Shane Waxler

918415347

May 18, 2021

Assignment 5 Documentation

Github Repository

Link to Assignment 5 Github

Project Introduction and Overview

For this project, we implemented a debugger to work in conjunction with compilable language from our last projects. The debugger UI is presented through the command line and it implements standard features of a debugger (step, breakpoints, continue, etc.)

Scope of Work

- Implement three new bytecodes to work with a function environment record
    - Use function environment record to track local variables as well as the current function scope and state
    - Modify existing bytecodes to pass information to the function environment record
        - Make a new code table for the debugger bytecodes
- Implement a debugger as a subclass of interpreter
    - Add a debug prompt that shows whenever a breakpoint is reached and at the start of executing the program
    - Store each line of the original .x program in a vector with additional information for apt debugging
    - Add proper commands for a debugger
- Create a debugger shell which provides UI for prompting the user for debugging instructions
    - Keep track of the current line of code as well as the other important information for each line
- Implement a debugger virtual machine which inherits the functionality of the compiler's virtual machine

Execution and Development Environment

For the development and writing portion of this project I used VSCode as a text editor (no debugging) and ran the program through windows powershell. For creating the Class Diagrams I used IntelliJ's IDE.

JDK version 15.0.1

Command Line Instructions to Compile and Execute

To run this project I used the main method in Interpreter.java. For example, running through Windows command line:

```
javac interpreter/bytecode/debuggercodes/*.java;
javac interpreter/bytecode/*.java;
javac interpreter/debugger/commands/*.java;
javac interpreter/debugger/ui/*.java;
javac interpreter/debugger/*.java;
javac interpreter/*.java;
java interpreter/Interpreter -d "sample_files/factorial.x.cod"
```

Assumptions

When writing this program I assumed that the '*.x.cod' file was generated from a program that was sensible (i.e. did not cause errors within the logic of the language [i.e. no infinite loops, no unknown bytecodes, ends in a HALT]). In addition, I assumed that the user may enter invalid commands to the debugger; if they do, they are simply prompted again to enter a debugger command.

Implementation

ByteCodes/ DebuggerByteCodes (mostly copied from last assignment)

Bytecode.java is an abstract class which defines the structure for each individual bytecode (too many to reasonably list here). It has two methods. The first is execute, which takes in a Virtual Machine in order to execute runtime stack functions on the code that is being executed. The second is init, which is called for each individual bytecode in Program.java before execution of the Virtual Machine in order to pass the necessary elements to initialize each one. Each bytecode file also overrides the toString function to print more accurate and readable information.

RunTimeStack (copied from last assignment)

RunTimeStack.java is meant to model a runtime stack with a vector of integers (runStack) and a stack of integers (framePointers). Frame pointers track where each frame starts and runStack holds the actual elements. The functions defined within the file are standard for stack operations, but, store and load functions manipulate the runStack based on offsets from frame pointers. A dump function is implemented to print all the elements in their individual frames for debugging.

FunctionEnvironmentRecord

The function environment record tracks the function scope as well as its state. This includes whatever variables are contained within that scope as well as the start, end, and name of the function. It also has the ability to pop local variables from the record when prompted. The toString override method formats the local variables in a readable format.

ByteCodeLoader

ByteCodeLoader.java is implemented in order to read all the individual lines from a '*.x.cod' file and add them to an instance of program according to their bytecode class definition. This definition is found from a bytecode table which holds Strings that help create instances of the bytecodes.

Debugger

Debugger inherits logic from the interpreter; however, it also initializes a vector of entries which are used for debugging in the virtual machine. Each entry contains a line of x code, the corresponding line number, and a flag to check whether or not that line is a breakpoint.

DebuggerShell

Debugger shell is implemented to provide a UI for debugging. It asks for a command and then refers to a hashmap to see which prompt the user chose. After successfully finding a debugger command, it will execute that command.

DebuggerCodeTable

Implements the same logic as the code table from our last assignment; however, it checks first to see if the bytecode is a debuggerbytecode.

