Dhruv Shah

CSC 413 SPRING 2018

Assignment 3 – The Interpreter

GitHub Link: https://github.com/csc413-02-sp18/csc413-p3-dhruvbshah.git

The Interpreter: Sample output of fib.x.cod for integer 2

```
run:
GotoCode 8
LIT 0 x
GotoCode 49
LIT 0 k
LIT 5
STORE 0 x   x=5
ARGS 0
CALLRead
READ
Please input an integer: 2
RETURN
ARGS 1
CALLfib<<2>>
LOAD 0 n   <load n
LIT 1
BOP <=
FALSEBRANCH 20
LOAD 0 n   <load n
LIT 2
BOP ==
FALSEBRANCH 29
LIT 1
RETURNfib<<2>>
ARGS 1
CALLWrite
LOAD 0 dummyFormal   <load dummyFormal
WRITE
1
RETURN
STORE 1 k   k=1
LIT 0 x
LIT 7
STORE 2 x   x=7
LIT 8
STORE 2 x   x=8
POP 1
POP 2
HALT
BUILD SUCCESSFUL (total time: 2 seconds)
```

SUMMARY:

This assignment is an interpreter for a mock language X. Given a set amount of base code, I implemented an abstract byte code class, their individual abstract classes, a byte code loader, resolved the loaded bytecodes' address and a virtual machine that uses an array list, which will be treated as a runtime stack. To summarize, when the Interpreter runs, it will load all the bytecodes from a x.cod file, generating all the bytecodes and resolving them to their correct addresses, which will be passed to the virtual machine and each bytecode will be executed accordingly. The result will be a small program,

prompting the user for an input and returning a factorial or a Fibonacci number. This was all completed on the NetBeans IDE.

How to Use The Project:

- Open the project in Net Beans 8.2 for best result
- The user must clone the repo and set up as a project with existing source code. Then set up the working directory by right clicking on the newly created project Set Configuration -> Run -> Working Directory
- Then select the cloned repo and continue
- The user must use the factorial.x.cod or fib.x.cod in order to execute the fully functioning interpreter.
- Afterwards, the program should compile and execute using the play button and it will prompt the user to enter a number.

Assumptions:

- First assumption was that the Interpreter.java file and the CodeTable.java file did not need any changes.
- The RunTimeStack and the FramePointerStack both run simultaneously.
- I was not allowed to break encapsulation, meaning everything must go through the Virtual Machine I implemented. The program uses integers to calculate the Factorial or Fibonacci output. It is also assumed the end user should not input large numbers and overflow the stack.

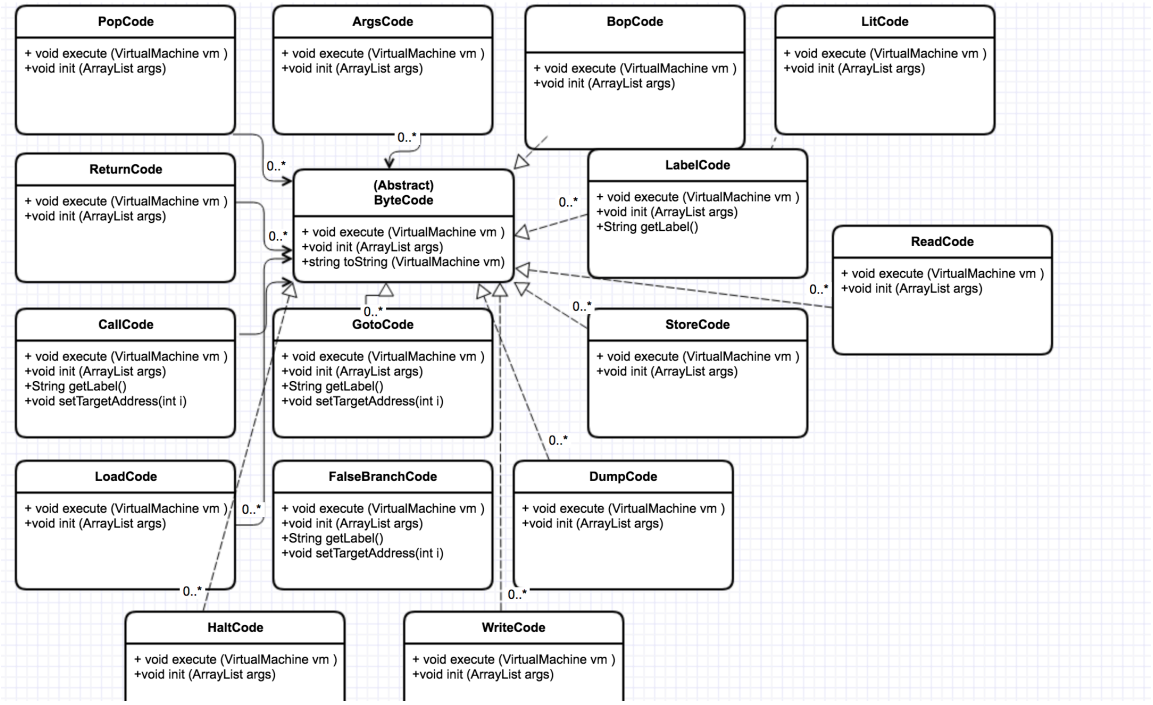
Difficulties:

- In the beginning I was confused on where to start the assignment and then I moved by just creating all the bytecodes
- One of the major difficulties I faced was that my fib.x.cod was working perfectly fine but I was getting error on my factorial.x.cod. It executes the factorial.x.cod correctly but the build was failing at end because my ArrayList was going out of bound. So in order to over come this I created public int getN().

```
public int getN(){
    return runTimeStack.size() - framePointer.peek() - 1;
}
```

I used getN() to check if my ArrayList is not going out of bound.

UML Class Diagram:



Implementation Process:

ByteCode Classes:

Before loading all the bytecodes, they must be created using an abstract parent class. The rest of the bytecodes will be generated accordingly later. The dashed arrays mean they extend from Bytecode and must implement its methods, but can also implement non-abstract methods.

Each byte code class has their own individual functions that were implemented after correctly implementing the Virtual Machine class with methods to access the RunTimeStack class.

Description of Each ByteCode Classes:

Bytecode	Description	Examples
HALT	*halt* execution	HALT
POP	*POP n*: Pop top *n* levels of runtime stack	POP 5 POP 0
FALSEBRANCH	*FALSEBRANCH <label>* - pop the top of the stack; if it's *false* (0) then branch to <label> else execute the next bytecode	FALSEBRANCH xyz<<3>>
GOTO	*GOTO <label>*	GOTO xyz<<3>>
STORE	*STORE n <id>* - pop the top of the stack; store value into the *offset n from the start of the frame*; <id> is used as a comment, it's the variable name where the data is stored	STORE 2 i
LOAD	*LOAD n <id>*; push the value in the slot which is *offset n from the start of the frame* onto the top of the stack; <id> is used as a comment, it's the variable name from which the data is loaded	LOAD 3 j
LIT	*LIT n* - load the *literal value n* *LIT 0 i* – this form of the Lit was generated to load 0 on the stack in order to initialize the variable *i* to 0 and reserve space on the runtime stack for *i*	LIT 5 LIT 0 i
ARGS	*ARGS n* ;Used prior to calling a function: n = #args this instruction is *immediately followed* by the *CALL* instruction; the function has *n args* so *ARGS n* instructs the interpreter to set up a new frame *n down from the top*, so it will include the arguments	ARGS 4
CALL	*CALL <funcname>* - transfer control to the indicated function	CALL f CALL f<<3>>
RETURN	*RETURN <funcname>* ; Return from the current function; *<funcname>* is used as a comment to indicate the current function *RETURN* is generated for intrinsic functions	RETURN f<<2>> RETURN
BOP	*bop <binary op>* - *pop top 2 levels* of the stack and perform indicated operation – operations are + - / * == != <= > >= < | & *| and & are logical operators, not bit operators* lower level is the *first operand*: e.g. *<second-level> + <top-level>*	BOP +
READ	*READ* ; Read an integer; prompt the user for input; put the value just read on top of the stack	READ
WRITE	*WRITE* ; Write the value on top of the stack to output; *leave the value on top of the stack*	WRITE
LABEL	*LABEL <label>;* target for branches; (see *FALSEBRANCH, GOTO*)	LABEL xyz<<3>> LABEL Read

