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):

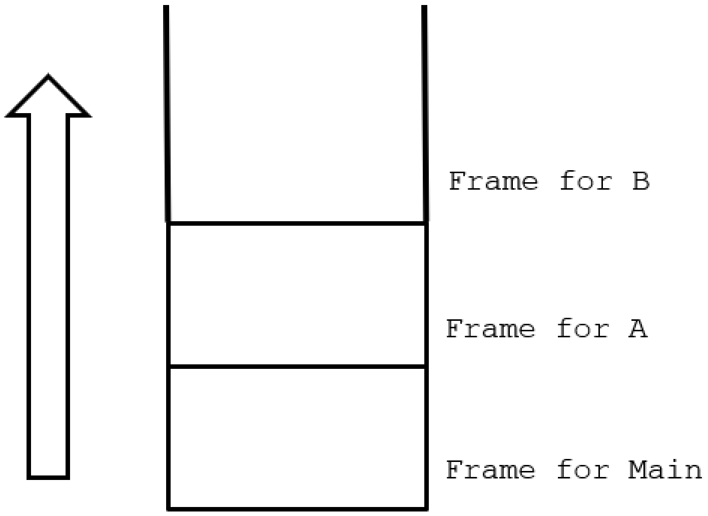
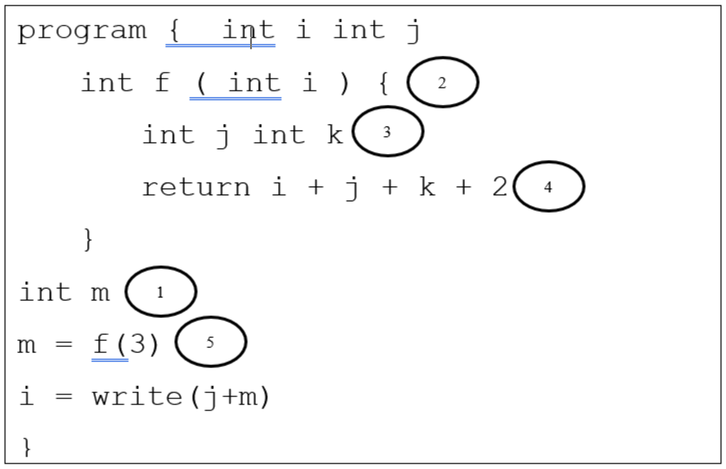


Figure 1: View of runtime stack as methods are called.

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.

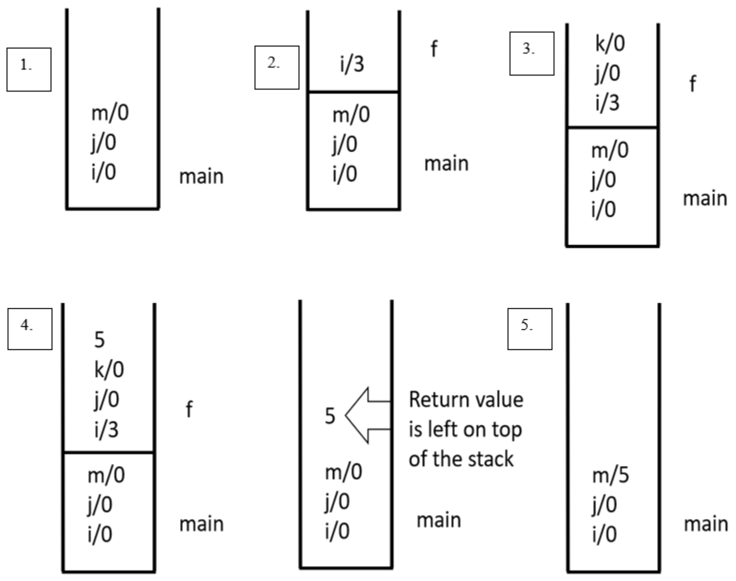


Figure 3: View of Runtime Stack as code is executed.

3 Supported ByteCodes

Below will you will find the descriptions of the ByteCodes your interpreter must support. Each ByteCode must be implemented to the correct specifications outline below. It is important that your understand what each ByteCode does as this will help you implement the ByteCodes.

