CSC 413 Project Documentation

Spring 2020

Duy Nguyen

Student ID: 917446249

CSC 413.03

GitHub Repository Link

Table of Contents

[1 Introduction 3](#_Toc522827688)

[1.1 Project Overview 3](#_Toc522827689)

[1.2 Technical Overview 3](#_Toc522827690)

[1.3 Summary of Work Completed 3](#_Toc522827691)

[2 Development Environment 3](#_Toc522827692)

[3 How to Build/Import your Project 3](#_Toc522827693)

[4 How to Run your Project 3](#_Toc522827694)

[5 Assumption Made 3](#_Toc522827695)

[6 Implementation Discussion 3](#_Toc522827696)

[6.1 Class Diagram 3](#_Toc522827697)

[7 Project Reflection 3](#_Toc522827698)

[8 Project Conclusion/Results 3](#_Toc522827699)

# Introduction

## Project Overview

This is an Interpreter program, which takes this programming language X, and interprets the language and runs a program using it. This language X is just a simple one that uses only a few commands to run a program. The Interpreter will go into files in the X language and interpret what its instructions. It will read files with type x.cod at the end of their names. The interpreter only supports certain commands.

## Technical Overview

The Interpreter program takes mock language X, it is like a simpler form of Java, and it processes byte codes created from source code files with the extension x. It runs takes and runs it alongside a virtual machine. The byte codes are put into run time stack (RTS) and there is a frame pointer stack (FPS) to hold and maintain the activation records.

The ByteCodes the interpreter supports are the Args ByteCode, Bop ByteCode, Call ByteCode, Dump ByteCode, False Branch ByteCode, Goto ByteCode, Halt ByteCode, Label ByteCode, Lit ByteCode, Load ByteCode, Pop ByteCode, Read ByteCode, Return ByteCode, Store ByteCode and Write ByteCode. Each of the ByteCodes are subclasses of an abstract class, ByteCode. They all must have the methods init, execute and toString. The init takes an ArrayList of strings and initializes the arguments. The execute takes a VirtualMachine as an argument and execute the function of the ByteCode called. The toString is to just format and returns a string that shows what is happening and what ByteCode is being called.

The Args ByteCode sets up how many arguments the function has. It will always be executed before the Call ByteCode, and takes only one argument, which is the number of arguments for the following function call. This number is used to figure out how many values from the top of the RTS will be a part of the activation frame of the next function call. It will find where the frame begins and push that index into the FPS.

The Bop ByteCode implements binary operations, such as basic mathematical operators and logical operators. It removes 2 values from the RTS, completes the operation asked using them and pushes the result back into the top of the stack.

The Call ByteCode is how the VirtualMachine moves from each location in the program to execute the code sections we call Function. When encountering this ByteCode, it jumps to the corresponding label in the program and is responsible for where control goes after a function is completed.

The Dump ByteCode turns dumping ON or OFF, where dumping only occurs when it is ON. This ByteCode is not to be dumped however.

The Halt ByteCode will stop the VirtualMachine object from running.

The Label ByteCode has no function and only marks the location in the program where other ByteCodes can jump to.

The Pop ByteCode takes an integer value and pops the same amount of objects in the VirtualMachine object’s RTS.

The Read ByteCode takes user input from the keyboard and the only inputs allowed are integer values.

The Return ByteCode helps return from functions and puts the correct return values in the correct position in the RTS. This is used for convention for returning values and handling arguments. Functions are required to return values in the correct spot in the RTS before returning said value.

The Write ByteCode displays the information to the console, and the only thing it can display is the top of the RTS and nothing else.

## Summary of Work Completed

The only class that didn’t need to be touched is the Interpreter class.

# Development Environment

The IDE used to program this project is IntelliJ using JDK 12.

# How to Build/Import your Project

# How to Run your Project

To run the project, you need to right-click the Interpreter class and select “Edit Interpreter.main()…” option. A menu should pop up, and in the “Program Arguments” field, type in the x.cod you would like to run, ex. factorial.x.cod, fib.x.cod, etc. Afterwards, you can hit Apply and OK and you can hit Run or the play button for the Interpreter class.

# Assumption Made

* The Interpreter class does not need to be edited
* Creating a toString() function makes everything cleaner
* Dumping happens after resolving addresses
* Dumping only occurs in the virtual machine
* Neither of the given cod files should throw exceptions, but it should be able to handle exceptions

# Implementation Discussion

## Class Diagram

# Project Reflection

The Interpreter definitely is harder than the Expression Evaluator. It was very helpful to go to lecture and hearing everyone’s questions about frame pointer stacks, the run time stack and more. I initially wanted to use the Operator class from the Expression Evaluator for BopCode but as I was doing it, it seemed like a lot more work, so I just decided to use a switch case instead. It would be a lot cleaner but it seemed like a long walk for a short glass of water. The whole project was a lot of typing. The ByteCodes took longer than I thought to make them all, not only because it’s repetitively creating a bunch of them but also understanding what each one does and how dumping should be handled. The concept of ByteCodes was really difficult to understand for a bit with some of the names and making sure they do the right thing and has the right toString format. Once dumping worked, it made it a lot easier to figure out where the issues were, but it also took some time to get dumping to work. But this helped me realized how important it is to break down the big problem into smaller ones and solving them bit by bit.

# Project Conclusion/Results

The interpreter is definitely a few steps higher in difficulty in comparison to the Expression Evaluator. It was a good way to realize that breaking down the problem is important and that big problems cannot be easily solved by directly trying to tackle them. Understanding the concepts before coding was imperative to being able to do the project, looking at examples and tracing the x.cod files is very helpful.