The Interpreter Project

CSC 413 - Software Development

Anthony J Souza

Sourced from Dr. Levine

Computer Science
San Francisco State University
September 2019

Contents

1 Introduction

For this assignment you will be implementing an interpreter for the mock language X. You can think of the mock language X as a simplified version of Java. The interpreter is responsible for processing byte codes that are created from the source code files with the extension x. The interpreter and the Virtual Machine (this will be implemented by you as well) will work together to run a program written in the Language X. The two sample programs are a recursive version of computing the nth Fibonacci number and recursively finding the factorial of a number. These files have the extension x.cod.

And as always, if there are any questions please ask them in slack AND in class. This promotes collaborative thinking which is important. NOTE THIS IS AN INDIVIDUAL ASSIGNMENT but you can collaborate with other students.

2 The Interpreter

In the following sections you will find coding hints, class requirements, and other information to help you implement and complete the interpreter project. Please read these pages a few times before you begin coding.

2.1 Frames (Activation Record) and the Runtime Stack

Frames or activation records are simply the set of variables (actual arguments, local variables, and temporary storage) for each function called during the execution of a program. Given that function calls and returns happen in a LIFO fashion, we will use a Stack to hold and maintain our frames.

For example, if we have the following execution sequence:

```
Main
    Call A from Main
        Call B From A
```

With the sequence above, we will get the following runtime stack(3):

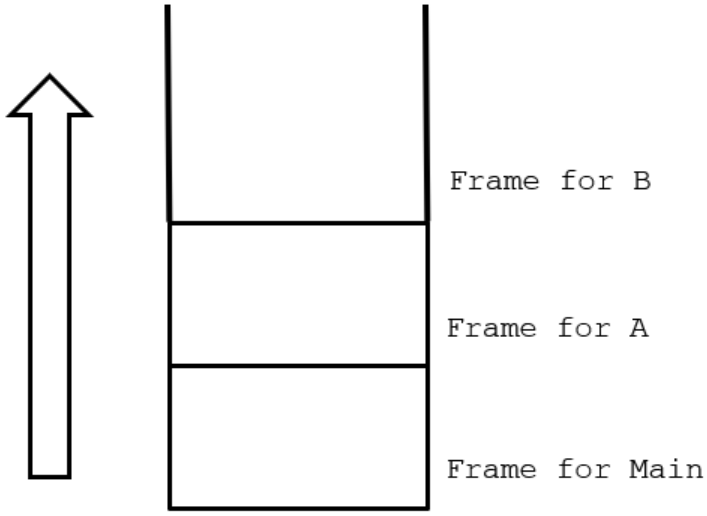

Figure 1: View of runtime stack as methods are called.

Now if we take this concept and apply it to some code written in the Language X we can see a better example of our runtime stack.

```
program {  int i int j
    int f ( int i ) { 2
        int j int k 3
        return i + j + k + 2 4
    }
int m 1
m = f(3) 5
i = write(j+m)
}
```

Figure 2: Small code sample in the Language X.

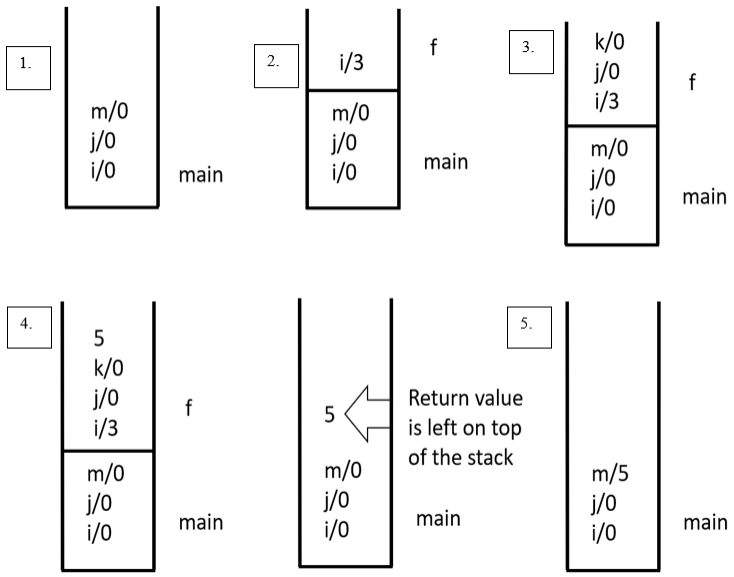

Figure 3: View of Runtime Stack as code is executed.

