CSC 413 Project 2 Documentation

Fall 2018

Jonathan Julian

918164239

CSC 413.02

https://github.com/csc413-02-spring2019/csc413-02-jjulian91

Table of Contents

1 Introduction

1.1 Project Overview

The interpreter is a virtual machine which has the purpose to run and interpret a new language similar to the MIPS architecture. The program was approximately 30% built.

1.2 Technical Overview

The program has a simplistic structure and call hierarchy. Entry to the program is done via the Interpreter class's main method which instantiates an interpreter object. The instantiation of the Interpreter initializes the Code table (list of all available "bytecodes" and calls the ByteCodeLoader class. The ByteCodeLoader class is passed in a file (arg[0] defined in the build instructions below). The ByteCodeLoader parses the file and puts it line by line (separated by "\n") into a buffer. From here the interpreter class invokes it's "run" method which instantiates a "Program" object (similar to a call stack in MIPS) as well as a "Virtual Machine" object used to interface with the program to handle the runtime stack.

The Program object is instantiated via the LoadCodes method call to the ByteCodeLoader class. The LoadCodes() method uses the buffer populated in the previous step and parses each line for a series of tokens. These tokens are then used to invoke a class via Java Reflection (using an indirect call to instantiate an object via a string resolving in a class path name). Each line is then parsed for subsequent tokens which are passed into the objects respective init function to populate its data members. Each of these "Codes" are placed into an arraylist which correlates to a call stack for the program . Before returning the Program object the LoadCodes method calls the ResolveAddrs method.

The ResolveAddrs method searches the ArrayList for any and all calls which are required to jump or return to a different program counter (another location in the array list). This is accomplished by creating 2 new ArrayLists, one for Calls to a function, one for the function being called. This allows for a single pass through the ArrayList, pulling out only the necessary elements then systematically searching the Label (Address to be jumped to) for the current element of the Jump instruction. This then updates the Jump instruction with the ordinal value in the ArrayList of the Label it calls for later use.

Once the Program's ArrayList is returned from the ByteCodeLoader class the code continues with the VirtualMachine object (VM) and calls the VM's execute program. This method sets the PC (ProgramCounter) to 0 to begin the project, creates a <Stack> FrameStack and <ArrayList> RunTimeStack. The program enters a While loop which then calls a code from the Program ArrayList, executes the code and increments the program counter to move to the next element of the Program ArrayList. This is continued until a "Halt" code is reached which then sets the isRunning Boolean to false to exit the while loop.

The runTimeStack and the FramePointer are used in tandem while executing the program to track every number which is placed on the stack as well as which method has placed the number there. Every call to a new method will increment the FramePointer so the program can reference the frame to which the variable or integer on the runStack belongs to. This is imperative for when the program calls a new function or returns from a function. Items are loaded to the runTimeStack then a new call to a function the FramePointer increments the current frame and then a subsequent Args code is called to establish where the new frame began. When returning from a function, the top argument is popped and stored, and the runTimeStack and the Frame pointer are popped until the frame reference no longer matches that of the stored argument. The stored argument is then returned to the runTimeStack with its FramePointer decremented to match the current frame. This behavior is continued until the program calls Halt.

1.3 Summary of Work Completed

The program required finishing the ByteCodeLoader.loadCodes() method. This was accomplished with a loop for reading in entries from the buffer. The arugments are parsed in a try catch block to catch any potential errors when creating the objects via Java Reflection. These arguments instantiate an object and update that objects datamembers via the objects init function.

Next the Program.ResolveAddrs() method required implementation which was completed via searching the entires of an arraylist and separating matching entries into 1 of 2 categories, Jump or Label. Then the entries are compared until a match is made between a jump and label and a jump instruction is updated with the labels ordinal address in the original ArrayList.

The RunTimeStack Class required complete set up via the guidance in the assignment PDF. The logic was applied with previous knowledge of the use of stacks and Arraylists. These methods then have corresponding accessors created in the Virtual Machine class (the handler of the RunTimeStack).

