CSC 413 Project Documentation

Fall 2018

Aung Hein

917649101

CSC413 02

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

Table of Contents

1 Introduction

1.1 Project Overview

A program that translate a made up language and executes them.

1.2 Technical Overview

This is an interpreter program for mock language X. The interpreter is responsible for processing byte codes that are created from the source code files with the extension x. The interpreter and the virtual machine will work together to run a program written in language X.

1.3 Summary of Work Completed

I created the different bytecode classes that corresponds to the new mock up language and implemented on how they are executed. A loader class for bytecodes which chops up the lines in the input file and creates instances of each bytecode. A program class which resolves the addresses in the argument parameter of certain bytecodes. A runtime stack which simulates memory space for each function with given parameters and has methods for manipulating the memory stack. Finally a virtual machine which will execute the program.

2 Development Environment

a. jdk 1.8.0_121

b. IntelliJ IDEA

3 How to Build/Import your Project

I started off with cloning the given repo off GitHub then imported the class folder instead of the interpreter as a normal java project.

4 How to Run your Project

I had to run the interpreter to compile the file then I had to give it one of the file name in an argument section at the configuration menu to be able to read the file.

5 Assumption Made

The file would not have any incorrectly written bytecodes or empty lines. It’s also implemented in a way that certain variables are encapsulated so that specific classes should not have access to them. Since there can be multiple instances of each bytecode, an arraylist data structure is used.

6 Implementation Discussion

This program uses object oriented programming with bytecode classes to create instances of written bytecodes in the file. Each bytecode has an arraylist to store arguments and addresses of each bytecode and has methods to execute, and return the data given. The bytecode loader uses a BufferReader to read the input file then breaks down the words and create instances of them. The CodeTable uses HashMap so that it’s easier to match the file input with the class names. The program class stores each bytecode instance inside an arraylist and resolves argument addresses which is done by using a HashMap to store each argument/address in the order that they are in then only initializing the bytecode that uses addresses. The runTimeStack has two primary data structure which is an array list to store the data and a frampointer stack to keep track of functions. The Virtual Machine uses Boolean values and counters to obtain and go through bytecode data information as well as runTImeStack and FramePointer Stack data fields.

6.1 Class Diagram

7 Project Reflection

This project was fairly hard to debug in general, due to amount of nested method calls. It also takes a pretty good understanding of how the project works in order to implement the classes and methods requested. Reading the pdf helps a lot. However, I wasn’t able to get the dump function working properly.

8 Project Conclusion/Results

I was able to get the right result on the Fibonacci file without dump but it did not go as well for factorial which meant that either the bytecode isn’t working as intended or some of the push and pop methods in runtimestack class. The extension did help since it gave me more time to figure out some bugs that appeared in my program.