Marc Castro
921720147
May 18, 2022

Github Repository

https://github.com/sfsu-csc-413-fall-2022-roberts/assignment-5---debugger-mcastro16

Project Introduction and Overview

The Debugger project is to continue our work in the large compiler codebase, and implement a debugger. This debugger is responsible for processing byte code files that are created from the source code files, allowing the user to use actions to step or run code. The Debugger, Interpreter and Virtual Machine work together to run a program written in the "x" language or "x.cod" language.

Execution and Development Environment

I used the Atom IDE on my Windows Desktop to complete this assignment and tested it using the Windows Command Prompt via JDK compilation and running.

Compilation Result

Using the instructions provided in the assignment specification:
1.javac interpreter/debugger/commands/*.java
javac interpreter/bytecode/*.java
javac interpreter/bytecode/debuggercodes/*.java
javac interpreter/Interpreter.java
java interpreter.Interpreter <bytecode file>

No warnings or errors reported.

Scope of Work

- Updated ByteCodeLoader, CodeTable, Program, RunTimeStack, and VirtualMachine classes with the appropriate code for new bytecode classes and to be adaptable for debugger mode.
- Updated Skeleton codes provided to implement previous assignment work, with changes to better suit the debugger mode.
- Classes Created in package interpreter.bytecode.debuggercodes for distinct function calls:
    - LineCode
    - FunctionCode
    - FormalCode

    - DebugCallCode
    - DebugLitCode
    - DebugPopCode
    - DebugReturnCode
- Classes created/edited in debugger folder to handle debugging mode commands and processes.
    - Debugger
    - DebuggerCodeTable
    - DebuggerVirtualMachine
    - FunctionEnvironmentRecord
    - SourceLine
- Classes created/edited in debugger.ui folder for the Debugger UI that the user will see in the command line.
    - DebuggerCommand
    - DebuggerShell

Assumptions

- The implemented methods for Interpreter Class are correct, as well as the given function for the DebuggerCodeTable class.
- The given .java files and their respective code are all correct unless specified.

Implementation

ByteCodeLoader

The LoadCodes function was the only implementation needed. This function sets all the byte codes in a Vector of strings and reads the vector, adding the byte code to the program and initiating the byteCode class via the “init” function, using the vector as a parameter.

CodeTable

The CodeTable class uses a HashMap for all the byte codes that I saw in the simple.x.cod file, as well as the byte codes specified in the assignment outline.

DebuggerCodeTable

The DebuggerCodeTable class uses a HashMap for all the byte codes that were specified in the original CodeTable, but also adds new bytecode classes that are necessary for debugging only.

Debugger

The LoadCodes function was the only implementation needed. This function sets all the byte codes in a Vector of strings and reads the vector, adding the byte code to the program and initiating the byteCode class via the “init” function, using the vector as a parameter.

Program

The function that I implemented here was resolveAddresses(). This was because when I would print the addresses and the byte code, it would all be incorrect or say the same thing. This function fixes all the addresses or particular byte codes.

RunTimeStack

A total of 11 functions were implemented or altered in this class. peek() returns the first element of the vector stack. pop() removes the first element of the stack. push() takes an int item and pushes it to the runTime stack, returning the item that was just pushed. Another form of the push() function takes an Integer literal and adds it to the stack, returning the literal. The newFrameAt() function creates a new frame with the offset parameter. popFrame() pops the last element in the frame. The store() function stores the element in the stack at a specified offset. The load() function loads an element in the stack at a specified offset. The getRunStackSize() function is used to return the size of the runStack found in the RunTimeStack class to other classes. The final function I added was getValue(), which returns the value of get() function in the stack.

VirtualMachine

The VirtualMachine class executes the x program and calls the functions in the RunTimeStack class. 14 functions were created within the class. Other than the default constructor, which was given, executeProgram() is meant to create instances of the returnAddresses stack and the RunTimeStack class. The function also retrieves the ByteCode code and dumps it, iterating through the program counter. newFrameAt() calls the RunTimeStack function newFrameAt with an integer parameter. The peek() function calls the RunTimeStack function peek(). This principle also applies to the pop(), popFrame(), Push(), store(), load(), and popFrame() functions. The function setRun() sets the “isRunning” variable to true or false depending on the boolean parameter that is specified, and getRun returns the status of “isRunning”. setPC() sets the program counter in the VirtualMachine, and the getPC() function returns the PC. Finally, the pushAddrs() function pushes the integer parameter to the returnAddrs stack and popAddress() pops the top iden off of the returnAddrs stack and returns it.

