Code Interpreter

Michael Roger Swanson

Project Repository: github.com/csc413-02-sp18/csc413-p3-Swabisan

Introduction

This application emulates the basic functions of a compiler. The interpreter reads and analyzes bytecode files for a mock language 'x'. The .x.code files then execute various functions resembling some sort of assembly operations within a VM.

Instructions

To run this application, you will need IntelliJ/ Netbeans, or an equivalent IDE with Java compatibility along with a corresponding Java SE JDK 8 or better (older versions may work but are untested). If Netbeans is already set up on your compute you will need to download the project repository for the source files and import it into Netbeans. Once you have your IDE of choice downloaded, to run the project, you will click on run -> edit configurations, and under program arguments either enter "factorial.x.cod" or "fib.x.cod" and save. Now you should be able to run the program.

Java SE JDK 8 w/ Netbeans 8.2 Download:
http://www.oracle.com/technetwork/java/javase/downloads/jdk-netbeans-jsp-142931.html

IntelliJ Download:

https://www.jetbrains.com/idea/download

Implementation

The entry point of this program is through the Interpreter clas. There is a second package, called ByteCode, inside of ByteCode contains classes and subclasses for each bytecode that may be parsed from a source file. The package is composed of an abstract ByteCode superclass, that holds two key abstract methods that every bytecode contains, init() and execute(), init(), the following bytecodes are listed on the next page.

BYTECODE CLASSES

HaltCode: ends current program

PopCode:

```
private int valuesToPop;
```

pops *valuesToPop* amount of values from the runtime stack

FalseBranchCode:

DATAFIELDS SHARED BY THIS, GOTOCODE, AND CALLCODE

THESE THREE CLASSES ARE DEFINED BY SUPERCLASS JUMPCODE

```
private String labelName;
private int target;
```

pops from the runtimestack, if it's 0, goes to line number of target, any other number causes *this* to continue as normal to the next bytecode

GotoCode:

goes to line number of target

CallCode:

```
private String labelName;
private int target;
```

creates a function, arguments and frame determined by ArgsCode, goes to line number of target, adds current *pc* to the returnAddressStack

StoreCode:

```
private int offset;
```

pops a value from the top of the runtime stack, and adds it back in the stack, *offset* number of indices from the top

LitCode:

```
int value;
String variableName = "";
```

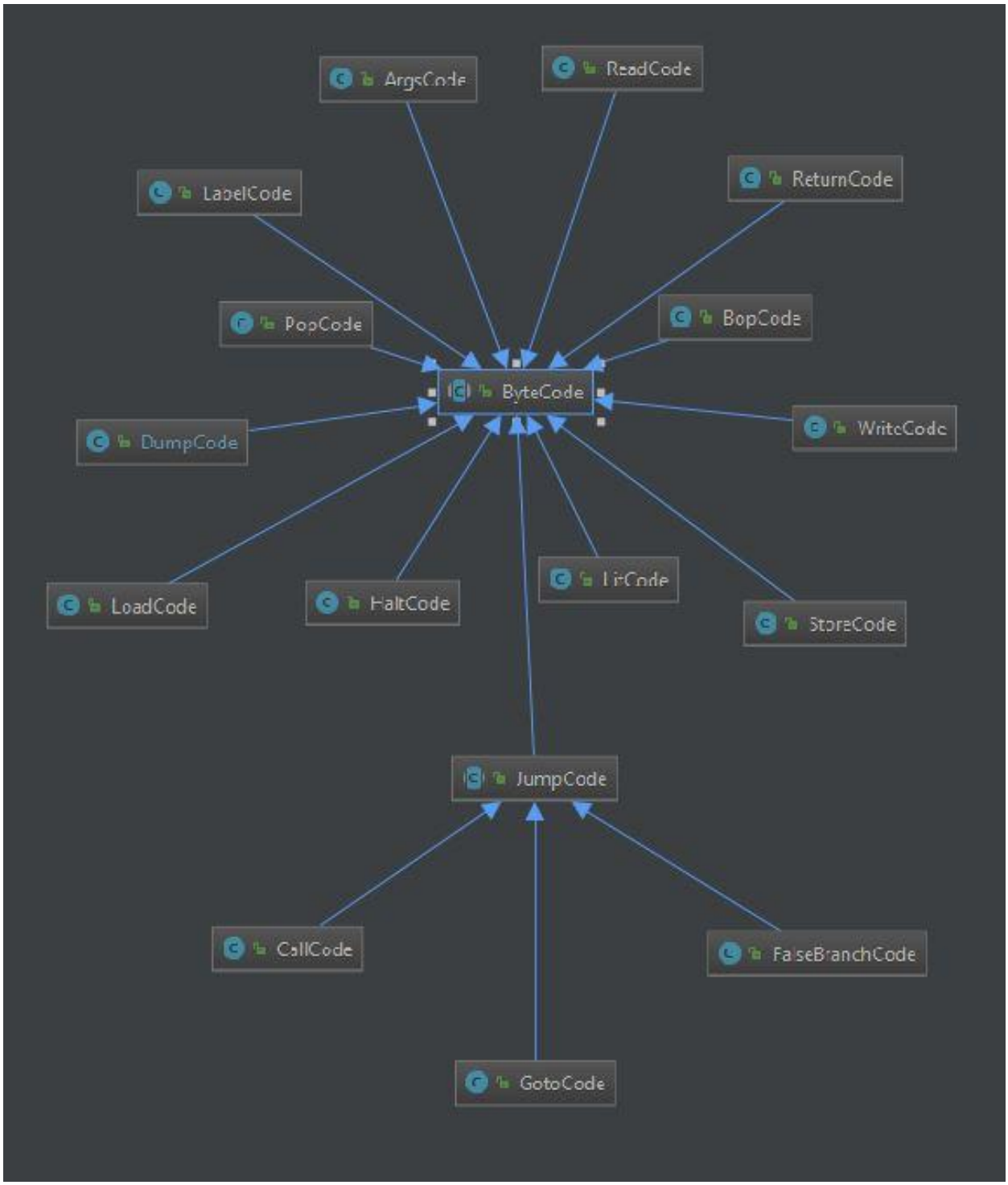
pushes *value* onto the runtime stack, if there is a *variableName* provided, attaches that name to the variable