After the environment was set up and the program was able to run and populate the Program ArrayList the final steps were to set up the ByteCode classes. During the setup of the ByteCode classes there was a conscience choice made to leave the name variable in the child classes to save the call of a getName method for every bytecode when using the toString class. To begin the set up of the bytecodes I looked for similarities in the arguments of each byte code. This lead me to see the easiest method to set up the program was through abstraction and interfacing used together. The ByteCode class had to be the parent of all Byte codes and since there was not a clear defining feature of all ByteCodes to have an abstract method I used the ByteCode parent class as an interface. This allowed for 2 more abstractions to be made for Branch ByteCodes and the LoadStore type ByteCodes.

The ByteCodes were set up in accordance with the PDF instructions and the logic was very straight forward. The next hurdle to overcome was the BOP code needing access to different operators. I accomplished this by just importing the working code from my evaluator project and updated/added the new functions.

2 Development Enviornment

The development environment used was Intellij IDEA 2018.3.4 x64. The program was built using JDK 10.

3 Build Instructions

If IntelliJ already has a file open, go to File - > Close Project. This will take you to another IntelliJ page and from there you can select checkout from Version control. With access to Git Repo go to the following link: https://github.com/csc413-02-spring2019/csc413-02-jjulian91 . You will be able to clone the repo from here. Once the repo is cloned you will be able to open this as a project with name of your choice following the prompts selecting all the default settings. From here select the drop down in between the hammer icon and the play icon in the top right corner of the IDE. Select the plus sign and choose application from the list on the left pane then ensure the following settings:

3.1.1.1 Main Class : interpreter.Interpreter

*3.1.1.2 Program Arguments : fib.x.cod **OR** factorial.x.cod (Or any .x.cod which was compiled from language X. This is arg[0] passed into main)*

3.1.1.3 Working Directory : $MODULE_WORKING_DIR$

*3.1.1.4 Use Class Path of Module : *the main directory of **your** project**

3.1.1.5 JRE: JDK 10

4 Run Instructions

With the settings as described above you will be able to just click the play button. If fib.x.cod or factorial.x.cod are selected you will be prompted in the console for a user input. Select an integer. You will be prompted to enter an integer until one is entered. The result of the function (fib or factorial) will be displayed in the console.

5 Assumptions Made

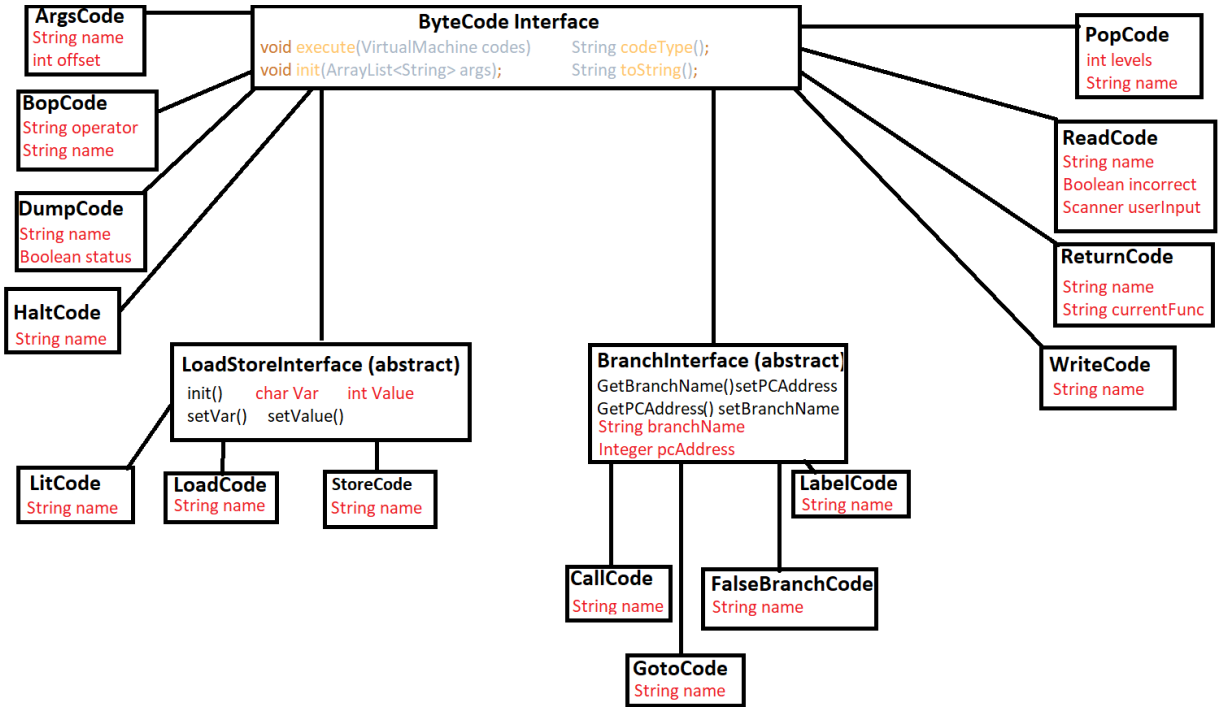
Assumptions made are:

A. *File being passed in is indeed a full and correct implementation of code X.*
B. *No additional ByteCodes are used (other than described in the pdf)*
C. *Program ends with a HaltCode.*
D. *Program does not overflow Integer limits. (factorial exceeds this rather quick)*
E. *Program length does not exceed Stack limits for JavaRuntimeStack.*

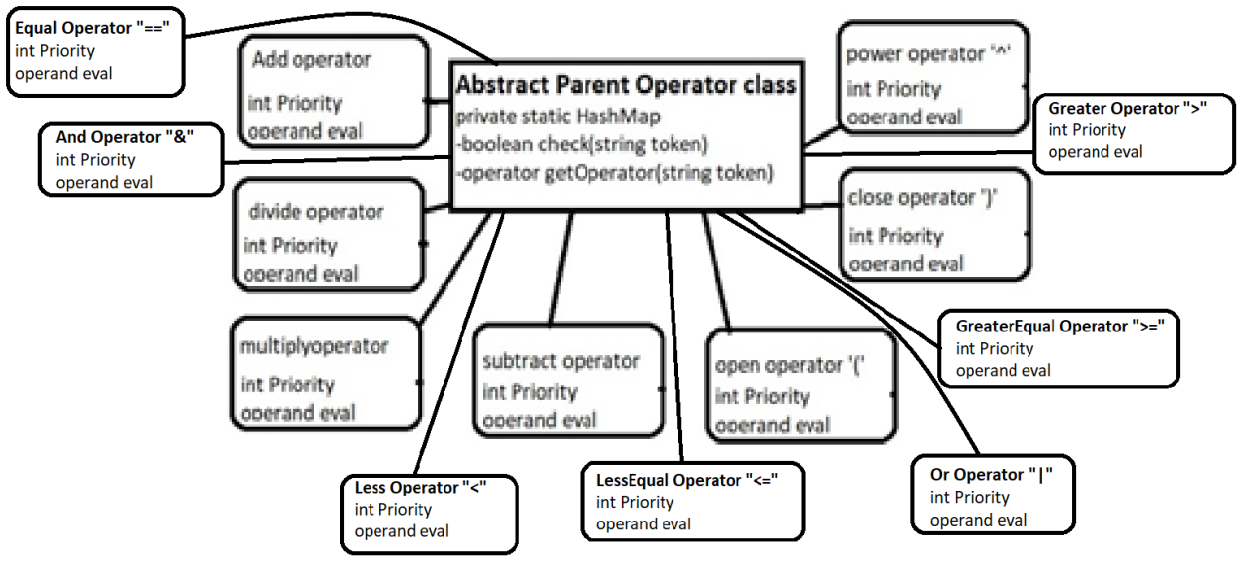
6 Implementation Discussion

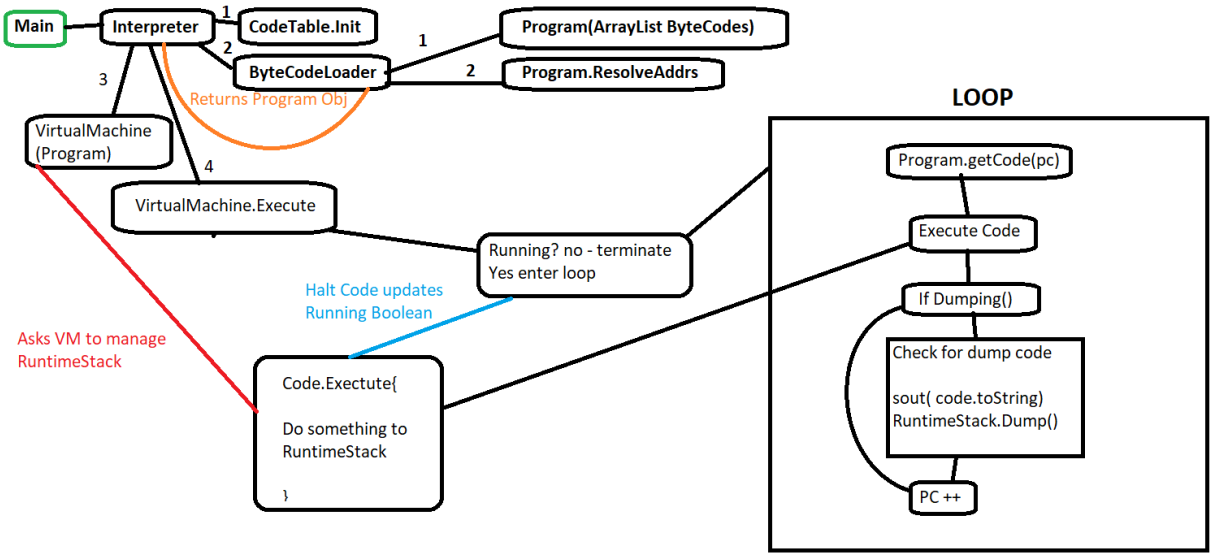
A Majority of the implementation is discussed above in technical overview. See below for class diagrams to provide additional insight to program implementation.

6.1 Class Diagram


ArgsCode
String name
int offset
ByteCode Interface
void execute(VirtualMachine codes)
void init(ArrayList<String> args);
String codeType();
String toString();
PopCode
int levels
String name
BopCode
String operator
String name
ReadCode
String name
Boolean incorrect
Scanner userInput
DumpCode
String name
Boolean status
ReturnCode
String name
String currentFunc
HaltCode
String name
WriteCode
String name
LoadStoreInterface (abstract)
init()
char Var
int Value
setVar()
setValue()
BranchInterface (abstract)
GetBranchName()setPCAddress
GetPCAddress() setBranchName
String branchName
Integer pcAddress
LitCode
String name
LoadCode
String name
StoreCode
String name
LabelCode
String name
CallCode
String name
FalseBranchCode
String name
GotoCode
String name

B


Equal Operator "=="
int Priority
operand eval
Add operator
int Priority
operand eval
Abstract Parent Operator class
private static HashMap
-boolean check(string token)
-operator getOperator(string token)
power operator '^'
int Priority
operand eval
Greater Operator ">"
int Priority
operand eval
And Operator "&"
int Priority
operand eval
divide operator
int Priority
operand eval
close operator ')'
int Priority
operand eval
multiplyoperator
int Priority
operand eval
subtract operator
int Priority
operand eval
open operator '('
int Priority
operand eval
GreaterEqual Operator ">="
int Priority
operand eval
Less Operator "<"
int Priority
operand eval
LessEqual Operator "<="
int Priority
operand eval
Or Operator "|"
int Priority
operand eval



7 Project Reflection

This project was not very difficult. I could see students having trouble is starting. Without a prior knowledge of how a runTimeStack works or how a program counter in this sense works by tracking the "address" of the next bytecode to run it would be very difficult for the student to identify the proper flow of the program. Using reflection is something which I can see being very useful in the future when you don't know exactly what class will need to be instantiated based on something like user input. The lack of a unit test could make it difficult to isolate problems. This is one of those programs that you just have to jump in and hope your logic is sound and your programming is not sloppy.

8 Project Conclusion/Results

This project was overall a good experience to get a better understanding of what your computer does on a program by program basis. The results are the program runs and should have no issues assuming that the assumptions are met and the file passed in is valid. The most beneficial portion of this project was reflection because I can see it as something we will be using again and again in our professional careers.