3 Supported ByteCodes

Bytecode	Description	Example
HALT	Halt the execution of a program	HALT
POP	POP n: pop the top n levels of the runtime stack	POP 5 POP 0
FALSEBRANCH	FALSEBRANCH <label> pop the top of the stack; if it is false (0) then branch to <label> else execute the next bytecode	FALSEBRANCH xyz<<3>>
GOTO	GOTO <label>	GOTO zyx<<3>>
STORE	STORE N <id> - pop the top of the stack; store the value into the offset n from the start of the frame; <id> is used as a comment and for dumping, it's the variable name where the data is stored.	STORE 3 i STORE 2
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 and for dumping, it's the variable name where the data is loaded.	LOAD 3 LOAD 2 i
LIT	LIT n – load the literal value n LIT 0 i – this form of Lit was generated to load 0 on the stack to initialize the variable i to the value 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 = # of 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 of the runtime stack. This will include the arguments in the new frame for the function.	ARGS 4 ARGS 0 ARGS 2
CALL	CALL <funcname> - transfer control to the indicated function. Make sure to save where the function should return to.	CALL f CALL f<<3>> Note: CALL f and Call f<<3>> are executed in the same way.

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 Note: returns with labels functions EXECUTE THE same as returns without labels.
BOP	BOP <binary op> , pop top 2 levels of the stack and perform the indicated operation , operations are +, −, /, ∗, ==, ! =, <=, >, >=, <, |, &. The operators | and & are logical operators not bitwise operators. Lower level is the first operand: Eg: <second-level> + <top-level>	BOP + BOP – BOP /
READ	READ; Read an integer; prompt the user for input and push the value to the stack. Make sure the input is validated.	READ
WRITE	WRITE; Write the value of the top of the stack to output. Leave the value on the top of the stack	WRITE
LABEL	LABEL <label>; target for branches;(FALSEBRANCH, GOTO and CALL)	LABEL xyz<<3>> LABEL Read
DUMP	This bytecode is used to set the state of dumping in the virtual machine. When dump is on, after the execution of each bytecode, the state of the runtime stack is dumped to the console.	DUMP ON DUMP OFF

Table 1: List of ByteCodes support by the Interpreter.

4 Sample compiled Code and Trace

4.1 Sample Compiled Code

```
GOTO start<<1>>          program {
LABEL Read
READ
RETURN
LABEL Write              <bodies for read/write functions>
```

```
LOAD 0 dummyformal
WRITE
RETURN
LABEL start<<1>>
LIT 0 i                 int i
LIT 0 j                 int j
GOTO continue<<3>>
LABEL f<<2>>            int f(int i) {
LIT 0 j                 int j
LIT 0 k                 int k
LOAD 0 i                i + j + k + 2
LOAD 1 j
BOP +
LOAD 2 k
BOP +
LIT 2
BOP +
RETURN f<<2>>           return i + j + k + 2
POP 2                   <remove local variables { j,k>
LIT 0
RETURN f<<2>>
LABEL continue<<3>>
LIT 0 m                 int m
LIT 3                   f(3)
ARGS 1
CALL f<<2>>
STORE 2 m               m = f(3)
LOAD 1 j
LOAD 2 m
BOP +                   j + m
ARGS 1
CALL Write              write(j+m)
STORE 0 i               i = write(j+m)
POP 3                   <remove local variables { i,j,m>
HALT
```

4.2 Sample Execution Trace

```
GOTO start<<1>>
LABEL start<<1>>
LIT 0 i                 [0]
```

```
LIT 0 j                 [0,0]
GOTO continue<<3>>      [0,0]
LABEL continue<<3>>     [0,0]
LIT 0                   [0,0,0]
LIT 3                   [0,0,0,3]
ARGS 1                  [0,0,0] [3]
CALL f<<2>>             [0,0,0] [3]
LABEL f<<2>>            [0,0,0] [3]

LIT 0                   [0,0,0] [3,0]
LIT 0                   [0,0,0] [3,0,0]
LOAD 0                  [0,0,0] [3,0,0,3]

LOAD 1                  [0,0,0] [3,0,0,3,0]
BOP +                   [0,0,0] [3,0,0,3]

LOAD 2                  [0,0,0] [3,0,0,3,0]
BOP +                   [0,0,0] [3,0,0,3]
LIT 2                   [0,0,0] [3,0,0,3,2]
BOP +                   [0,0,0] [3,0,0,5]
RETURN                  [0,0,0,5]

STORE 2                 [0,0,5]
```

5 ByteCodeLoader Class

The bytecode loader class is responsible for loading bytecodes from the source code file into a data-structure that stores the entire program. We will use an ArrayList to store our bytecodes. This ArrayList will be contained inside of a Program object. Adding and Getting bytecodes will go through the Program class.

The ByteCodeLoader class will also implement a function that does the following:

1. Reads in the next bytecode from the source file.
2. Build an instance of the class corresponding to the bytecode. For example, if we read LIT 2, we will create an instance of the LitCode class.
3. Read in any additional arguments for the given bytecode if any exists. Once all arguments are parsed, we will pass these arguments to the bytecode's init function.
4. Store the fully initialized bytecode instance into the program data-structure.

