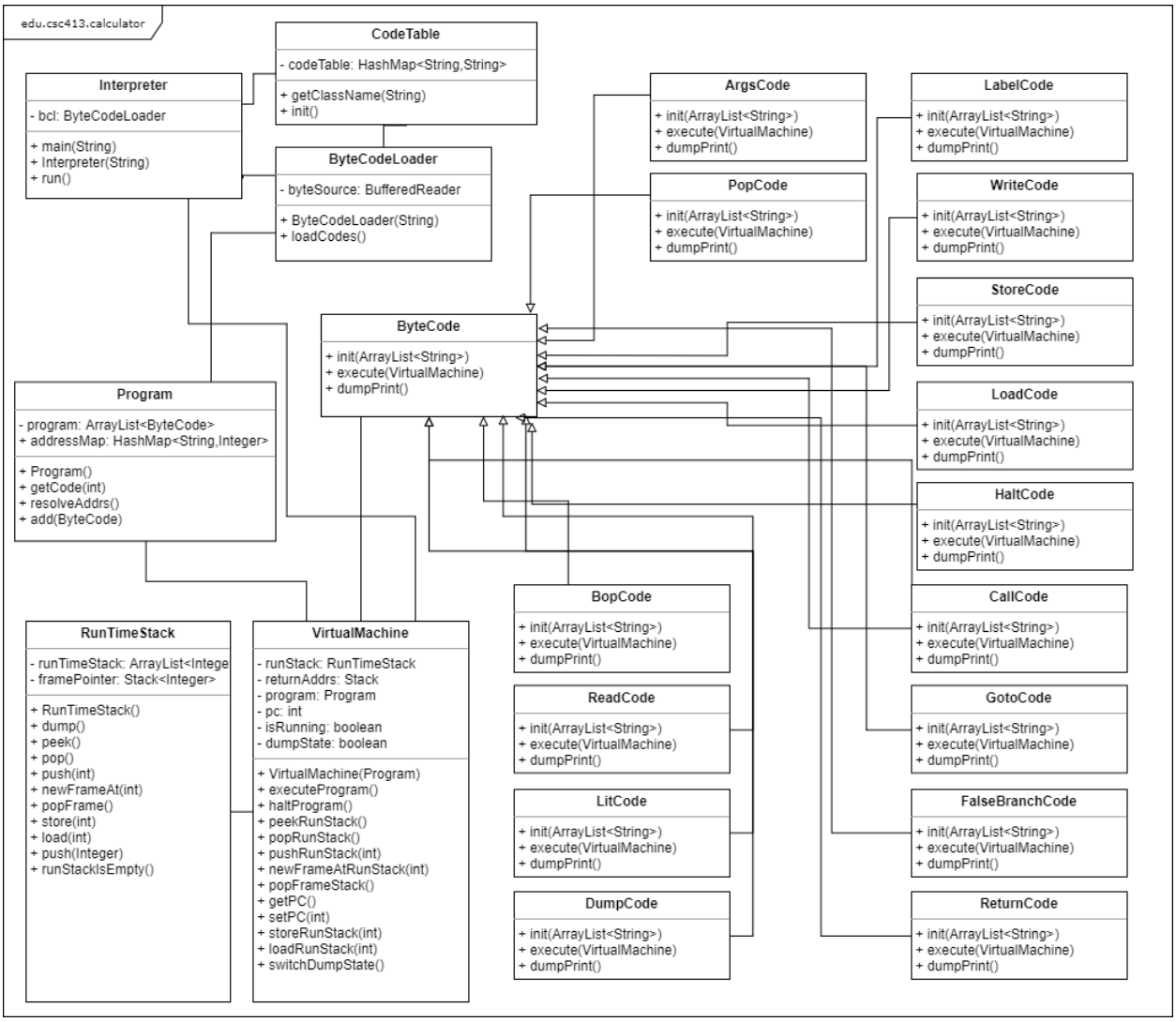
1. Right-click on the *Interpreter* file under the file navigation tab (typically on the left) and click *Build* or *Run.* Another option is just to open the *Interpreter* file and click on the green arrow button on the top right of the program to run it. The project will fail to run as it lacks a file to read but this will create a new configuration.
2. Now, click on "*Edit Configurations*".
3. A new window "Run/Debug Configurations" will pop up. Ensure you are on *Interpreter* under the Application tab. In the main section of the window, under "Program arguments: ", enter the file path for the file to be read. It is better to provide the absolute file path to avoid errors.
4. There are two test files provided along with the project, which are the *factorial.x.cod* & *fib.x.cod* files, which simulate the factorial algorithm and Fibonacci sequence respectively. These files' paths can be put under "Program arguments: ".