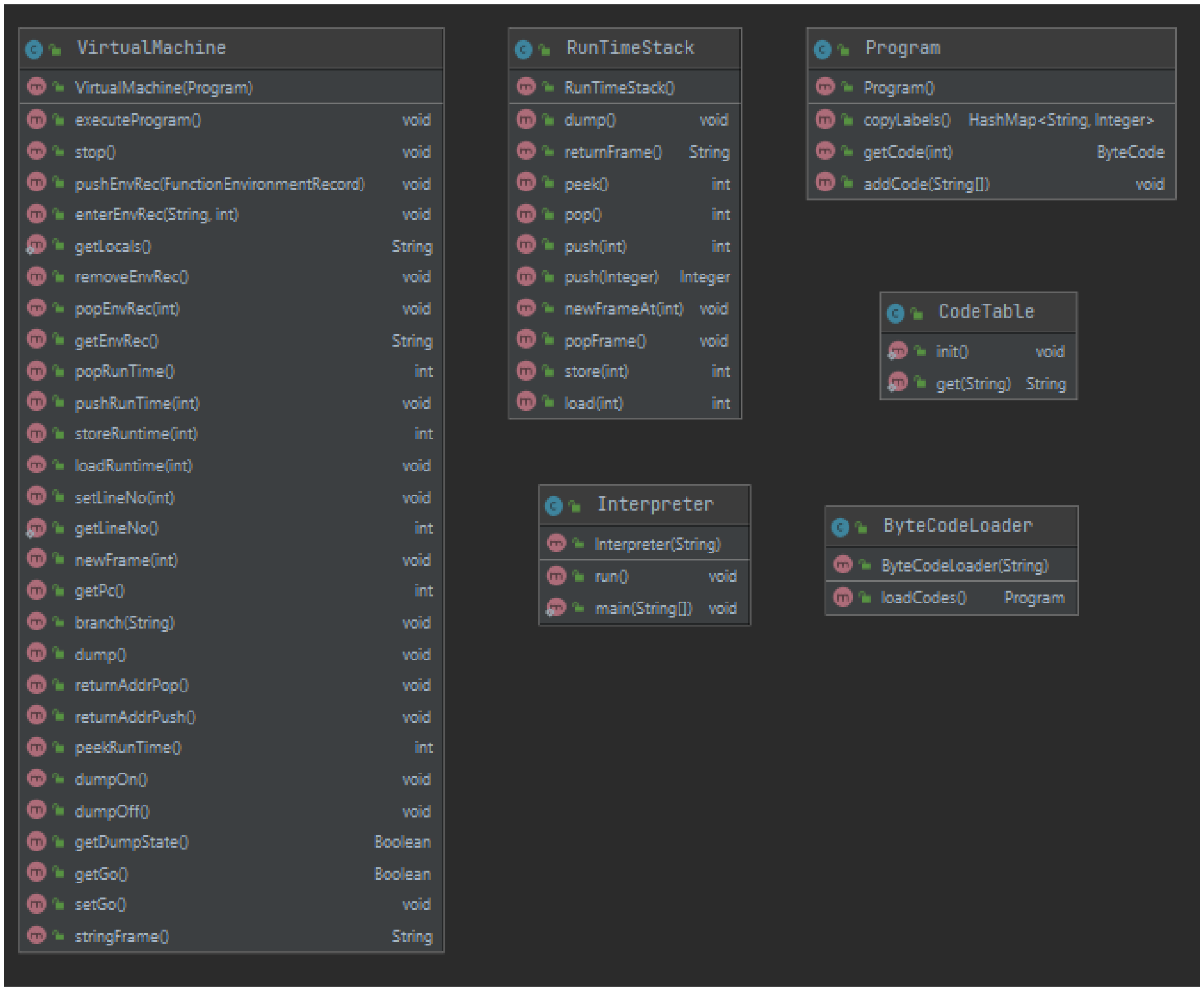
DebuggerVirtualMachine

DebuggerVirtualMachine.java inherits functionality from the previously implemented Virtual Machine. The inner logic is almost identical; however, there are a few statements to check if the current line has a breakpoint. If so, the vm will prompt the user with debugging instructions and execute the instructions.