5. Once all bytecodes are loaded, we will resolve all symbolic addresses

Address resolution will modify the source code in the following way:

The Program class will hold the bytecode program loaded from the file. It will also resolve symbolic addresses in the program. For example, if we have the following program below

```
0.  FALSEBRANCH continue<<6>>
1.  LIT 2
2.  LIT 2
3.  BOP ==
4.  FALSEBRANCH continue<<9>>
5.  LIT 1
6.  ARGS 1
7.  CALL Write
8.  STORE 0 i
9.  LABEL continue<<9>>
10. LABEL continue<<6>>
```

After address resolution has been completed the source code should look like the following (NOTE you should not modify the original source code file, these changes are made to the Program object):

```
0.  FALSEBRANCH 10
1.  LIT 2
2.  LIT 2
3.  BOP ==
4.  FALSEBRANCH 9
5.  LIT 1
6.  ARGS 1
7.  CALL Write
8.  STORE 0 i
9.  LABEL continue<<9>>
10. LABEL continue<<6>>
```

5.1 ByteCodeLoader Functions

```
1     /**
2      * This constructor will create a new ByteCodeLoader
3      * object which contains a BufferedReader object.
4      * The BUfferedReader object will be initialized with
5      * the filename stored in the file parameter.
```

```
6       * Constructor Simply creates a buffered reader.
7       * YOU ARE NOT ALLOWED TO READ FILE CONTENTS HERE
8       * THIS NEEDS TO HAPPEN IN LOADCODES.
9       */
10      public ByteCodeLoader(String file) throws IOException
11
12      /**
13       * This function is responsible for loading all
14       * bytecodes into the program object. Once all
15       * bytecodes have been loaded and initialized,
16       * the function then will request that the program
17       * object resolve all symbolic addresses before
18       * returning. An example of a before and after
19       * has been given in the Program section of
20       * this document
21       */
22      public Program loadCodes()
```

6 CodeTable Class

The code table class is used by the ByteCodeloader Class. It simply stores a HashMap which allows us to have a mapping between bytecodes as they appear in the source code and their respective classes in the Interpreter project.

The code table class is used by the ByteCodeloader Class. It simply stores a HashMap which allows us to have a mapping between bytecodes as they appear in the source code and their respective classes in the Interpreter project.

The code table can be populated though an initialization method. It is ok to hard-code the statements that populate the data in the CodeTable class.

With the CodeTable HashMap correctly populated we can have the ByteCodeLoader loadCode's function build instances of each bytecode with strings. This is a different approach compared to what we did in Assignment 1 where the objects themselves were in the HashMap. The reason for the different approach is because two bytecode objects of the same type can be two distinct objects in memory. For example,

```
ARGS 0
ARGS 3
```

These two bytecodes will both create an instance of the ArgsCode class, but the objects will contain different values since the bytecode arguments are different. Which means they must

be two distinct objects.

With this mapping (strings to Class names) we can use Java Reflection to build instances of classes. Java reflection is used to inspect classes, interfaces and their members (methods and data-fields). In other languages (including SQL) this is called introspection.

Therefore, using reflection and our HashMap, we can create new instances in the following way:

```
1  // The string value below is a sample, your code
2  // should get this value from the bytecode source file.
3  String code = "HALT";
4
5  // Using the code, get the class name from the CodeTable
6  String className = CodeTable.get(code);
7
8  // Using the class name, load the Class blueprint into the JVM
9  Class c = Class.forName("interpreter.bytecode."+className);
10 // Note when using the forName function, you need to specify
11 // the fully qualified class name. This includes the packages
12 // the class is contained in. The names will be separated
13 // by . not /
14
15
16 // Get a reference to the construction of the class blueprint
17 // referenced by c.Create an instance for the class blueprint
     c.
18 BytceCode bc = (BytceCode)
     c.getDeclaredConstructor().newInstance();
```

Note: that technically each ByteCode subclass does not need a defined constructor.

At this point we have an instance of the given bytecode. Although, its reference type is ByteCode. Its actual type will be HaltCode.

The example above uses the no-arg constructor. This is ok since all bytecodes do not need to have any constructors defined. And when classes do not define their own constructors, Java will give you a no-arg constructor for free. Then we will use the init function defined in each ByteCode subclass to initialize the data-fields of each ByteCode class.

The above code gives us the ability to dynamically create instances of class during runtime.

Please note that in older versions of Java we can create an instance using the newInstance method directly with the class object. However, this is not recommended as the Class.newInstance method bypasses certasin exception checking.
http://errorprone.info/bugpattern/ClassNewInstance

6.1 Code Table Functions

```
    /**
     * The init function will create an entry in the
     * HashMap for each byte code listed in the table
     * presented earlier. This table will be used to
     * map bytecode names to their bytecode classes.
     * For example, POP to PopCode.
     */
    public static void init()

    /**
     * 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)
```