ByteCodeLoader.java:

Given a set code table, a hash map used to grab the correct byte code classes, I implemented the ByteCodeLoader (BCL) class first, BCL for short. The BCL reads the fib and factorial files line by line using the BufferedReader and FileReader libraries.
A string tokenizer was used to tokenize each string read. The first token is assumed to always be a byte code class type within the code table.

```
public Program loadCodes() throws IOException {
    Program program = new Program();

    ArrayList<String> args = new ArrayList<>();

    String code = byteSource.readLine(); //reads line from the program
```

Furthermore, using simple java reflection, I created a byte code class instance for every line read.

```
ByteCode byteCode = (ByteCode)
        (Class.forName("interpreter.ByteCode."+codeClass).newInstance());
```

If the string read is not fully tokenized by the tokenizer, the BCL will then parse more tokens for that specific line. These will be the arguments of the specific byte code instance. They will be passed to their respective bytecode classes.

```
while (code != null){

    StringTokenizer token = new StringTokenizer(code);
    args.clear(); //clears argument


    String storedToken = token.nextToken();
    if (storedToken != null){

        String codeClass = CodeTable.getClassName(storedToken);
```

Afterwards, these bytecodes are then stored into an ArrayList named program. Before exiting out of the BCL, the bytecodes' address will be resolved in the Program class.

CodeTable.le.java:

```
package interpreter;

import java.util.HashMap;

public class CodeTable {

    private static HashMap<String,String> codeTable;

    private CodeTable(){}

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

    /**
     * A method to facilitate the retrieval of the names
     * of a specific byte code class.
     * @param key for byte code.
     * @return class name of desired byte code.
     */
    public static String getClassName(String key){

        return codeTable.get(key);

    }
}
```

Program.java:

Right after the bytecodes are stored, another hashmap is created to resolve addresses later. The hashmap will only store instances of the LABEL class. It will store the string argument of LABEL as the key, and its index as the value.

```
public void add(ByteCode byteCode){
    if (byteCode instanceof LabelCode){
        LabelCode label = (LabelCode) byteCode;
        labelList(label.getLabel(), program.size());
    }
    program.add(byteCode);
}
```

Resolving addresses involve looping through the bytecodes stored in Program, checking if one of the following codes contains a label, GotoCode, FalseBranchCode, and CallCode. It will also set the address to the one found in the hashmap.

```
public void resolveAddrs(Program program) {

    for(int i = 0; i < program.getSize() - 1; i++) {

        if(program.getCode(i) instanceof GotoCode) {
            GotoCode code = (GotoCode) program.getCode(i);
            code.setTargetAddress(labelList.get(code.getLabel()));
        } else if(program.getCode(i) instanceof FalseBranchCode) {
            FalseBranchCode code = (FalseBranchCode) program.getCode(i);
            code.setTargetAddress(labelList.get(code.getLabel()));
        } else if(program.getCode(i) instanceof CallCode) {
            CallCode code = (CallCode) program.getCode(i);
            code.setTargetAddress(labelList.get(code.getLabel()));
        }
    }
}
```

Resolving addresses will do the following:

```
1. GOTO addr            GOTO 5
2. LOAD 3               LOAD 3
3. LIT 2        ==>     LIT 2
4. STORE 4              STORE 4
5. LABEL addr           LABEL addr
```

RunTimeStack.java:

The RunTimeStack class maintains the stack of active frames. When using the CALL bytecode, the runtime stack will push new frames onto an ArrayList, which will be treated as a stack. The VirtualMachine class will use this class.

```
public RunTimeStack()
{
    runTimeStack = new ArrayList<Integer>();
    framePointer = new Stack<Integer>();
    //Add initial Frame Pointer, main is the entry
    // point of our language, so its frame pointer is 0.
    framePointer.add(0);
}
```