3.1 Halt ByteCode

The HALT ByteCode is used to alert the virtual machine that program execution is to be stopped. Halt is not allowed to kill the interpreter program. Therefore, Halt may not call System.out.exit to stop the execution of the program.

3.1.1 Requirements

- Notify the VirtualMachine that execution needs to be Halted.
- Halt takes no arguments.
- Halt ByteCodes are not be Dumped.
- Halt cannot execute a system.exit function call.

3.2 Pop ByteCode

The Pop ByteCode will be used to remove values from the run time stack. The Pop ByteCode is not allowed to remove values across frame boundaries. It is the implementers responsibility to ensure that Pop is only allowed to pop the appropriate number of values. Pop has one argument, N , which is the number of values to be popped. You cannot make any assumptions about the value of N other than it is a strictly positive number.

3.2.1 Requirements

- Pop takes one argument which is the number of values to remove from the run time stack.
- Pop is not allowed operate across frame boundaries.
- If dump is on, Pop is required to be dumped. Examples are given in this document.

3.3 FalseBranch ByteCode

The FalseBranch ByteCode will be used to execute conditional jumps(think of executing control structures like if-statements and loops). FalseBranch will have one argument. This argument is a Label that will mark a place in the program to jump to. FalseBranch will remove the top value of the run time stack and check to see if the value is 0. If the value is 0, jump the corresponding label. If the value is something else, move to the next ByteCode in the program. FalseBranch will need to have its label address calculated before the program begins executing. This requires finding where the destination of the jump is going to be numerically(address in the program) in the program.

3.3.1 Requirements

- FalseBranch takes one argument, a label to jump to.
- FalseBranch's label address will need to be resolved. This requires computing where FalseBranch will jump to if the value popped from the stack is 0. Address resolution needs to be done before you began executing the program. This will be discussed later in this document.
- Remove the top value of the stack.
    - if value is 0, jump to label.
    - if value is not 0, move to next ByteCode.
- If dump is on, FalseBranch ByteCode is required to be dumped. Examples are given later in this document.

3.4 Goto ByteCode

The Goto ByteCode is used to jump to Labels in our programs. Goto is considered an unconditional jump. This means regardless of the state of the program, we take the jump. Goto has one argument, the label it needs to jump to. Like FalseBranch, Goto's label needs to go through address resolution as well.

3.4.1 Requirement

- Goto has one argument, a label to jump to.
- Goto's Label must have its address resolved before the program begins executing. More on this later.
- If dump is on, Goto is required to be dumped. Examples are given in this document.

3.5 Store ByteCode

The Store ByteCode will be used to move values from the top of the run time stack to an offset in the current frame. This offset starts from the beginning of the frame. The idea behind this ByteCode is it is needed to do operations like assignments. The Store ByteCode is not allowed to operate across frame boundaries.

3.5.1 Requirements

- The Store ByteCode can have 1 to 2 arguments.
  - one argument is the offset in the current frame where the value that is popped is to be stored.
  - The second argument, if present, is the identifier (variable) the value being moved belongs to. This we be used for dumping.
- Store must pop the top of the runtime stack and store the value at the offset in the current frame.
- Store cannot operate across frame boundaries.
- If dump is on, Store needs to be dumped according the specifications given in the Dumping Formats section. Store ByteCode Format

3.6 Load ByteCode

The Load ByteCode will be used to move values from an offset in the current frame to the top of the stack. This offset works from the beginning of the frame. The purpose behind this ByteCode is it is needed to setup copies of values for things like expressions and arguments for functions. The load ByteCode is not allowed to operate across frame boundaries.

3.6.1 Requirements

- The Load ByteCode can have 1 to 2 arguments.
  - one argument is the offset in the current frame where the value is to be copied from.
  - The second argument, if present, is the identifier (variable) the value belongs to. This we be used for dumping.
- Load must copy the value at the offset in the current and push it to the top of the stack.
- Load must not remove any values from the runtime stack.
- Load cannot operate across frame boundaries.
- If dump is on, Load needs to be dumped according the specifications given in the Dumping Formats section. Load ByteCode Format

3.7 Lit ByteCode

The Lit ByteCode is used to push literal values to the runtime stack. In some cases, Lit ByteCodes will be accompanied with an id (a variable name), this id represents the variable name the value belongs to. This id is optional.