Interpreter

Interpreter implements a main method which runs an instance of the previously defined VM. This main method takes a string argument for the file directory of the '.x.cod' file that the user would like to run. Interpreter also initializes the codetable, bytecode loader, and program (which is then passed to VM). When the -d flag is passed before the file name, the user must enter only the '.x' file, not the '.x.cod' file for debugging.

Class Diagrams



```
VirtualMachine
VirtualMachine(Program)
executeProgram()                         void
stop()                                   void
pushEnvRec(FunctionEnvironmentRecord)    void
enterEnvRec(String, int)                 void
getLocals()                              String
removeEnvRec()                           void
popEnvRec(int)                           void
getEnvRec()                              String
popRunTime()                             int
pushRunTime(int)                         void
storeRuntime(int)                        int
loadRuntime(int)                         void
setLineNo(int)                           void
getLineNo()                              int
newFrame(int)                            void
getPc()                                  int
branch(String)                           void
dump()                                   void
returnAddrPop()                          void
returnAddrPush()                         void
peekRunTime()                            int
dumpOn()                                 void
dumpOff()                                void
getDumpState()                           Boolean
getGo()                                  Boolean
setGo()                                  void
stringFrame()                            String
```

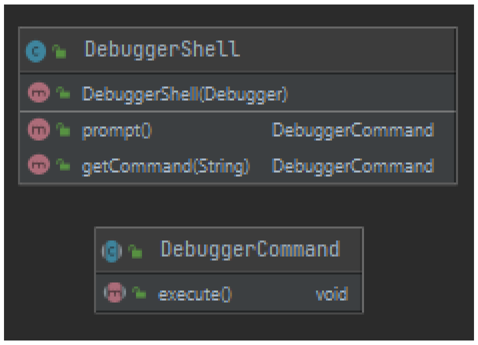
```
RunTimeStack
RunTimeStack()
dump()             void
returnFrame()      String
peek()             int
pop()              int
push(int)          int
push(Integer)      Integer
newFrameAt(int)    void
popFrame()         void
store(int)         int
load(int)          int
```

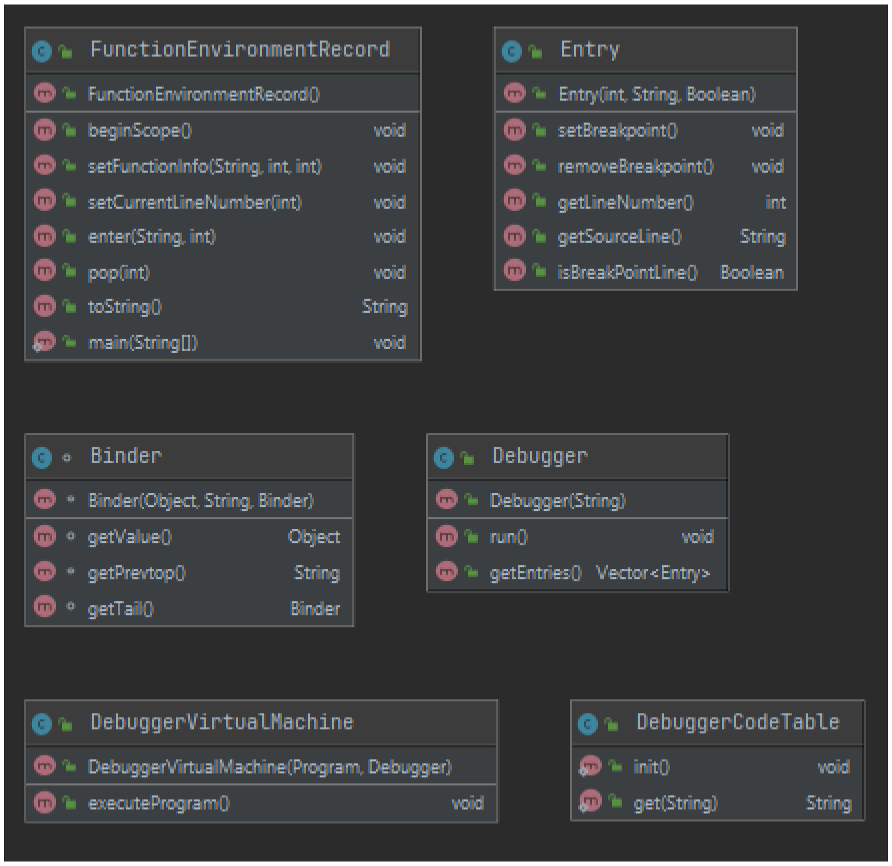
```
Program
Program()
copyLabels()     HashMap<String, Integer>
getCode(int)     ByteCode
addCode(String[])    void
```