DebuggerVirtualMachine

The DebuggerVirtualMachine class extends the Virtual Machine class and executes the x program and calls the functions in the RunTimeStack class if the command line specifies we are in debugging mode. 22 functions were created within the class. The default constructor establishes a bunch of stacks and arrays that contain different specifications within the environment in order to break down the source file, ultimately parsing through the source code with the readLine() functionality. setBreak sets a break in the source code so that the debugger can stop there if the user enters “continue”. clearBreak() clears the break container. sourceSize() returns the size of the source code. runTrue() sets the “isRunning” variable to true. stepIn() sets the “stepin” variable to true. executeProgram() is meant to create instances of the new debugger codes specified in the outline if they appear in the source code. The function also retrieves the ByteCode code and iterates through the program counter. debugPop() pops the top of the function record. getCode() returns the program’s bytecode at the given program counter. end(0 pops the top of the function environment record stack. stackPeek() allows for checking the top of the environment stack. setFunct()

sets the name, start, and end of a function. getFunctStackSize() returns the function stack size. getEnd() gets the end of the function if it is not empty. funcRecEnter() enters new function records into the function environment record. currentLine() sets the FER's current line number. getStart retrieves the start of the function. getName() returns the FER's name. pushfuncRecordIntoStack pushes the FER into the environment stack, creates a new instance of an FER, and begins the scope again. getCurrent gets the current int of the function. displayVar displays the variable from the run stack.

FunctionEnvironmentRecord

The Function Environment Record is used to track the current function state in the debugger. It consists of 14 functions. The default constructor just establishes a new table class that I created to help with organization, a new string name, and sets the start, end, and current line all equal to 0. beginScope() just runs the respective table class "beginScope()" function. This also applies to endScope() and pop(). setCurrentLineNumber sets the current line to the given int variable. setFunctionInfo sets the name, startline, and endline to the given parameters. getKey() gets the table keys. getName() retrieves the name of the FER. This property also applies to retrieving variables for getStart(), getEnd() and getCurrent(). getOffset() returns the integer form of the return value from the table's get() function. Finally, enter() puts the variable and value in the table.

SourceLine

SourceLine is a class I created that was not part of the requirements but proved extremely helpful. Essentially the class had 4 functions. SourceLine() set the sourceLine variable and the isBreakptSet variable to their respective parameters. BreakptSet() set the variable on whether or not a breakpoint was set at the source line. getSourceLine() returned the sourceLine. Finally, breakptGet() returned the break point. This class mostly helped out when working with the set functionality of the debugger.

ByteCode Classes:

ArgsCode

Used prior to calling a function.

BopCode

Pops top two levels of the stack and performs the specified operation.

ByteCode

Initializes the functions for the byte codes.

CallCode

Transfers control to the specified function call.

DumpCode

Used to set the dumping state in the VirtualMachine class.

GotoCode

Initializes execution.

HaltCode

Stops execution.

LabelCode

Targets for branches.

LitCode

Loads the literal value.

LoadCode

Pushes the value in the position which is the offset from the start of the frame onto the stop of the stack.

PopCode

Pops the top of the stack.

ReadCode

Prompts the user to input an integer, reads the integer, and pushes the value to the stack.

ReturnCode

Returns the current function.

StoreCode

Pops the top of the stack and stores that value into the offset from the start of the frame.

WriteCode

Writes the value of the top of the stack to the output.

document continues below →

DebuggerCode Classes:

DebugCallCode

Extends the normal Interpreter Call Code. Implements an execute function that calls the Debugger Virtual Machine function that pushes the function record into the stack.

DebugLitCode

Extends the normal Interpreter Lit Code. Implements Debugger Virtual Machine Functionality that calls Debugger Virtual Machine function that enters a new function record.

DebugPopCode

Extends the normal Interpreter Lit Code. Implements Debugger Virtual Machine Functionality that just executes the Debugger Virtual Machine.

DebugReturnCode

Extends the normal Interpreter Return Code. Implements Debugger Virtual Machine Functionality to run the end command in the DVM.

FormalCode

Runs the DVM function that enters a new function record.

FunctionCode

Traces the code to read for the declaration of the function, finding the start, and finding the end.