3.7.1 Requirements

- The Lit ByteCode takes 1 or 2 arguments.
- The Lit ByteCode should only push 1 value to the top of the runtime stack.
- If dumping is on, Lit ByteCode needs to be dumped according the specifications in the Dumping formats section. Lit ByteCode Format

3.8 Args ByteCode

The Args ByteCode is going to be used to setup how many arguments a function has. The Args ByteCode will always be executed just before a Call ByteCode. The Args ByteCode has one argument, the number of values that are arguments for the next function call. This value N, will be used to determine how many values starting from the top of the runtime stack will be a part of a newly created activation frame for the next function call. Args will need to figure out where in the runtime stack this new frame begins at and push this index into the FramePointer stack.

3.8.1 Requirements

- The Args byteCode has one argument, the number of values that will be a part of the new activation frame.
- The Args ByteCode will need to push the starting index of the new frame to the framePointer stack.
- If dump is on, the Args ByteCode is required to be dumped. Examples are given in this document.

3.9 Call ByteCode

The Call ByteCode is what the VirtualMachine uses to jump to locations in the program to execute sections of code we call Functions. When encountered the Call ByteCode will jump to the corresponding label in the program. The ByteCde is also responsible of keeping track of where control should return to when a function completes its execution.

3.9.1 Requriements

- Call ByteCode takes 1 argument, a label to jump to.
- Call Code must go through address resolution to figure out where it needs to jump to in the Program before the program is ran.
- Call Code must store a return address onto the Return Address Stack.

- Call Code must Jump the address in the program that corresponds to a label code (this address is computed during address resolution).
- If dumping is on, the Call ByteCode needs to be dumped according the specifications in the Dumping formats section. Call ByteCode Format

3.10 Return ByteCode

The Return ByteCode will be used to return from functions but also to put return values in the correct position on the runtime stack. The interpreter project will use this convention for handling arguments and return values. Callers of functions are required to setup argument for the functions they call. Functions themselves (callees) are required to put return values in the correct spot just before returning from a function. Note that this is a convention we will adhere to and is something that is not enforced programmatically. This means if you fail to follow this convention, transient bugs can happen. The Return ByteCode has a lot of responsibility. The steps for completing a return is important.

3.10.1 Requirements

- The Return ByteCode can take 0 to 1 arguments. The arguments have no effect on its functionality. But does effect the Dumping process.
- The Return ByteCode must store the return value at the top of the runtime stack.
- The Return ByteCode must empty the current frame of all values when the function is complete.
- The Return ByteCode must pop the top value from the framePointer stack to remove the frame boundary.
- The return ByteCode must pop the top of the return address stack and save it into program counter.
- If dumping is on, the Return ByteCode needs to be dumped according the specifications in the Dumping formats section. Return ByteCode Format

3.11 Bop ByteCode

The Bop ByteCode is used to implement binary operations for the Interpreter Project. The Bop ByteCode will need to remove 2 values from the runtime stack and operate on them according to an operation. The result needs to be pushed back to the top of the stack. Be careful, the order of the operands matter. HINT: operands will be pushed in the correct order but the popped in the reverser order.

3.11.1 Requirements

- Bop must pop 2 values from the runtime stack.
- Bop must push 1 value, the result, back to the top of the runtime stack.
- Bop must implement the following binary operations:

  Addition: $+$

  Subtraction: $-$

  Division: $/$

  Multiplication: $*$

  Equality: $==$

  Not-Equal To: $!=$

  Less-Than Equal To: $<=$

  Greater Than: $>$

  Greater Than Equal To: $>=$

  Less Than: $<$

  Logical OR: $|$

  Logical AND: &
- If dump is on, the Bop ByteCode is required to be dumped. Examples are given in this document.

