CSC 413 Project Documentation

Summer 2019

Alan Ordorica

917139397

CSC413.02

https://github.com/csc413-02-summer2019/csc413-p2-aordorica.git

Table of Contents

Introduction:

1.1 Project Overview:

This project consisted of implementing a bytecode interpreter which was able to read a file containing a program translated into a series of byte codes with their respective arguments. It is able to parse each line of file into its token byteCodes and their arguments which are then sorted into bytecode classes and loaded into a bytecode Array List contained in a Program class. Using this method of sorting and organizing lines and byteCodes combined with an abstract ByteCode super class extending every instance of the bytecodes, as well as the use of a Virtual machine class, we are able to abstract the program so that encapsulation is not broken and the code will not end up breaking if small changes are made to other parts of the program. This Object-Oriented approach enables us to develop the code in a modular fashion consistent with encapsulation and program abstraction principles.

1.2 Technical Overview:

The finer details of the workings of this program consist of varying communication between the byteCode subclasses, the Virtual Machine, Runtime Stack and Program classes. To begin, the program starts by reading the source file which contains the byteCodes to be executed and places them, after parsing into corresponding labels, integer values and addresses, into a Program objects' byteCode Array List which we continuously access via the virtual machine class. By placing them into the program object, we then proceed to execute each bytecode using the virtual machines' execute method, but because they are in a program class array list, we are able to branch to the indicated bytecode, such as for Call, GoTo, and FalseBranch byteCodes, by modifying the VM class' program counter(pc) variable. Just like this the program proceeds back and forth between the Virtual machine and the bytecodes until the HALT bytecode is executed or the file is completely read. The program also makes use of the abstraction by passing any bytecode requests that modify the runtime stack through the virtual machine first so by creating pseudo-methods to the corresponding methods in the runtime stack class.

1.3 Summary of work completed:

The work I have completed is an essentially code-filled program in which all the byteCode subclasses are implemented as well as the Virtual Machine, Runtime Stack, and Program classes. However, I was not able to get a functioning factorial execution as there were some exceptions that propagated into the interpreter and runtime stack classes via the Bytecodes. From debugging I observed what looks like correct code but at some point, seems to repeat forever and thus eventually gives an index out of bounds exception.

Development Environment:

IDE: IntelliJ IDEA 2019.1.3 (Ultimate Edition) **--** Build #IU-191.7479.19, built on May 27, 2019

Java Version: java JDK 12

How to Build/Import Project:

In order to build the project, you must first import it to your IDE or local device. The way to do that is to first proceed to the GitHub link provided in the title page and, after following said link, to "clone" the entire repository via the clone button on the right side which provides a URL which must then be copied. If using IntelliJ IDE, proceed to open IDE and select the "import project" option and then navigate to where you downloaded the repo and select the outer CSC413 folder as the source not the inner Interpreter package folder. The next step is to configure the project settings, which consists of selecting the default settings shown and just select next until the project view appears. Ta-DA! It is done, now time to build and run it!

How to Run your Project:

First, start out but selecting the run menu and clicking 'edit Configurations', then selecting the Interpreter main class as the Main class subject and entering the filename for the factorial.x.cod file you would like to test run with. Next select the run menu and click 'Run Interpreter'.

Assumptions Made:

Some assumptions that were made while crafting the code are namely that the file containing all the byteCodes has no errors and is ready to read and execute. This allows to avoid accounting for file typos and other error checking. Other assumptions are that the LIT byteCode is dealing only with Integer values and no strings or other types, as well as that there is no divide by zero cases present in the logic of the file code.

Implementation:

I began first by crafting the ByteCode subclass skeletons, excluding any solid method implementations. Next, I followed the suggested guide and focused on implementing the ByteCode loader() method, going back occasionally to the byteCode subclasses to add and remove arguments. Next, I proceeded to form the Program class, which mainly focused on the resolveAddrs() method. I tackled this problem first by attempting three if() statements, one for each of the branching subclasses, until the suggested JumpCode subclass came, which led me to use the JumpClass instead and extending the other bracnhib classes from it, thus allowing me to reduce the lines to a single if statement to check if the selected byteCode is an instance of JumpCode. To resolve the address, I implemented a hashmap to store the corresponding labels for each of the jumpCodes in the HashMap paired with their index value from the program object as the value and the string label as the key. The next step for me was to implement the Runtime Stack and Virtual Machine classes which were made possible using similar set of methods on the VM as were on the RTS class. Essentially I implemented the methods in the runtime stack class which directly worked on the stack but for the bytecodes to use them they has to call the virtual machine pseudo-methods which in turn called the RTS methods, essentially “requesting” the required actions from the VM which then passes the request to the RTS class. Lastly, I completed the implementation of the byteCode subclasses corresponding to their required actions.

Reflection:

I at first really did believe the project was difficult and for me it seemed incredibly daunting and intimidating. It took a very long time for me to fully grasp how the flow of the code was supposed to go but after I did, the code generation went much smoother. Looking back the one major thing I would change is that I would dedicate more of the time I spent on this project into understanding the code flow rather than trying to just fill everything in. It was more difficult and time costly to go over the same lines multiple times trying to understand what they were supposed to do. The part that gave me the most trouble was definitely the Program and resolve address sections. All in all, however, this project was incredibly rewarding, and sure enough gave me a boost in confidence as to what I can create.

Conclusion and Results:

To sum up this project, I would say the program as of yet does not fully load a bytecode program but will soon! I would say it is also implemented well so as to keep to the abstraction and encapsulation guideline and further provide modularity in the event that it is modified in the future.