Yan Peng
921759056
Nov 3rd, 2021

Assignment 4 Documentation

GitHub Repository

Link for assignment 4

https://github.com/sfsu-csc-413-fall-2021/assignment-4---interpreter-yuqiao1205

Project Introduction and Overview

The purpose of this project is to continue the work done on the large compiler code base and implement a functioning Interpreter. The file contains byte codes and arguments for those byte codes. Each byte code represents a specific function, the program will parse each byte code to identify their function and arguments. The program interprets a single file, reads each line of byte code, manages a stack and frames to store values that are generated by the various byte code, and advances frame pointer, then decides what action to perform, such as handling immediate values, handling jumps and calls etc. to calculate a result.

Summary of technical work: In order to implement an Interpreter. I created an abstract class Bytecode and subclasses that extends ByteCode. I created another abstract class, AddressLabelCode,which is implemented by a few of the Bytecode subclasses. I implemented the ByteCodeLoader class to load a sequence of byte codes, creating an appropriate instance of ByteCode for each line. Internally the ByteCodeLoader uses a Map from the string name of the bytecode to the required class. The ByteCodeLoader generates a Program, a sequence of ByteCode subclass instances with additional features. I implemented the Program class to resolve the addresses of Bytecodes and cross references between them. I implemented the RunTimeStack class with various methods to manage the Stack. The VirtualMachine class allows control of the machine, creation of frames, access to the stack and program counter. Each ByteCode class except for Dump class can dump itself if the dump flag in the VirtualMachine object is set to true. The program will dump the runtime stack and its frames while the program is working.

Scope of Work

Task	Completed
Test the implementation with various byte codes that test all possible cases, including the following cases:	✔
Call case | DUMP ON/OFF | function case | branch case (GOTO, FALSEBRANCH)	✔
Wrtie case | Read case | Halt case | fib.x.cod / factorial.x.cod case	✔
Return case | example case from class | BOP case(<,>,<=,>=,==,|,&)	✔
Implemented CodeTable.java:	✔
1) Read byteCodes.txt including all the possible bytecodes.	✔
2) Created a Hashmap of those ByteCodes and handled I/O exceptions.	✔
Implemented ByteCodeLoader.java	✔
1) Read each line from bytecode file and parse bytecode.	✔
2) Creat a bytecode object for each line using reflection.	✔
2) Created a Program object based on the content of bytecode.	✔
Implemented Program.java to resolve the symbolic addresses in the program:	✔
1) Stored addresses of labels in HashMap.	✔
2) Matches the label in bytecode with the label in the hashmap and sets the same address.	✔
Implemented RuntimeStack.java to record and process the stack of active frames:	✔
1) Created a Vector<Integer> that will act as the runStack.	✔
2) Created a Stack<Integer> that will manage the frames of the runStack.	✔
3) Implemented dump function to dump the current state of the RuntimeStack.	✔
4) Implementd various methods (peek(),pop(),push(),newFrameAt(),popFrame(),store(),load()to manage the runtime stack.	✔
5) Created a method getCurrentStackFrame() to check current stack frame for dump bytecode.	✔
Implemented VirtualMachine.java class with various methods that the ByteCode can be use to manage the runtime stack, control the machine, control the program counter and create new stack frames.	✔
Created a package bytecode containing the abstract class ByteCode and all the possbile ByteCode classes.	✔
Created the abstract class AddressLabelCode to help implementing bytecodes that branch.	✔
Created a help class CodeUtils to check number of arguments in each byte code class.	✔

