CSC 413 Project Documentation

Spring 2019

Raya Farshad

917940106

CSC413-03

https://github.com/csc415-03-spring2019/csc413-p1-rayafsh.git

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

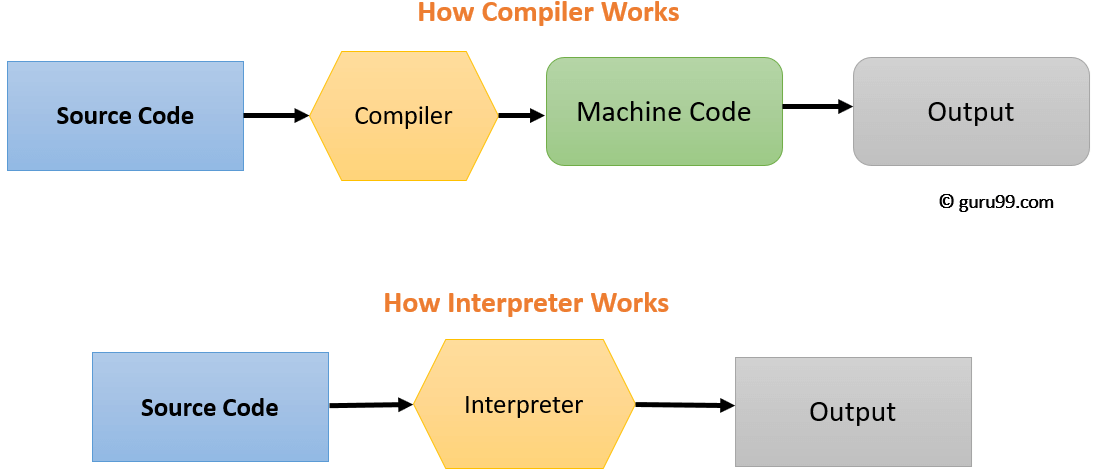
Programs written in high level languages are either directly executed by some kind of interpreter or converted into the machine code by a compiler to execute.

For this assignment I implemented the interpreter. The interpreter program is responsible for processing the bytecodes that are created from the source code files with the extension x (fib.x.cod).

## Project Overview

We have two ways for translating the high-level language, compiling, which reads the program input and the output is the machine language code. As an example, the Unix program g++ transforms a C++ source file into the machine executable file a.out. As a second example the java compiler javac transform a .java source file into a .class file that is written in java bytecode, and the interpreter translate the .class file into the code that can be executed. The interpreter is a program that translate the source program instruction by instruction.

In the current project we have had two bytecodes fib.x.cod and factorial.x.cod and we want to create an interpreter that can execute it.



## Technical Overview

The basic idea of this project was to create an interpreter and virtual machine to run a program written in language x. There are some basic components(modules) in order to make this project done properly.

Few notes:

* We have a code table (in CodeTable class) which maps each bytecode to specific separate class name.
* We should create separate sub class for each class name (e.g. class readCode) in separate file.
* In order to keep the encapsulation, we create an abstract ByteCode class which each bytecode subclass, indirectly extends from that.

In following paragraph I am explaining the important modules I worked on, during the project, in order to translate each bytecode line by line to the machine code.

1. First we should work on the BytecodeLoader. This class is responsible for loading bytecodes from the source code file into the data structure that stores the entire program. The arraylist is used to store the bytecodes which is inside the program object.
2. The Program class is for storing all the bytecodes that reads from the source file in the array list and resolving all the addresses.

-in order to resolve the addresses I go through the whole program to find all the labels and save their addresses into a hashmap.

Then after that I loop again through the program to search for GoToCode, CallCode and FalseBranchCode in order to find the relevant address for those bytecodes labels and set the address to those classes.

1. Runtime stack class is a class to help the virtual machine execute the program. This class is responsible for recording the stack of active frames.

* We have two data structures in order to correctly execute the program: Stack Frame pointer and runtime stack.
* Stack frame pointer: Is used for recording the beginning of each activation record (frame) when calling each function.
* Runtime stack is contain the input value and the temporary values in the process which is in an array list format in order to access all locations of the runtime stack,



1. Virtual machine is the main class used for executing the program. VM is the controller of the program and all the operations needs to go through this class. VM contains the method to manipulate the runtime stack, program count, return address and dump.

## Summary of Work Completed

During this project I have completed the required class and I added more bytecode subclasses in order to deal with factorial and Fibonacci .x.cod file.

# Development Environment

Following development environment made working with evaluator expression project possible:

1. Intellij IDEA version 2018.1(Ultimate version)
2. Jdk1.8.0\_161.jdk

# How to Build/Import your Project

Steps to import project:

1. From the git bash clone the repository from GitHub using the command:

**git clone** https://github.com/csc415-03-spring2019/csc413-p1-rayafsh.git

2. Open IntelliJ and select: **Import Project**

3. Navigate to the folder where the project folder was cloned. Inside the project, folder navigate to the folder path: **rayafarshad\** **csc413-p2-rayafsh**

4. Next select: Create project from existing resourses

5. Select: **Finish** to complete the Project Import Steps.

Steps to build the project:

1. In the IntelliJ under **Build** Menu select: **clean project** to guarantee clean build for the first time

2. Next under **Build** Menu select: **build project**

# How to Run your Project

Run -> Edit Configuration -> Under the Program argument choose either fib.x.cod or factorial.x.cod

# Assumption Made

During this project I assume that those given programs fib.x.code and factorial.x.cod are working properly.

# Implementation Discussion

## Class Diagram

# Project Reflection

In this project I have learned that in order to create a big project I need to separate it first into smaller module. Follow it step by step and try to make a connection between each smaller module. Another thing that I learned was it is very important to keep the encapsulation. In the clearer word, we want to create a program to work for any other bytecode input just the same. That’s why in the project the instructor insists on not to break the encapsulation. For example, bytecodes was not allowed to access or write to the runtime stack. It needs to request from the virtual machine. We have been asked to code to the VM not in the bytecode. After working on this project, I realized how important is that to keep those encapsulations in order to generalize the program to different inputs.

# Project Conclusion/Results