7 Interpreter Class

The interpreter class is used as the entry point for this project. It is responsible for reading in a command line argument that is the file to be ran. Initializing the CodeTable. CAlling loadCodes to create a Progam object, then givening this Program Object to the VirtualMachine to execute.

```
package interpreter;

import java.io.*;

/**
 * <pre>
 *      Interpreter class runs the interpreter:
 *      1. Perform all initializations
 *      2. Load the bytecodes from file
 *      3. Run the virtual machine
 *
 *      THIS FILE CANNOT BE MODIFIED. DO NOT
 *      LET ANY EXCETIONS REACH THE
 *      INTERPRETER CLASS. ONLY EXCEPTION TO THIS RULE IS
 *      BYTECODELOADER CONSTRUCTOR WHICH IS
```

```
 *      ALREADY IMPLEMENTED.
 * </pre>
 */
public class Interpreter {

    private ByteCodeLoader bcl;

    public Interpreter(String codeFile) {
        try {
            CodeTable.init();
            bcl = new ByteCodeLoader(codeFile);
        } catch (IOException e) {
            System.out.println("**** " + e);
        }
    }

    void run() {
        Program program = bcl.loadCodes();
        VirtualMachine vm = new VirtualMachine(program);
        vm.executeProgram();
    }

    public static void main(String args[]) {

        if (args.length == 0) {
            System.out.println("***Incorrect usage, try: java
               interpreter.Interpreter <file>");
            System.exit(1);
        }
        (new Interpreter(args[0])).run();
    }
}
```

8 Program Class

The program class will be responsible for storing all the bytecodes read form the source file. We will store bytecodes in an ArrayList which has a designated type of ByteCode. This will ensure only ByteCodes and its subclass can only be added to the ArrayList.

```
// this function returns the ByteCode at a given index.
public ByteCode getCode(int index)
```

```
3
4 // this function will resolve all symbolic addresses
5 // in the program.
6 public void resolveAddress()
```

Resolving symbolic addresses can be done in many ways. It is up to you to design a clean implementation for mapping the generated labels the compiler uses to absolute addresses in the program. An example is given below.

The Program class will hold the bytecode program loaded from the file. It will also resolve symbolic addresses in the program. For example, if we have the following program below

```
0.  FALSEBRANCH continue<<6>>
1.  LIT 2
2.  LIT 2
3.  BOP ==
4.  FALSEBRANCH continue<<9>>
5.  LIT 1
6.  ARGS 1
7.  CALL Write
8.  STORE 0 i
9.  LABEL continue<<9>>
10. LABEL continue<<6>>
```

After address resolution has been completed the source code should look like the following (NOTE you should not modify the original source code file, these changes are made to the Program object):

```
0.  FALSEBRANCH 10
1.  LIT 2
2.  LIT 2
3.  BOP ==
4.  FALSEBRANCH 9
5.  LIT 1
6.  ARGS 1
7.  CALL Write
8.  STORE 0 i
9.  LABEL continue<<9>>
10. LABEL continue<<6>>
```

9 RuntimeStack Class

Records and processes the stack of active frames. This class will contain two data structures used to help the VirtualMachine execute the program. It is **VERY** important that you do not return any of these data structures. Which means there should be NO getters and setters for these data structures. These data-structures must remain private as well.

The RuntimeStack class will use the following two data-structures:

```
1     // This stack is used to record the beginning of
2     // each activation record(frame) when calling functions.
3     private Stack<Integer> FramePointer
4
5     // This ArrayList is used to represent the runtime stack.
6     // It will be an ArrayList because we will need to access
7     // ALL locations of the runtime stack.
8     private ArrayList<Integer> runStack
```

When trying to understand the Runtime stack class you should always use BOTH data structures. Both are needed for the correct executing of the program.

Recall from earlier:

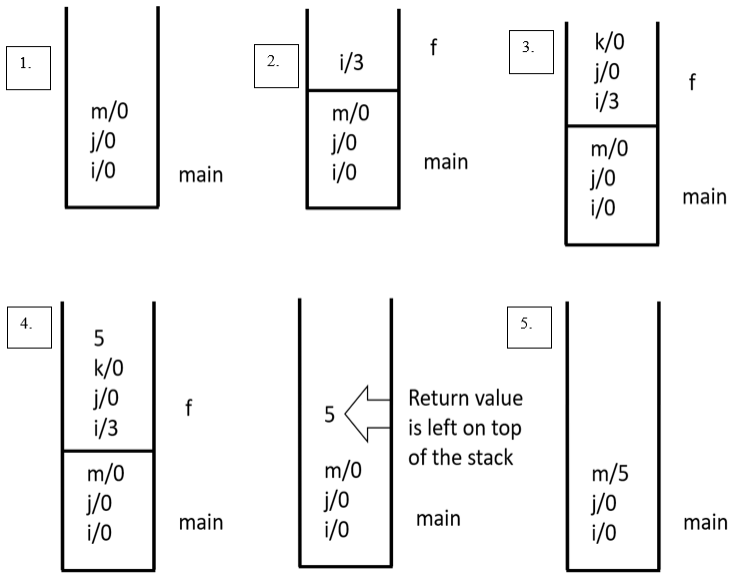

Figure 4: View of Runtime Stack as code is executed.

Initially the framePointer stack will have 0 for step 1 above. When we call function f at step 2, framePointer stack will have the values 0 and 3 since 0 is the start of main and 3 is the start

of f. At step 4.1 we pop the framePointer stack (3 is popped). Then the framePointer Stack will only contain the value 0. This is Mains starting frame index. Note when transitioning from step 4 and 4.1 there is going to be more work that is needed to be done to clean up each frame before returning from functions.

