Computer Science Department - San Francisco State University

CSC 413 – Software Development – Fall 2017

**Gabriel Equitz – ID: 915254839**

Assignment 3 – Interpreter

Repository at: https://github.com/CSC-413-SFSU-02/csc413-p3-interpreter-gequitz

**Introduction**: The project consisted of writing a computer code that works as an interpreter. An interpreter is a program that reads in as input a source program, along with data for the program, and translates the source program instruction by instruction in a language machines can understand. In this project, the source program is written in bytecodes, which are executed by the virtual machine. The virtual machine has no registers for storing arbitrary values; therefore, everything must be pushed into the stack before it can be used for calculation. This project gave me background on how interpreted languages work.

The packages and their respective classes of the project are the following:

**Package name**: interpreter

Classes:

***Interpreter.java***: It contains the “main” and its functions are: to perform all the initializations, to load the bytecodes from file and to run the virtual machine.

***ByteCodeLoader.java***: Its functions are: to load the bytecodes to the program class; to build an instance of the class corresponding to the bytecode – e.g. if we read LIT 2 then we build a new LitCode instance and then add that class instance to the “Program” object, where “Program” is another class of the package. After all the bytecodes are loaded, the symbolic addresses are resolved.

***CodeTable.java***: This class is used by ByteCodeLoader. java. It contains a HashMap with keys being the bytecode strings, such as, HALT, POP, and the values being the name of the corresponding class, such as HaltCode, PopCode.

***Program.java***: This class has methods to add the bytecode instance to the program class and to resolve the address for each bytecode instance in the program class.

***RunTimeStack.java***: This class records and processes the stack of active frames. It uses the following members : stack framePointers, which maintains the stack of active frames and an ArrayList runStack which holds the actual data.

***VirtualMachine.java***: This class executes the Program.

**Package name**: interpreter.ByteCode

Classes:

***ArgsCode.java***: This function is used to instruct the interpreter to set up a new frame that includes a given number of arguments.

***Bop.java***: This class levels of the stack and perform the indicated operation.

***BranchCode.java***: This is an abstract class for branching bytecode. Superclass for CallCode, GotoCode and FalseBranchCode. It is also a subclass of ByteCode.

***ByteCode.java***: This is an abstract class for the bytecodes.

***CallCode.java***: This class transfers control to the indicated function.

***DumpCode.java***: This class determines if the instructions are dumped.

***FalseBranchCode.java***: This class pops the top of the stack; if it is false then branches to the corresponding label, otherwise it executes the next bytecode.

***GotoCode.java***: This class sets the code to jump to a given label.

***HaltCode.java***: This class terminates the program.

***LabelCode.java***: This class sets targets for the branches.

***LitCode.java***: This class when used with one argument, such as, LIT n it loads the literal value n. If it has two arguments, such as, LIT 0 i it loads 0 on the stack in order to initialize the variable i to 0 and reserve space on runtime stack for i.

***LoadCode.java***: This class pushes the value in the slot which is offset by a given number from the start of the frame onto the top of the stack.

***PopCode.java***: This class pops a given number of levels of the top of runtime stack.

***ReadCode.java***: It prompts the user for an integer value and then puts the value just read on top of the stack.

***ReturnCode.java***: This class sets the code to return from the current function.

***StoreCode.java***: This class pops the top of the stack and stores the value offset a given number from the start of the frame.

***WriteCode.java***: This class writes the value on top of the stack to output and leaves the value on top of the stack.

The diagram below shows how the classes interact with each other:

CodeTable

CallCode, FalseBranchCode,

GotoCode

ByteCodeLoader

BranchCode

ByteCode

ArgsCode,

BopCode,

DumpCode,

HaltCode,

LabelCode,

LitCode,

LoadCode,

PopCode,

ReadCode,

ReturnCode,

StoreCode,

WriteCode

Interpreter

Program

RunTimeStack

Virtual Machine

To compile the project go to the directory where the files are and type the following:

(NOTE: TO COMPILE USING THE COMMAND LINE, ONE SHOULD COMMENT OUT THE LINE THAT SAYS: “package interpreter;” and “package interpreter.ByteCode;” OTHERWISE THE COMPILATION USING THE COMMAND LINE MIGHT GIVE AN ERROR. HOWEVER IF THE COMPILATION IS DONE IN NETBEANS, THOSE LINES ARE NECESSARY.)

javac \*.java ByteCode\\*.java

to run with an input file given by factorial.x.COD:

java Interpreter ByteCodeLoader CodeTable Program RunTimeStack VirtualMachine factorial.x.COD

to create a jar file:

jar –cvfe Interpreter.jar Interpreter \*.class

To run the jar file:

java –jar Interpreter.jar factorial.x.COD