CSC 413 Project Documentation

Fall 2019

Lukas Pettersson

916329042

CSC 413.01

https://github.com/csc413-01-spring2020/csc413-p2-LukasPettersson

Table of Contents

1 Introduction

1.1 Project Overview

When a Java program is compiled the code that the programmer writes is interpreted by the IDE (integrated development environment) and turned into a list of “Bytecodes.” These bytecodes consist of constants, numeric codes and references that the compiler executes. These bytecodes together form a program. The purpose of this project is to read in these bytecodes and execute them line by line to calculate the output of the program. Each bytecode has its own functionality and arguments that need to be taken into consideration when executing them. The programs that we are working with are written in a made-up programming language with its own syntax.

1.2 Technical Overview

This program assumes that we have a program already compiled into bytecodes. We start with parsing each of these bytecodes and creating objects of each bytecode with the name and each of the arguments loaded into it. After this we load our bytecodes into an ArrayList of bytecodes that represents our entire program. Some of the bytecodes have symbolic addresses that symbolize points in the program to jump to called “Labels” (labels themselves are a bytecode in our program) These symbolic addresses need to be interpreted and given a integer value of the line that needs to be jumped to. The bytecodes that have these addresses are part of a subset of codes called Jump codes. These codes implement an interface with supporting functions to give them the extra functionality needed by the codes. After the program has been created and addresses are resolved the program is ready to be ran and is sent to the virtual machine.

The virtual machines job is to execute the program. To do this it has 6 data members that help us out. There is the program object that we read into earlier. Aa program counter to keep track of what the next instruction to execute is. A runtime stack object to keep track of and store values that are used by our program. This is done by keeping track of activation records in a frame pointer stack, which is part of the runtime stack object. A return address stack to keep track of where the program counter should jump to after a function call is made in our program. Lastly there are two Boolean values, isRunning and isDump, which keeps track

of if the program is running and if the program should be dumping bytecodes and the runtime stack to the console. The virtual machine is the of the project and communicates with the runtime stack object to add and remove values as per the bytecode's instruction. The runtime stack is abstracted and does not know any information about the bytecodes it simply stores the required values. The interpreter class acts as a wrapper, calling on the functions that need to be called to load and then execute the program.

1.3 Summary of Work Completed

I implemented four classes to get this project working correctly. They are as follows: ByteCodeLoader, Program, RuntimeStack, VirtualMachine. A descrtion of the work that was completed for each of the classes is listed below.

ByteCodeLoader:

This class is responsible for taking the bytecode data from a file and creating an instance of the program class with all the bytecodes in it. The way that I did this was to first read in each bytecode from the file and parsing it, with that parsed data the correct bytecode object was created. Which is then initialized and added to an instance of the program class.

Program:

The Program class' responsible is to store all the bytecodes and to resolve symbolic addresses as discussed in the technical overview. All the codes that have symbolic addresses to resolve are implementing the jumpCodes interface. This makes it easy to parse through the program using two for loops, the first for loop is looking for a code that has an address that needs to be resolved by checking if the code implements the jumpcode interface. When such a code is found, the second for loop runs through the program a second time looking for the matching a label with the matching symbolic address and stores the index of that label code to an integer "resolvedLabel" inside the jumpCode.

RuntimeStack:

The runtimeStack stores all the values that are required by the program later. There is a dump function to dump state of the runtime stack, this function is called from the virtual machine if the isDump Boolean is true. The dump function was implemented by copying the runtime and framepointer stack that is stored inside the runtime stack object and popping off the copied objects storing them inside a an arrayList "outputList". The outputList is the written to the console at the end of the function through a for loop. I also implemented member function to add and remove objects to the stacks. Peeking the stacks, adding activation records to the stack, getting values from the stack, etc. These functions are only called by the virtual machine.

VirtualMachine:

The virtual machine holds the execute function for the whole project, this is where the run loop is held. It fetches the byte codes from our program object at the current programCounter, calls this bytecode's execute function, increments the program counter. It dumbs the bytecode information if the isDump variable is set to true. In addition, it also implements member functions to interact with the runtime stack. Note that this is the only point in the program where I am interacting with the runtime stack class. Many of the member functions are used in multiple byte codes the retrieve information and interact with the runtime stack class.

2 Development Environment

Java version: 13.0.1

IDE: IntelliJ IDEA community edition

3 How to Build/Import your Project

Steps to import and build project are for MacOS

1) Import the project using git, this can be done by cloning this repository with the link provided at the top of this document.
2) Launch IntelliJ IDEA and from file > open > csc413-p2-LukasPettersson
3) Select the configuration to build from in the top right drop down menu between the build and run buttons.

4) Build the program using the hammer icon in the top right corner next to the drop-down menu

4 How to Run your Project

By default, the programs will print information about the bytecodes and the runtime stack. To turn this behavior off navigate to the virtual machine class located in:

/csc413-p2-LukasPettersson/Interpreter/VirtualMachine.java

After the desired behavior is configured, press the run button in the top right corner of the IntelliJ IDE

5 Assumption Made

- The two test programs (fib.x.cod and factorial.x.cod) are generated correctly and contain no errors
- No test case where division by 0 occurs.
- Lit is always an int declaration

6 Implementation Discussion

6.1 Class Diagram

Too large for this page, included in documentation folder as ".png"

7 Project Reflection

Throughout this project I felt that I learned a lot about how to use Java to create OOP programs. I had trouble time managing due to other school-work, but in the end I got everything done and it is working as expected. I better understand Java handles Bytecodes. I found it to be really fun to do this project as a Computer Engineer because I can draw a lot of ties between computer systems classes and what we are doing here with the virtual machine with the runtime stack and frame pointer. The hardest part for me was figuring out how the runtime stack and the frame pointer worked together, as well as popping and pushing them from the perspective of the virtual machine. Overall, I learned a lot on this project and I had a lot of programming it.

8 Project Conclusion/Results

In the end I was able to finish the project with the correct output. The biggest challenge that I faced while writing this program was not being able to test the functionality of my program until all the bytecodes were written. However, this also led me to focus and make sure that the Bytecodes functioned correctly logically without testing so that I would not have any bugs in the final product and have to debug a lot of the Bytecodes. This was one of the most challenging projects that I have done in college so far and I am proud of my problem-solving skills.