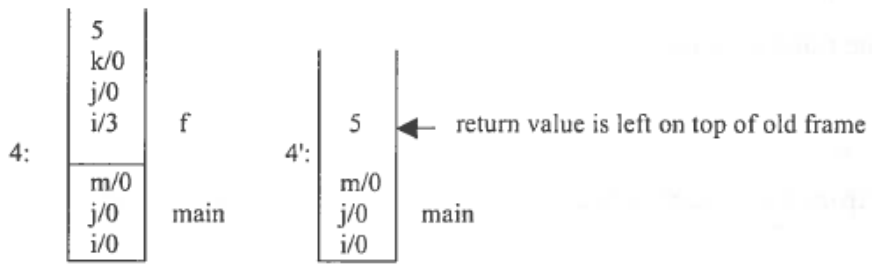
Using ArrayList built in methods, I created basic methods that would treat it as if it was a stack.

```
public int pop() {
    int temp = (int) runTimeStack.size() - 1;
    return (int) runTimeStack.remove(temp);
}
```

Other than stack pop, push, peek function, I had to implement popping the frame, when to add a new frame, load and store method. This is essential to represent how a stack operates.

The store method sets a value on the stack according to some *n* from the offset and the load method loads a value *n* from the offset and puts it to the top of the stack. STORE and LOAD bytecode class instead of implementing them directly in their class will call these two-implemented methods.

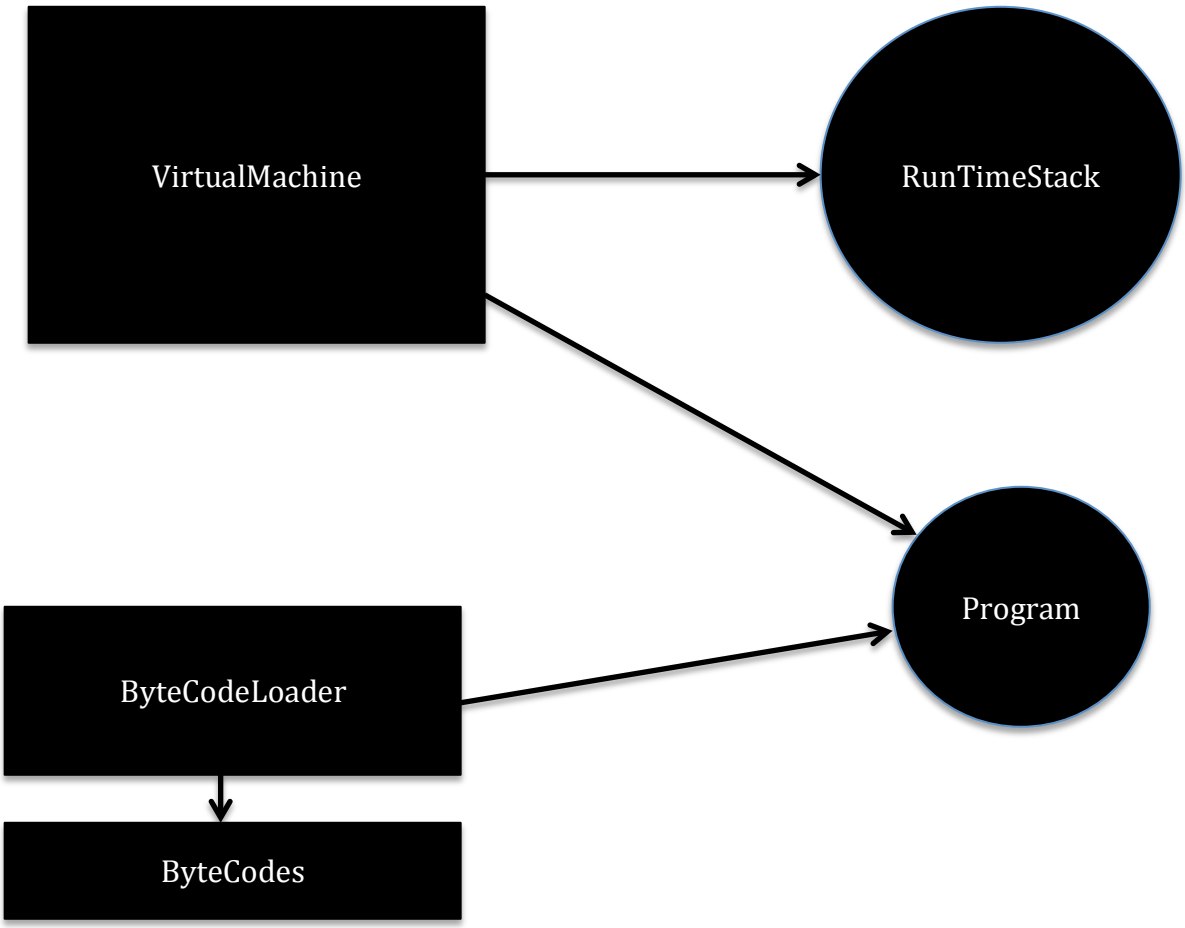
Popping the frame means the stack will be popped until it reaches the same index as the frame that was popped. In the figure below, the frame is at the 3rd index and will be popped