9.1 RunTimeStack Class Functions

```
1  /**
2   * Used for dumping the current state of the runTimeStack.
3   * It will print portions of the stack based on respective
4   * frame markers.
5   * Example [1,2,3] [4,5,6] [7,8]
6   * Frame pointers would be 0,3,6,8
7   */
8  public void dump() { }
9
10 /**
11  * returns the top of the runtime stack, but does not remove
12  * @return copy of the top of the stack.
13  */
14 public int peek() { }
15
16 /**
17  * push the value i to the top of the stack.
18  * @param i value to be pushed.
19  * @return value pushed
20  */
21 public int push(int i) { }
22
23 /**
24  * removes to the top of the runtime stack.
25  * @return the value popped.
26  */
27 public int pop() { }
28 /**
29  * Takes the top item of the run time stack, and stores
30  * it into a offset starting from the current frame.
31  * @param offset number of slots above current frame marker
32  * @return the item just stored
33  */
34 public int store(int offset) { }
35 /**
```

```
36  * Takes a value from the run time stack that is at offset
37  * from the current frame marker and pushes it onto the top of
38  * the stack.
39  * @param offset number of slots above current frame marker
40  * @return item just loaded into the offset
41  */
42 public int load(int offset) { }
43
44 /**
45  * create a new frame pointer at the index offset slots down
46  * from the top of the runtime stack.
47  * @param offset slots down from the top of the runtime stack
48  */
49 public void newFrameAt(int offset) { }
50
51 /**
52  * pop the current frame off the runtime stack. Also removes
53  * the frame pointer value from the FramePointer Stack.
54  */
55 public void popFrame() { }
```

10 VirtualMachine Class

VirtualMachine Class is used for executing the given program. The VirtualMachine basically acts as the controller of this program. All operations need to go through the VirtualMachine class. It will contain the following data-fields (more can be added IF needed):

```
1     // Used to store all variables in program.
2     private RunTimeStack   runStack;
3     //Used to store return addresses for each called
         function(excluding main)
4     private Stack<Integer> returnAddrs;
5     //Reference to the program object where all bytecodes are
         stored.
6     private Program program;
7     //the program counter (current bytecode being executed).
8     private int            pc;
9     //Used to determine whether the VritualMachine should be
         executing bytecodes.
10    private boolean        isRunning;
```

The VritualMachine will contain many functions. These will not be listed. The reason for this is that you are expected to abstract the operations needed by the bytecodes and create clean and concise functions. Do your best to limit the amount of duplicate code. Points will be taken away from VritualMachines that contain a lot of duplicate code (or simply creating a function for each byte code). Also, limit the amount of ByteCode logic that appears in the VritualMachine. For example, if you are executing the ReadCode bytecode, the process of reading is not done by the VritualMachine, it is done by the ReadCode class and then the ReadCode class requests the VritualMachine to push the read value onto the RuntimeStack.

The returnAddrs stack stores the bytdecode index(PC) that the virtual machine should execute when the current function exits. Each time a function is entered, the PC should be pushed onto the returnAddrs stack. When a function exits the PC should be restored to the value that is popped from the top of the returnAddrs Stack.

One important function in the VritualMachine is execute program. A sample base function is given:

```
1  public void executeProgram(){
2      pc = 0;
3      runStack = new RunTimeStack();
4      returnAddrs = new Stack<Integer>();
5      isRunning = true;
6      while(isRunning){
7          ByteCode code = program.getCode(pc);
8          code.execute(this);
9          pc++;
10     }
11 }
```

Note that we can easily add new bytecodes without breaking the operation of the executeProgram function. We are using Dynamic Binding to achieve code flexibility, extensibility and readability. This loop should be very easy to read and understand.

11 Interpreter Dumping

There is one special bytecode that is used to determine whether the VirtualMachine should dump the current state of the RunTimeStack and the information about the currently executed ByteCode. This bytecode is named DUMP. It will have 2 states ON and OFF. No other form of DUMP will occur in this program or any given program. Note that your test files(.x.cod) may not contain these and you will need to add them yourself.

- DUMP ON is an interpreter command to turn on runtime dumping. This will set an interpreter switch that will cause runStack dumping AFTER execution of EACH

bytecode. This switch or variable will be stored as a private data-field in the Virtual-Machine class.

- DUMP OFF will reset the switch to end dumping. Note that DUMP instructions will not be printed.
- DO NOT dump program state unless dumping is turned ON.