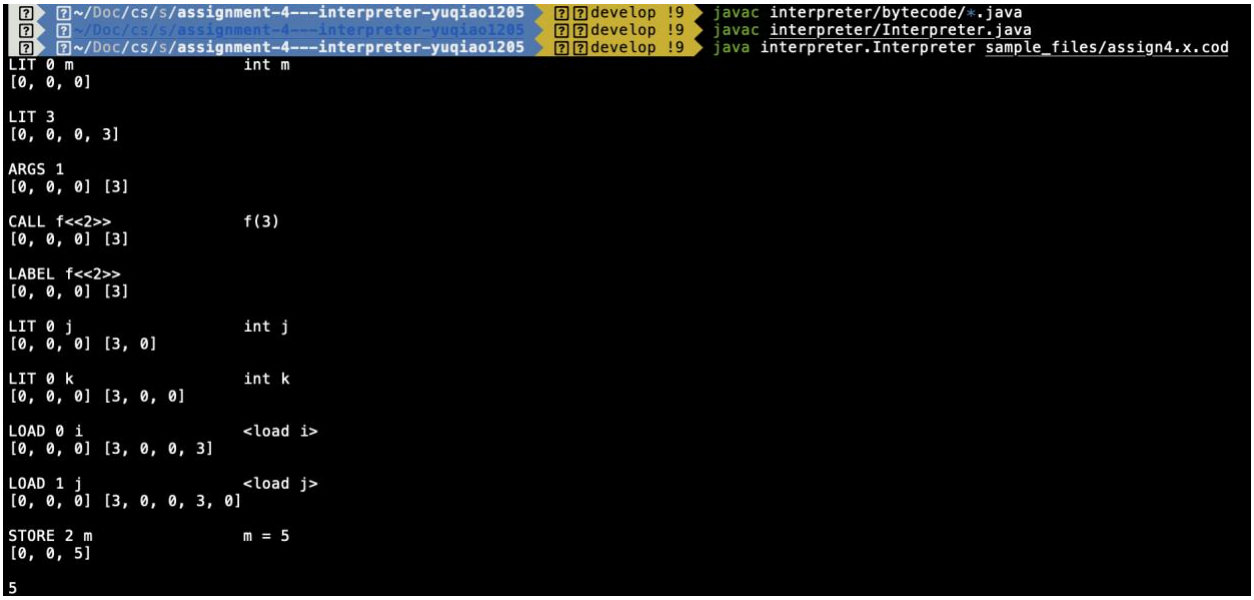
Execution and Development Environment

I developed from this code base using IntelliJ IDEA Community v. 2021.1.3. This Java application was compiled using Java JDK version 11.0.11 which is the most up to date version of the JDK.

Compilation Result

Using the instructions provided in the assignment one specification:

```
> javac interpreter/bytecode/*.java
> javac interpreter/Interpreter.java
> java interpreter.Interpreter sample_files/assign4.x.cod
```



No error messages or warnings were displayed, and the application ran as expected

Assumptions

I implemented some error handling for some byte codes case.

Implementation

ByteCodeLoader

This class read the given ByteCode file, tokenizes the string to break it into parts, grabs the correct class name for the given ByteCode from CodeTable and creates an instance of the ByteCode class corresponding to the byte code. These objects are stored in an ArrayList that is passed to a Program object.

CodeTable

The CodeTable class provides a map of all the possible ByteCode commands from a regular string to the relevant ByteCode subclass. This can be used to look up and create ByteCode objects when bytecodes are parsed from the .cod file, e.g., “HALT” => HaltCode.

Program

Implemented the Program class to resolve the addresses of ByteCode objects that implement the AddressLabelCode class. This will tell these objects what offset labels to point to when the ByteCode executes.

RunTimeStack

Implemented the RuntimeStack class to manage the runtime stack. Also, the RuntimeStack contains various methods used by bytecodes during execution.

VirtualMachine

VirtualMachine manage all of the ByteCodes actions. Some bytecodes push and pop values from the runtime stack. Other control the flow of execution by influencing the program counter or creating new stack frames. Finally the machine may be halted by calling halt().

bytecode package

This package contains all the ByteCode classes for all the possible ByteCodes, their superclass and AddressCodeLabel to distinguish bytecodes that branch.