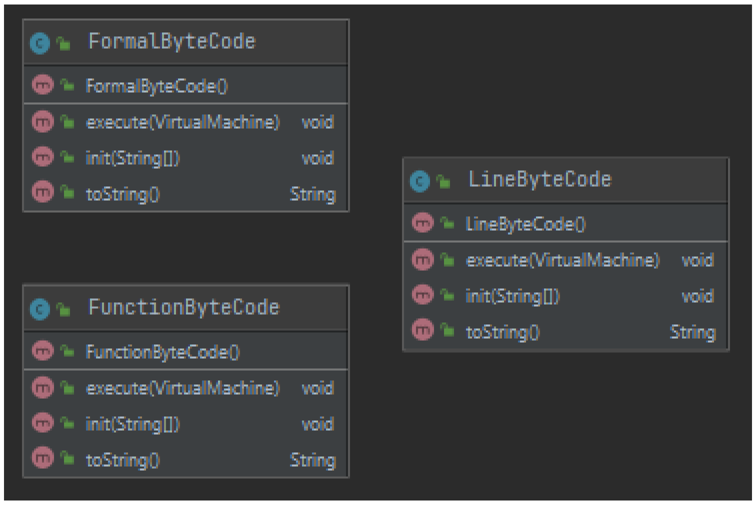
```
CodeTable
init()         void
get(String)    String
```

```
Interpreter
Interpreter(String)
run()             void
main(String[])    void
```

```
ByteCodeLoader
ByteCodeLoader(String)
loadCodes()    Program
```


DebuggerShell
DebuggerShell(Debugger)
prompt() DebuggerCommand
getCommand(String) DebuggerCommand
DebuggerCommand
execute() void


FunctionEnvironmentRecord
FunctionEnvironmentRecord()
beginScope() void
setFunctionInfo(String, int, int) void
setCurrentLineNumber(int) void
enter(String, int) void
pop(int) void
toString() String
main(String[]) void
Entry
Entry(int, String, Boolean)
setBreakpoint() void
removeBreakpoint() void
getLineNumber() int
getSourceLine() String
isBreakPointLine() Boolean
Binder
Binder(Object, String, Binder)
getValue() Object
getPrevtop() String
getTail() Binder
Debugger
Debugger(String)
run() void
getEntries() Vector<Entry>
DebuggerVirtualMachine
DebuggerVirtualMachine(Program, Debugger)
executeProgram() void
DebuggerCodeTable
init() void
get(String) String



Results and Conclusion

This project gave me a much greater understanding of how debugging works on top of a compiler. This was especially helpful for me since I rarely use a debugger when coding. In addition, it was the largest coding project I've ever worked on, so that provided a great deal of understanding how to manage big codebases.

Challenges

The biggest challenge for me in this project was simply working with such a large codebase. Since there were so many parts working in conjunction with each other, it was difficult to tell where errors were encountered. In addition, inheritance is something I was not *super* comfortable with for practical applications, so exercising that aspect of coding was difficult.

Future Work

In the future, I can imagine adding a better user interface which allows the user to place breakpoints and stop the code at any time, rather than only when a breakpoint is encountered. If no breakpoints are set on the first debugger prompt, then the program simply executes all the way through without further prompting. Having a proper DebuggerUI would make the process of debugging x code much smoother.