CSC 413 Project Documentation
Fall 2019

Vaisakh Kusabhadran
920442702
413.[03]

https://github.com/csc413-03-fall2019/csc413-p2-vkusabhadran

Table of Contents

1. Introduction

1.1. Project Overview

This project implements an interpreter and Virtual Machine. The interpreter is for a mock language X. It works together with the Virtual machine to processes the byte codes that are created from the source code file. The Virtual Machine controls the interpreter. Two input files Fib.x.cod and Factorial.x.cod which computes the Fibonacci and Factorial, respectively, of a number is being processed here.

1.2. Technical Overview

This interpreter is implemented in Java 11.0.3 in IntelliJ IDEA largely based on the data structures HashMap,Stack, ArrayList. The input file of byte codes get processed by the interpreter and get stored in the ArrayList and Stack data structures based on the byte codes and are then processed.

1.3. Summary of Work Completed

The interpreter has been implemented having the fibonacci and factorial getting calculated as expected in terms of the results as well as the dumping the byte codes. No issues were found on cross checking of the dump between this interpreter and the reference dump for factorial of 10.

2. Development Environment

The project has been implemented in IntelliJ IDEA ULTIMATE 2019.2 edition.

3. How to Build/Import your Project

Please follow the below instructions to import this project:

1)Open command prompt and type in git clone git@github.com:csc413-03-fall2019/csc413-p2-vkusabhadran.git. This will download the zip file to you current folder.

2)Unzip this folder to get the “csc413-p2-vkusabhadran” folder.

3)Open IntelliJ now and import the interpreter folder inside the folder mentioned in step 3.

4)Please keep the project name as interpreter itself.

Now the project and its related libraries will be imported to IntelliJ.

4. How to Run your Project

Please follow the below instructions to run the project:

1)Open the Interpreter Class - interpreter.Interpreter.

2)Edit the configuration of the the class to give the below details :

Program arguments : fib.x.cod/factorial.x.cod(depending on which program you want to run. For an input of n, fib.x.cod will give the n^{th} fibonacci number and the factorial.x.cod will give you n factorial.

Working directory : This should be the location of the above file(fib.x.cod/factorial.x.cod).

3)Apply the above configurations and run the interpreter class.

4)You will be prompted to enter an integer which for which the fibonacci/factorial(depending on the byte code file you are using) has to be found out.

5. Assumption Made

1)The given byte code source programs used for testing are generated correctly and error free.

2)All the byte codes gets created in capital letters.

3)There will always be a HALT instruction at the end of the byte code file to stop the execution of the program.

6. Implementation Discussion

The interpreter is implemented on top of Stack, HaspMap and ArrayList. The important data structure specifications are as follows :

DATA STRUCTURE	USAGE	PURPOSE
ArrayList<Int>	runTimeStack	Represent the runtime stack
Stack	framePointer	Record beginning of each activation record
Stack	returnAddrs	Stores return address of each called function
HashMap	codeTable	Maps byte codes with their classes
HashMap	labelMap	Maps the label and its line for address resolving

6.1. Program flow

This section explains how the program flows for a sample byte code. The program flow is as follows :

1. Run the program from the interpreter class.
2. The interpreter take the input file(fib.x.cod or factorial.x.cod) and reads it line by line.
3. For each line the *loadCodes* function in ByteCodeLoader Class is called which parses the byte code and separates the byte code and the argument.
4. This byte code gets added to an ArrayList *program*. Also, in case of jump codes(GOTO, CALL & FALSEBRANCH) the elements in the program(which are byte codes) gets amended to substitute the next instruction address instead of the label name. This is done in the *resolveAddrs method*.
5. Once all the addresses are resolved, the program is ready to be processed. This is where the Virtual Machine takes over the processing.
6. The Virtual Machine traverses through each element in the *program* ArrayList and execute it. All the operations on the data structures(runTimeStack and framePointer) goes through the Virtual Machine.

These steps gets repeated through each of the byte code causing the bytecode file to be processed.

6.2. Students involved

Jair Gonzalez

Siddhi Rote

Vaisakh Kusabhadran

6.3. Class Diagram

Class Diagram is also available at : **https://github.com/csc413-03-fall2019/csc413-p2-vkusabhadran/blob/master/Interpreter_UML.pdf**

7. Project Reflection

- Better understanding of Hashmap, Stack and ArrayList
- Better understanding on the background processes involved in a
    - Function call and return
    - Data type declaration and initialization
    - Arithmetic operation
    - Segregation on local variable of a function from another's

8. Project Conclusion/Results

The results were as expected as processing both the input files(fib.x.cod and factorial.x.cod) gave correct results and also the dump of 10 factorial matched the reference.