Consider the following bytecode program (bytecodes marked with arrows are dumped):

```
GOTO start<<1>>
LABEL Read
REUTRN
LABEL Write
LOAD 0 dummyformal
WRITE
RETURN
LABEL start<<1>>
LIT 0 i
LIT 0 j
GOTO continue<<3>>
LABEL f<<2>>                <-- Dumped
LIT 0 j                     <-- Dumped
LIT 0 k                     <-- Dumped
LOAD 0 i                    <-- Dumped
LOAD 0 j                    <-- Dumped
DUMP OFF                    DUMP IS TURNED OFF
BOP +
LOAD 2 k
BOP +
LIT 2
BOP +
RETURN f<<2>>
POP 2
LIT 0 GRATIS-RETURN-VALUE
RETURN f<<2>>
LABEL continue<<3>>
DUMP ON                     DUMP IS TURNED ON
LIT 0 m                     <-- Dumped
LIT 3                       <-- Dumped
ARGS 1                      <-- Dumped
CALL f<<2>>                 <-- Dumped
DUMP ON                     DUMP IS TURNED ON
```

```
STORE 2                          <-- Dumped
DUMP OFF                         DUMP IS TURNED OFF
LOAD 1 j
LOAD 2 m
BOP +
ARGS 1
WRITE
STORE 0 i
POP 3
HALT
```

When dumping is turned on you should print the following information **JUST AFTER** executing the next bytecode:

- Print the bytecode that was just executed (DO NOT PRINT the DUMP bytecodes)
- Print the runtime stack with spaces separating frames (just after the bytecode was executed). Values contained in one frame should be surrounded by [and].
- If dump is not on then DO NOT print the bytecode, nor dump the runtime stack.

```
LIT 0 m    int m
[0,0,0]
LIT 3
[0,0,0,3]
ARGS 1
[0,0,0] [3]
CALL f<<2>>  f(3)
[0,0,0] [3]
LIT 0 j    int j
[0,0,0] [3,0]
LIT 0 k    int k
[0,0,0] [3,0,0]
LOAD 0 i <load i>
[0,0,0] [3,0,0,3]
LOAD 1 j <load j>
[0,0,0] [3,0,0,3,0]
...
STORE 2 m    = 2
[0,0,5]
```

Note : Following shows the output if dump is on and 0 is at the top of the runtime

stack. RETURN f<< 2 >> exit f:0 note that 0 is returned from f<< 2 >>

- If Dumping is turned on and we encounter an instruction such as WRITE then output as usual; the value may be printed either before or after dumping information e.g.,

```
LIT 3
[0,0,0,3]
3
WRITE
[0,0,0,3]
```

11.1 Dumping Formats

The following dumping actions are taken for the indicated bytecodes, other bytecodes do not have any special treatment when being dumped.

LitCode
For simplicity ALWAYS assume lit is an int declaration.

```
Basic Syntax : LIT <value> <id>  int <id>
               <vallue> is the value being pushed to RuntimeStack
               <id> is a variable identifier
Example      : LIT 0 j     int j
```

LoadCode

```
Basic Syntax : LOAD <offset> <id>  <load id>
               <offset> is the index in the current from to load from
               <id> is a variable identifier
Example      : LOAD 2 j     <load a>
```

StoreCode

```
Basic Syntax : STORE <offset> <id>  <id>=<top-of-stack>
               <offset> is the index in the current from to load from
               <id> is a variable identifier
Example      :
   If we assume the RuntimeStack has the following values:
   [0,1,2,3]
   Then executing a Store 1 k would prodoce:
   STORE 1 k    k=3
```

ReturnCode

```
Basic Syntax : RETURN <id>    EXIT <base-id>:<value>
               <id> is a function identifier
               <value> is the value being returned from the function
               <base-id> is the actual id of the function;
                         e.g. for RETURN f<<2>>, the <base-id> is f
Example      : RETURN f<<2>>  EXIT f : 4
```

CallCode

```
Basic Syntax : CALL <id>   <base-id>(<args>)
                 <id> is a function identifier
                 <base-id> is the actual id of the function;
                         e.g. for CALL f<<2>>, the <base-id> is f
                 <args> are the function arguments.
Example      :
    If we assume the RuntimeStack has the following values
    [0,1,2] [3,4,5]
    And we execute a CALL f<<3>>.
    Before the CALL code is executed, an ARGS 3 has been executed.
    Then the dumping of the call code looks as follows:
    CALL f<<3>>   f(3,4,5)
```

We will also strip any brackets (< and >); the ARGS bytecode just seen tells us that we have a function with 3 arguments, which are the top 3 levels of the stack – the first arg was pushed first, etc.

11.2 DUMPING IMPLEMENTATION NOTES

The Virtual Machine maintains the state of your running program, so it is a good place to have the dump flag. You should not use a stack variable in the ByteCode class.

The dump method should be a part of the RunTimeStack class. This method is called without any arguments. Therefore, there is no way to pass any information about the VritualMachine or the bytecode classes into RunTimeStack. As a result, you can't really do much dumping inside RunTimeStack.dump() except for dumping the state of the RunTimeStack itself. Also, **NO BYTECODE CLASS OR SUBCLASS SHOULD BE CALLING DUMP. The VirtualMachine is the only entity that can trigger either RuntimeStack dumps or ByteCode dumps**.

It is impossible to determine the declared type of a variable by looking at the bytecode file. To simplify matters you should assume whenever the interpreter encounters LIT 0 x (for

example), that it is an int. When you are dumping the bytecodes, it is ok to represent the values as int.