ArgsCode:

```
private int numberOfArgs;
```

the *numberOfArgs* amount of values starting from the top of the runtime stack, are moved into a new frame that is also created *this*

CLASS DIAGRAM OF PACKAGE BYTECODE



ReturnCode:

ends the current function, pops one value from returnAddressStack, which is the *pc* of the start of the function, and goes to that number to continue on with the program

BopCode:

```
String operator;
```

if an arithmetic operator + ,- ,* ,/ ,<= ,< ,== ,!= ,>= ,> ,| ,&

ReadCode:

inputs a value defined by the user, then pushes it to the top of the runtime stack

WriteCode:

prints the value at the top of the runtime stack

LabelCode:

```
private String labelName;
```

contains variable *labelName,* which designates the line number when parsed through the method returnAddress() defined in Program class

All ByteCode tokens are defined within the CodeTable class:

```
private static HashMap<String,String> codeTable;

private CodeTable(){}

public static void init(){
    codeTable =  new HashMap<>();
    codeTable.put( k: "HALT",  v: "HaltCode");
    codeTable.put( k: "POP",  v: "PopCode");
    codeTable.put( k: "FALSEBRANCH",  v: "FalseBranchCode");
    codeTable.put( k: "GOTO",  v: "GotoCode");
    codeTable.put( k: "STORE",  v: "StoreCode");
    codeTable.put( k: "LOAD",  v: "LoadCode");
    codeTable.put( k: "LIT",  v: "LitCode");
    codeTable.put( k: "ARGS",  v: "ArgsCode");
    codeTable.put( k: "CALL",  v: "CallCode");
    codeTable.put( k: "RETURN",  v: "ReturnCode");
    codeTable.put( k: "BOP",  v: "BopCode");
    codeTable.put( k: "READ",  v: "ReadCode");
    codeTable.put( k: "WRITE",  v: "WriteCode");
    codeTable.put( k: "LABEL",  v: "LabelCode");
    codeTable.put( k: "DUMP",  v: "DumpCode");
}
```

Interpreter:

The Interpreter class is a flow-control class that calls and executes other classes within the package, ByteCodeLoader reads and parses the source file into readable bytecode class name tokens, which are all stored in the data container Program, Programs are built based on the source code loaded from ByteCodeLoader. This Program object is then fed into VirtualMachine and executes operations based on the definitions located within the ByteCode package.

In this execution within VirtualMachine, the RunTimeStack class acts as a memory structure in which the operations are based off of.