CodeUtils

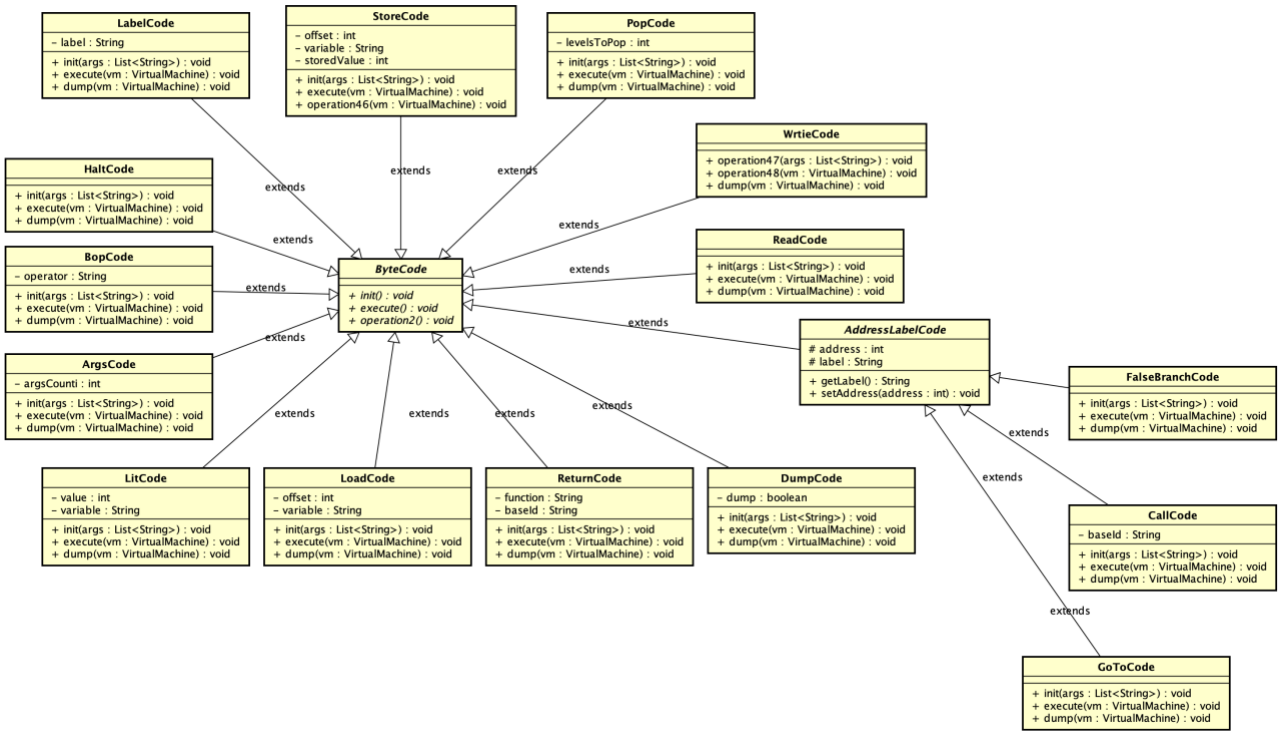
I decided to create this helper class to check the number of arguments in each byte code class.

Code Organization

To organize the code as much as possible, I put all bytecode subclass in bytecode package. Each bytecode subclass will have its own file and will be stored in the same directory including the other abstract class that distinguishes bytecode classes which need address resolution.

Class Diagram

The following class diagrams show the details of all the classes in this project, including the inheritance hierarchy



VirtualMachine

- pc : int
- runTimeStack : RunTimeStack
- returnAddresses : Stack<Integer>
- isRuning : boolean
- program : Program
- dumpFlag : boolean = false

+ VirtualMachine(program : Program)
+ executeProgram() : void
+ getRunTimeStack() : RunTimeStack
+ setDump(flag : boolean) : void
+ halt() : void
+ peekRunStack() : int
+ pushRunStack(value : Integer) : void
+ storeRunStack(offset : int) : void
+ loadRunStack(offset : Integer) : void
+ newFrameAt(offset : int) : void
+ popFrame() : void
+ pushPC(value : int) : void
+ returnPC() : void
+ savePC() : void
+ setPC(value : int) : void
+ setArgsCount(count : int) : void

RunTimeStack

- framePointers : Stack<Integer>
- runStack : Vector<Integer>
- argsCount : int

+ RunTimeStack()
+ dump() : void
+ peek() : int
+ pop() : int
+ push(item : int) : int
+ push(literal : Integer) : Integer
+ newFrameAt(offset : int) : void
+ popFrame() : void
+ store(offset : int) : int
+ load(offset : int) : int
+ setArgsCount(count : int) : void
+ getCurrectStackFrame() : List<Integer>

ByteCodeLoader

- byteCodeFile : String

+ ByteCodeLoader(byteCodeFile : String)
+ loadCodes() : Program

CodeTable

- codeTable : Map<String,String>
- byteCodesTXT : String = "interpreter/byteCodes.txt"

+ init() : void
+ mapCodeTable() : void
+ get(code : String) : String

Program

- codeList : List<ByteCode>
- addresses : Map<String,Integer>

+ Program(loadedByteCodes : List<ByteCode>)
+ getCode(programCounter : int) : ByteCode
- resolveAddrs() : void

CodeUtils

+ checkArgs(args : List<String>, expectedArgs : int, codeName : String) : void

Results and Conclusion

The Interpreter works as described and has been tested by dumping the flow of execution. I tested many cases. For example, fib.x.cod and factorial.x.cod return the correct values when I run them.

Challenges

I was overwhelmed by the amount of information I needed to digest when I started to read this assignment. It was challenging to me to implement the RunTimeStack because it is difficult to manage correctly during runtime. Especially, the popFrame and store methods. The popFrame method would not pop every value in the RunTimeStack before pushing the top value back on top even though the program calculated the correct output. The interpreter project helped me learn a lot about programming both technically and conceptually.