12 Coding Hints

12.1 Possible Order of Implementation

Below is an ordered list of how you may start this assignment. Note that this is not the only way. If you have a better way you may use your way.

1. **Read this PDF 3 or 4 times minimum.**
2. Think and design the ByteCode abstraction. This will help with completing other classes as well.
3. Create all ByteCode classes listed in the table of bytecodes listed above. The classes do not need to be fully implemented but having the method stubs is enough.
4. Implement the ByteCodeLoader class. Note as you implement the loadcodes function, you will need to start filling in the init functions for each of the bytecode classes.
5. Implement the Program Class, specifically the resolveAddress function.
6. Implement the RuntimeStackStack class.
7. Implement the VirtualMachine Class.
8. While completing the implementaion of the VirtualMachine class, you should also start to fill in the execute method of each of the bytecode classes.

12.2 Other Coding Hints

Below are general coding hints and assumptions you may make about the project and the source files used.

- You will assume that the given bytecode source programs (.cod files) you will use for testing are generated correctly and contain NO ERRORS.
- You will need to make sure that you protect the RunTimeStack from stack overflow or stack underflow errors. Also make sure no bytecode can pop past any frame boundary.
- You should provide as much documentation as you can when implementing The Interpreter. Short, OBVIOUS functions don't need comments. However, you should comment each class describing its function and purpose. Take in consideration that

your code is the first level of documentation. How you name things in your code matters. If you find yourself writing a lot of comments, then you may be either explaining a complex algorithm, which is OK, or your code contains named variable and methods that are not descriptive enough.

- **DO NOT** provide any methods that return any components contained WITHIN the VirtualMachine(this is the exact situation that will break encapsulation) – you should request the VirtualMachine to perform operations on its components. This implies that the VirtualMachine owns the components and is free to change them as needed without breaking clients' code (for example, suppose I decided to change the name of the variables that holds my runtime stack - if your code had a reference to that variable then your code would break. This is not an unusual situation – you can consider the names of methods in the Java libraries that have been marked deprecated).

  The only downside is it might be a bit inefficient. Since I want to impress on everyone important software engineering issues, such as encapsulation benefits, I want to enforce the requirement that you **DO NOT BREAK** encapsulation.

  Consider that the VritualMachine calls the individual ByteCodes' execute method and passes itself as a parameter. For the ByteCode to execute, it must invoke 1 or more methods in the runStack object. It can do this by executing VritualMachine.runStack.pop(); however, this DOES break encapsulation. To avoid this, you will need to have a corresponding set of methods within the VritualMachine that do nothing more than pass the call to the runStack. For example, a pop function can be implemented in the VritualMachine in the following way

```
1 public int popRunStack() {
2     return runStack.pop();
3 }
```

  Then in a ByteCodes' execute method a pop can be executed as:

```
1 int temp = VritualMachine.popRunStack();
```

- Each bytecode class should have fields for their specific arguments. The abstract class ByteCode **SHOULD NOT CONTAIN ANY FIELDS (instance variables) THAT ARE RELATED TO ARGUMENTS. BYTECODE SUBCLASSES MUST NOT STORE ARGUMENTS AS STRINGS, A LIST OF STRINGS OR AN ARRAY OF STRINGS. EACH ARGUMENT MUST BE A DATA-FIELD AND HAVE THE CORRECT TYPE.** This is a design requirement.

  It is easier to think in more general terms (i.e. plan for any number of arguments for a bytecode). Note that the ByteCode abstract class should not be aware of the peculiarities of any bytecode. That is, some bytecodes might have zero arguments (HALT) or one argument, etc. Consider providing an init function with each bytecode

class. After constructing a bytecode instance during the loadCodes phase, you can then call init passing in an ArrayList of String as arguments. Each ByteCode object will then interrogate the ArrayList and extract the needed arguments itself. The ByteCode abstract class should not record any arguments for any bytecodes. Each ByteCode concrete class will need instance variables for each of their arguments. There may be a need for additional instance variables as well.

- When you read a line from the bytecode file, you should parse the arguments and place them in an ArrayList of Strings. Then pass this ArrayList to the bytecode's init function. Each bytecode is responsible for extracting relevant information from the ArrayList and storing it as private data.
- **The ByteCode abstract class and its concrete classes must be contained in a package named bytecode.**
- Any output produced by the WRITE bytecode will be interspersed with the output from dumping (if dumping is turned on). In the WRITE action you should print one number per line. DO NOT print out something like :

```
Program Output: 2
```

  Instead, only need to include the value and that is it. It should just be:

```
2
```

- There is no need to check for division by zero error. Assume that you not have a test case where this occurs.

13 Setup

13.1 Importing Project

When importing The interpreter project you will use the root of your repository as the source. **DO NOT USE** the interpreter folder as the root. This will cause the project to not work and in the end force you to change the file structure. As has been noted many times,**YOU ARE NOT ALLOWED TO MODIFY THE THE BASE STRUCUTRE OF THIS PROJECT. DOING WILL CUASE A POINT PENLTY TO BE APPLIED TO YOUR GRADE.**

When executing this project you will need to create the run configurations and set the cammand line arguments. The Interpreter only is able to handle the .x.cod files. The .x are not going to be used for this project, and are present only as a reference.