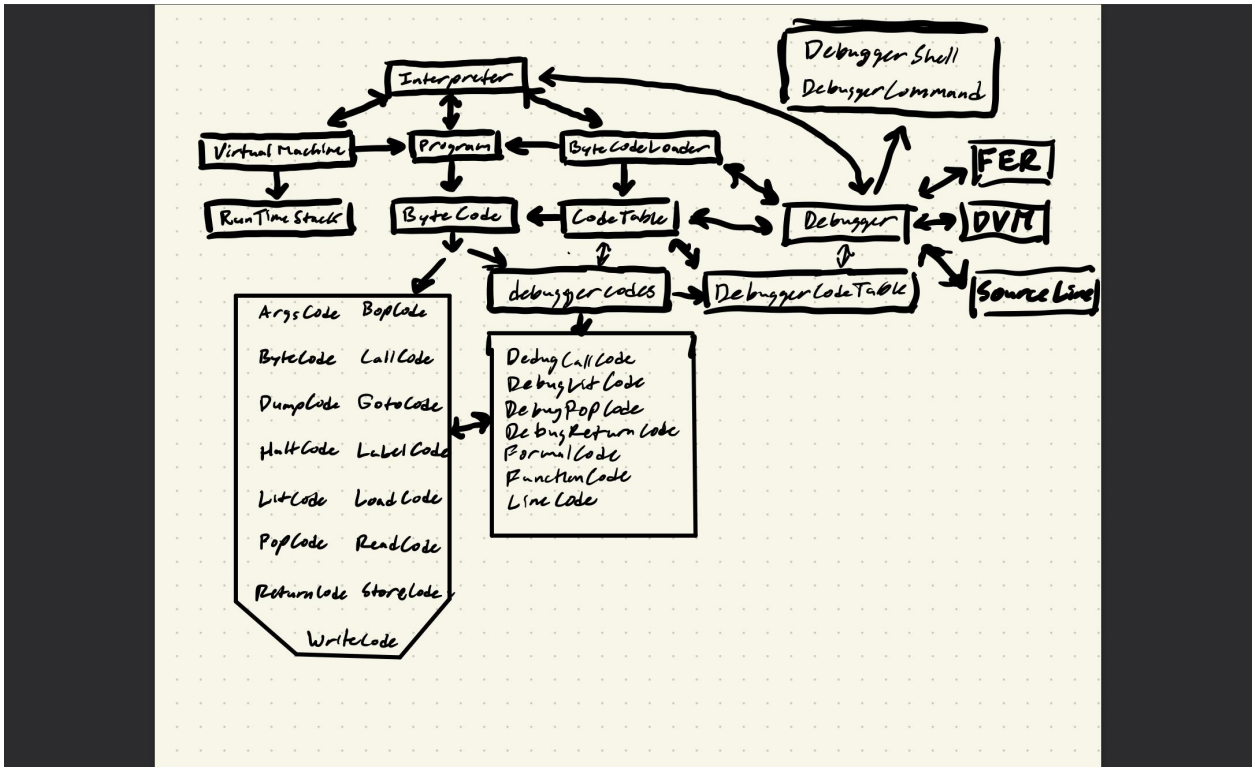
LineCode

Sets the current line of the program and makes said line position available for retrieval by other processes.

WriteCode

Writes the value of the top of the stack to the output.

Class Diagram



Results and Conclusion

```
C:\Users\Majocast\Desktop\SPRING2022\CSC413\assignment-5---debugger-mcastro16>java interpreter.Interpreter -d sample_files/factorial
GoToCode
LabelCode
debuggercodes.LineCode
debuggercodes.FunctionCode
ReadCode
debuggercodes.DebugReturnCode
LabelCode
debuggercodes.LineCode
debuggercodes.FunctionCode
debuggercodes.FormalCode
LoadCode
WriteCode
debuggercodes.DebugReturnCode
LabelCode
debuggercodes.LineCode
debuggercodes.FunctionCode
debuggercodes.DebugLitCode
debuggercodes.DebugLitCode
GoToCode
LabelCode
debuggercodes.LineCode
debuggercodes.FunctionCode
debuggercodes.FormalCode
debuggercodes.LineCode
LoadCode
debuggercodes.DebugLitCode
BopCode
FalseBranchCode
debuggercodes.LineCode
debuggercodes.DebugLitCode
debuggercodes.DebugReturnCode
debuggercodes.DebugPopCode
GoToCode
LabelCode
debuggercodes.LineCode
LoadCode
LoadCode
debuggercodes.DebugLitCode
BopCode
ArgsCode
debuggercodes.DebugCallCode
BopCode
debuggercodes.DebugReturnCode
debuggercodes.DebugPopCode
LabelCode
debuggercodes.DebugPopCode
debuggercodes.DebugLitCode
debuggercodes.DebugReturnCode
LabelCode
LabelCode
debuggercodes.LineCode
debuggercodes.DebugLitCode
debuggercodes.DebugLitCode
BopCode
FalseBranchCode
debuggercodes.LineCode
ArgsCode
debuggercodes.DebugCallCode
ArgsCode
debuggercodes.DebugCallCode
ArgsCode
debuggercodes.DebugCallCode
```