VirtualMachine.java:

The virtual machine is where everything happens. It uses every class that was implemented to run the program.

VM Class Diagram:



Though this is extremely simplified, the VirtualMachine uses the Program class to gain access to the bytecodes loaded in by the BCL. The ByteCode class containing all the bytecodes should be all initialized already to avoid running errors.

The virtual machine can then execute the bytecodes by looping through them.

```
while(isRunning) {
    ByteCode code = program.getCode(pc);
    System.out.println(code.toString(this));
```

The VirtualMachine contains methods used to access the RunTimeStack class to avoid breaking encapsulation. This is because the VirtualMachine owns everything. Thus, the bytecodes accessed stack operations through the VirtualMachine.

```
public int popRunStack(){
    return runStack.pop();
}
```

nes added

For a function to complete, the CallCode class kept track it's the call function address by pushing onto the ReturnAddress stack in the VirtualMachine.

When Return is called it will pop off the ReturnAddress stack and reset the program counter to continue through the program.

Dump:

Dumping means to just print all the frames on the stack based on if the DumpCode class reading an "ON" or "OFF". This will flip a switch, a getter and a setter, and will cause all bytecodes to continue printing from the stack until the virtual machine reads an "OFF".

```
public void dumpSwitch(String flag){
    dumpMode = flag;
}
```

Each bytecode has their own implementation of how they dump by printing their class name and their arguments, with some bytecodes containing special functions as to how they are dumped.

The figure below shows the special dump action for the LIT bytecode.

```
    //verify if dumpMode is on
    if("ON".equals(vm.dumpMode)) {
        String output = "LIT " + n + " " + id;
        if(!id.equals("")){
            output = output + "    int " + id;
        }
        System.out.println(output);
    }

}
```

Conclusion:

One of the toughest projects I have done in my academic career so far.

This is one of the largest learning curves from a project. There was a lot to keep track of before even trying to make the program start running at all.

This was a bit refreshing to use getters and setters after a while of not using them in roughly a year.

I am still learning to import properly because I feel like if I do, then I am doing something wrong, but it is actually the opposite.

It was difficult to make some work around and a lot of grinding to figure out what was a more optimal solution. I should probably draw out and practice more on pseudocode before diving into the problem as this helped solve my problem with printing the dump codes.

Using class method in conjunction with a hashmap sure helped a lot to solve problems when I could not actually retrieve private data fields

A lot of teamwork and a lot of logical discussing of the code were needed to get through this assignment.

People I collaborated with (helped each other):

- JED AHMDIA
- ADITYA SHEORAN

References:

https://stackoverflow.com/questions/12277091/what-is-the-meaning-of-thread-dump

http://doc.pypy.org/en/latest/interpreter.html

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html