A sample picture of a run configuration is given below:

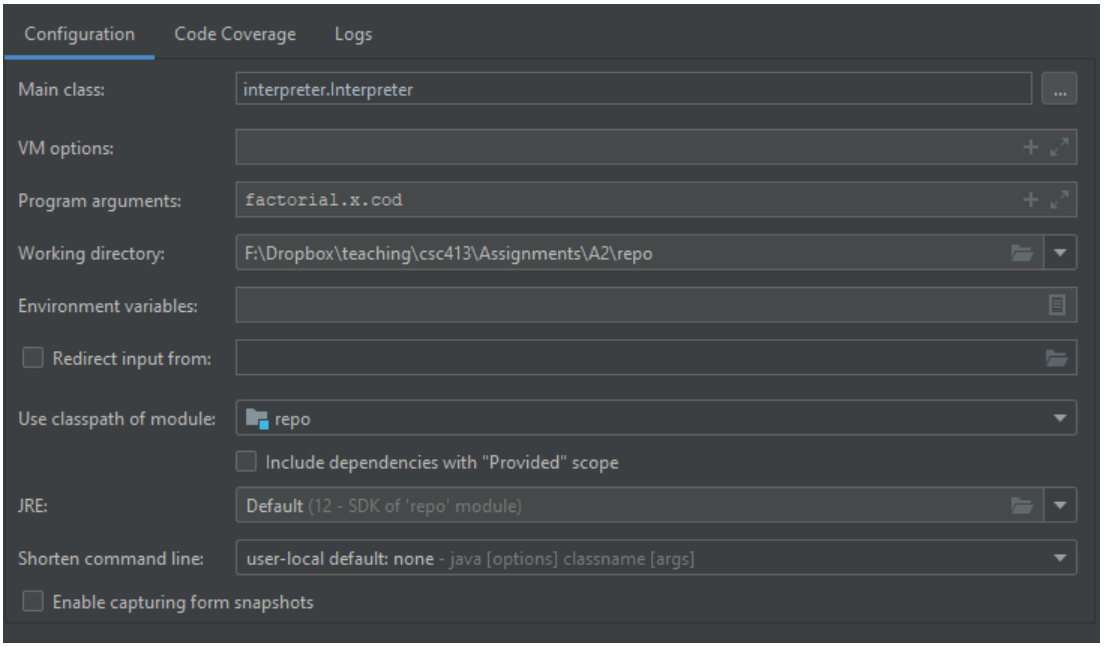

Figure 5: Sample picture of a InteliJ run configuration for The Interpreter project.

Note that the working directory and the Use Classpath of module values may not be the same. The purpose of the image it show where the filename of the bytecode source file goes.

14 Requirements

1. Implement ALL the ByteCode classes listed in Table 1 the table on page 5 of this document. Be sure to create the correct abstractions for the bytecodes. It is possible to have multiple abstractions within the set of byte codes classes.
2. Make sure to adhere to the design requirements listed in the above sections for each of the classes and the ByteCode abstraction.
3. Complete the implementation of the following classes:
   a ByteCodeLoader
   b Program
   c RuntimeStack
   d VirtualMachine

   The Interpreter and CodeTable classes have already been implemented for you. The Interpreter class is the entry point to this project. All projects will be graded using

this entry point. **The Interpreter class MUST NOT be changed. Doing so will cause you project to lose points.**

4. Make sure that all variables have their correct modifiers. Projects with variables using the incorrect modifiers will lose points.

5. Make sure not to break encapsulation. Projects that contain objects or classes trying to access members that it should not be allowed to will lose points. For example, if a bytecode needs to access or write to the runtime stack, it should **NOT** be allowed to. It needs to request these operations from the virtual machine. Then the virtual machine will carry out the operation.

   It would also be incorrect to make a method in the virtual machine for each byte code. While this approach will work, this solution will produce a lot of duplicate code. Points will be deducted for this type of solution. Do your best to understand the operations of each bytecode and how they manipulate the data-structures the Virtual Machine maintains. You will be able to see that some bytecodes operate on these data-structures in a very similar way. Basically, you are being asked to code to the virtual machine and not to the byte codes.

6. The basic structure of the interpreter folder and its contents cannot be changed. You may add subfolders to the bytecode folder. **10 POINTS WILL BE DEDUCTED if the structure of the project is changed.**

15 Submission

1. All completed source code needs to be submitted to your repository created with the link on iLearn. Use the following commands to commit your code. Please note you must be in the repository folder to run these commands.

```
1 $ git add .
2 $ git commmit -m "message"
3 $ git push origin master
```

2. Complete the required documentation as outlined in the documentation guidelines. This is stored in the documentaiton folder. Convert the completed document to a PDF file and save in the documentation folder. **PDF FILES ARE REQUIRED. 5 POINTS WILL BE DEDUCTED FOR NOT SUBMITTING A PDF**