5 Assumption Made

1. Only files with compatible/suitable format will be run. (source code files with extension x.)
2. Files' content will not consist of byte codes outside of the 15 byte codes planned for.
3. The byte codes will be provided with appropriate arguments, if any.
4. The user input when prompted for an integer will always be a valid integer.
5. "CALL" code will always be executed after "ARGS" code.
6. When a "CALL" code is present, a corresponding "RETURN" code will be present too.
7. The argument for "BOP" is only limited to these operators: "+", "-", "*", "/", "==", "!=", ">=", ">", "<=", "<", "&", and "|".

6 Implementation Discussion

6.1 Class Diagram



The UML Diagram above shows the relationship between the classes.

The Interpreter class is the entry point of the program. The Interpreter class is run to interpret a source code file, and it takes a file as its argument. It will construct an object of itself and load the source code's byte codes into a program object. This will be done by the ByteCodeLoader class and CodeTable class.

The ByteCodeLoader class is responsible for loading the bytecodes into a Program object while the CodeTable assists it. The CodeTable class contains a HashMap that maps the source code file's

ByteCodes to their class representations by providing their class names. The CodeTable has two static functions, one to create an entry in the HashMap for the byte codes and the other to return the corresponding class name of a byte code in the source code. For example, the return value is "CallCode" if the value is "CALL". Meanwhile in the ByteCodeLoader class's loadCodes() function, it uses a BufferedReader to read the source code file line by line. It also uses a tokenizer to tokenize the line of String, in order to obtain the arguments of a byte code, if there is any. When a ByteCode is read, an instance of that bytecode class is created by referring to the CodeTable. The tokenizer will then read if there is an argument, and store it into an arrayList of String. This arrayList will then passed into that bytecode class's init() function to initialize it. After that, the bytecode instance will be stored into the program data-structure. After the ByteCodeLoader loaded all the bytecodes, the symbolic addresses will be resolved by calling the Program class's resolveAddrs() function.

The Program class is responsible for storing all the bytecodes read from the source code file. The bytecodes will be stored in an ArrayList of type ByteCode to ensure nothing else other than bytecodes can be added to it. The Program class has three functions: getCode, which returns the ByteCode at a given index; resolveAddrs, which resolve all the symbolic addresses in the program; add, which adds a given bytecode to the ArrayList of type ByteCode. For efficiency purposes, the bytecodes will be checked if they are an instance of LabelCode before adding into the ArrayList. This allows us to get the labels' addresses as we store them into the ArrayList. Therefore, in the resolveAddrs function, we only need to loop through the list of bytecodes once, looking for branch codes like CallCode, GotoCode and FalseBranchCode. These codes' symbolic addresses will be resolved to their corresponding target addresses by referring to the address map created earlier.

The RunTimeStack class is responsible for recording and processing the stack of active frames. This class contains two important data-structures to help the VirtualMachine execute the program. One of the data-structures is a Stack called FramePointer, which is used to record the beginning of each activation record (frame) when a function is called. In other words, it sets frames by keeping track of the indices of the runtime stack. The other data-structure would be the ArrayList<Integer> called runStack, which is used to represent the runtime stack. Since we need to access all locations of the runtime stack, an ArrayList is used instead of a Stack.