documentation continues below →

```
LabelCode
debuggercodes.DebugPopCode
HaltCode
program {boolean j int i
  int factorial(int n) {
      if (n < 2) then
         { return 1 }
      else
         {return n*factorial(n-1) }
  }
  while (1==1) {
      i = write(factorial(read()))
  }
}
1:  program {boolean j int i
2:    int factorial(int n) {
3:        if (n < 2) then
4:           { return 1 }
5:        else
6:           {return n*factorial(n-1) }
7:    }
8:    while (1==1) {
9:        i = write(factorial(read()))
10:    }
11:  }
Type ? for help
>
?
Command:        Use:
set             sets breakpoints
source          prints code
continue        continues
exit            exits
locals          display variables
step            steps into
list            shows breakpoints
Type ? for help
>
```

```
[2] Type ? for help
>
source
1:  program {boolean j int i
2: *   int factorial(int n) {
3:        if (n < 2) then
4:           { return 1 }
5:        else
6:           {return n*factorial(n-1) }
7:    }
8:    while (1==1) {
9:        i = write(factorial(read()))
10:    }
11:  }
Type ? for help
>
locals
Type ? for help
>
set 5
Error could not set breakpoints
Type ? for help
>
list
Breakpoints set at:
2
2 5
Type ? for help
>
step
1:  program {boolean j int i
2: *   int factorial(int n) {
3:        if (n < 2) then
4:           { return 1 }
5:        else
6:           {return n*factorial(n-1) }
7:    }
8:    while (1==1) {
9:        i = write(factorial(read()))
10:    }
11:  }
Type ? for help
>
step
input number 1
1:  program {boolean j int i
2: *   int factorial(int n) {
3:        if (n < 2) then
4:           { return 1 }
5:        else
6:           {return n*factorial(n-1) }
7:    }
8:    while (1==1) {
9:        i = write(factorial(read()))
10:    }
11:  }
Type ? for help
>
locals
n=0
Type ? for help
>
```

Output for factorial.x.cod

Reflection

This project was hands down the hardest project I had to do. Although we were already given the opportunity to use the Interpreter code from Assignment 4. I realized extremely quickly that a lot of my functionality was centered around either dumping or functionalities that the debugger requirements would not be happy with. The implementation of the debuggercodes was relatively easy considering it was outlined what the functionalities would be. I had a lot of fun solving it though, it felt very rewarding

getting it done for such a large scale solo project. This project also makes me happy the semester is over because my brain is fried.

Challenges

This project was definitely a challenge time wise. I spent a lot more time working on it than expected, mostly due to the amount of functionalities that were required for the debugger alone. On top of that, reading through the requirements seemed doable but ended up being extremely complex due to the nature of the debugger. I had to go back into my old code as well to account for the instances that the debugger mode was instantiated. My main challenge was in the breakpoint setting. I had a lot of problems with let alone getting the break point to be set in the function. When I got that solved, I struggled just showing the breakpoint at all. I also had to record the DebuggerVirtualMachine entirely. Granted, for some of it it felt like a carbon copy of the original Virtual Machine, but I had to account for the fact that the new byte codes were in their own folders, as well as how to handle my new breakpoints. There were also plenty of functions that I realized I could not just have run all their requirements over and over in other classes because that would just be tedious, so I created many functions in the DVM that covered a lot of ground.

Future Work

Some future work could be to expand the functionality of the Debugger to read through more bytecode types. I would also change the interface because even though it is simple and fulfills the requirement of what should be outputted with each functionality. It could be a lot nicer and maybe centered in the console? It would be a bunch of aesthetic changes that would make it nicer to look at. I would also change the code to account for double instances of functions because I feel as though I copy and pasted a lot of my functions in multiple classes just so that they would be included and to get over some of the errors I was facing.