The VirtualMachine class is responsible for executing the given program, which makes it the controller of the program. The bytecode classes will need to go through the VM to make changes to the runtime stack. It has a program counter of type int which keeps track of which bytecode is executed, a returnAddrs Stack to store the return addresses for each called function excluding main, a isRunning Boolean to determine if the VM should continue executing bytecodes, a program object to refer to the program object where all the bytecodes are stored and a isDumping Boolean which determines if dumping is turned on. Except for the dump function in the RunTimeStack class, the VM basically contains the same functions which call the corresponding functions in the RunTimeStack class, acting as a middleman between the bytecode classes and the RunTimeStack class to avoid breaking encapsulation.

The ByteCode abstract superclass has no variables and concrete methods. All of its methods are overridden in its 15 children classes. Each ByteCode subclass has its unique responsibility. All of them have 3 methods, which are——init, for initializing the bytecode itself with the arguments given if any; execute, for carrying out its responsibility; dumpPrint, which constructs a String and passes it back to the VM for dumping if dumping is turned on.

7 Project Reflection

This is arguably the hardest project I have worked on in school due to its size & complexity regarding the relationships between classes. Though I started earlier than I usually would, it still took longer than I expected it to.

I read the assignment pdf file several times before I started doing the project. I followed its suggestion and started with creating the *ByteCode* superclass and its subclasses. I did not struggle much with the *ByteCodeLoader* class and *Program* class, though I did really have to brainstorm for resolving the addresses and storing the arguments of the byte codes. I am glad I went to class when our professor explained how it would be easy to use a hashMap structure to store the labels and their corresponding addresses.

Moving on, the *RunTimeStack* class was not too hard to implement, since the functions' instructions were given in the pdf file. At least, that was what I initially thought. But I did leave the dump() function empty and moved onto the *VirtualMachine* class.

I planned to implement the VM class before completing the ByteCode classes, but then I realized it was easier to do them together since the ByteCode classes rely on the VM to make changes to the *RunTimeStack* class. I struggled with the *init()* and *execute()* functions of the ByteCode classes, as well as figuring out what kind of functions these classes needed in the VM. I spent about a few days working on the VM and the subclasses. The ReturnCode & CallCode took longer than the others due to the complexity of their goals. Next, I had completed everything except for dumping. So I went back to the RunTimeStack to complete its dump() function. Again, I am glad I attended class when our professor explained that it would probably be easier to have two loops, with the outer loop based on the framePointer stack when printing. By printing in the perspective of the framePointer stack, I could focus on printing the stack frame by frame.

I kind of got everything done in the last minute on Friday night. I assumed my program will compile without errors and run successfully as I was careful when implementing it. Unfortunately, errors popped out one after another. I was in a panic to solve them, but I also realized I was not going to complete it on time. But thankfully, our professor extended the deadline for the project, which gave me extra time to debug my program.

As I started debugging, I realized most of my errors were related to the *RunTimeStack* class, the class which I initially thought was the easiest. I was already familiar with running in Debug mode thanks to the previous assignment and my internship over the summer, but this time I also had to learn to set breakpoints at suitable positions to debug more efficiently. One of the main issues I had was stack underflow. My initial fix only solved half of the problem, and I did not come up with a better solution until a day later. Also, I was confused to why using the ".add" method was causing me errors that the ".push" method would not. I switched to use the ".push" method instead and the problems were gone. But it took me hours to reach to that conclusion. I am happy that my program was able to compile without errors in the end.

8 Project Conclusion/Results

Through this project, I really strengthened my OOP skills, as it is a big project indeed. The classes are working with each other as they are supposed to.

Thankfully, the program is now able to compile without errors when running the two .cod files provided, which are the Factorial and Fibonacci files. When “dump” is turned on manually, the program prints the output on the console with the desired format & correct results.

In short, the project is a success.