3.12 Read ByteCode

The Read ByteCode is used to read user input from the keyboard. Only integers should be accepted from users. You may use Scanners or BufferedReaders to read input from the user.

3.12.1 Requirements

- When asking for user input, use the following prompt: "Please enter an integer : "
- The Read ByteCode needs to verify that the value given is actually a number. If an invalid number is given, state that the input is invalid and ask for another value. Continue to do so until a valid value is given.
- If dumping is on, Simply print "READ" to the console.

3.13 Write ByteCode

The Write ByteCode is used to display information to the console. The only thing Write is allowed to display is the top value of the runtime stack. No other information is allowed to be shown.

3.13.1 Requirements

- Prints the top of the runtime stack to the console.
- NO OTHER information can be printed by the Write ByteCode when printing the value.
- If dumping is on, Simply print ”WRITE” to the console.

3.14 Label ByteCode

The Label ByteCode is a ByteCode that has no functionality. Its sole purpose is to mark locations in the program where other ByteCodes can jump to. Label ByteCodes will be used to address resolution so other ByteCodes know where they are supposed to jump to.

3.14.1 Requirements

- Label takes one argument, a label which is used to denote a location in the program.
- Dumping Label ByteCodes is optional.

3.15 Dump ByteCode

The Dump ByteCode is used to turn dumping ON and OFF. Dumping in the interpreter project is only done when dumping is ON.

3.15.1 Requirements

- The Dump ByteCode has 1 argument. Either ”ON” or ”OFF”
- The Dump ByteCode must request the VirtualMachine to turn dumping either ”ON” or ”OFF”.
- The Dump ByteCode is NOT TO BE DUMPED.

.

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 ByteCodeLoader 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

```java
    /**
     * This constructor will create a new ByteCodeLoader
     * object which contains a BufferedReader object.
     * The BufferedReader object will be initialized with
     * the filename stored in the file parameter.
     * Constructor Simply creates a buffered reader.
     * YOU ARE NOT ALLOWED TO READ FILE CONTENTS HERE
     * THIS NEEDS TO HAPPEN IN LoadCodes.
     */
    public ByteCodeLoader(String file) throws IOException

    /**
     * This function is responsible for loading all
     * bytecodes into the program object. Once all
     * bytecodes have been loaded and initialized,
     * the function then will request that the program
     * object resolve all symbolic addresses before
     * returning. An example of a before and after
     * has been given in the Program section of
     * this document
     */
    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 load-Code'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:

```java
// The string value below is a sample, your code
// should get this value from the bytecode source file.
String code = "HALT";

// Using the code, get the class name from the CodeTable
String className = CodeTable.get(code);

// Using the class name, load the Class blueprint into the JVM
Class c = Class.forName("interpreter.bytecode."+className);
// Note when using the forName function, you need to specify
// the fully qualified class name. This includes the packages
// the class is contained in. The names will be separated
// by . not /


// Get a reference to the construction of the class blueprint
// referenced by c.Create an instance for the class blueprint
   c.
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 certain 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 className for byte code.
     * @return class name of desired byte code.
     */
    public static String getClassName(String className)
```

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 Program object, then giving 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
```

```java
 *
 *      THIS FILE CANNOT BE MODIFIED. DO NOT
 *      LET ANY EXCEPTIONS REACH THE
 *      INTERPRETER CLASS. ONLY EXCEPTION TO THIS RULE IS
 *      ByteCodeLoader CONSTRUCTOR WHICH IS
 *      ALREADY IMPLEMENTED.
 * </pre>
 */
public class Interpreter {

    private ByteCodeLoader byteCodeLoader;

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

    void run() {
        Program program = byteCodeLoader.loadCodes();
        VirtualMachine virtualMachine = new
           VirtualMachine(program);
        virtualMachine.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.

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

// this function will resolve all symbolic addresses
// in the program.
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:

```
    // This stack is used to record the beginning of
    // each activation record(frame) when calling functions.
    private Stack<Integer> framePointer

    // This ArrayList is used to represent the runtime stack.
    // It will be an ArrayList because we will need to access
    // ALL locations of the runtime stack.
    private ArrayList<Integer> runTimeStack
```

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:

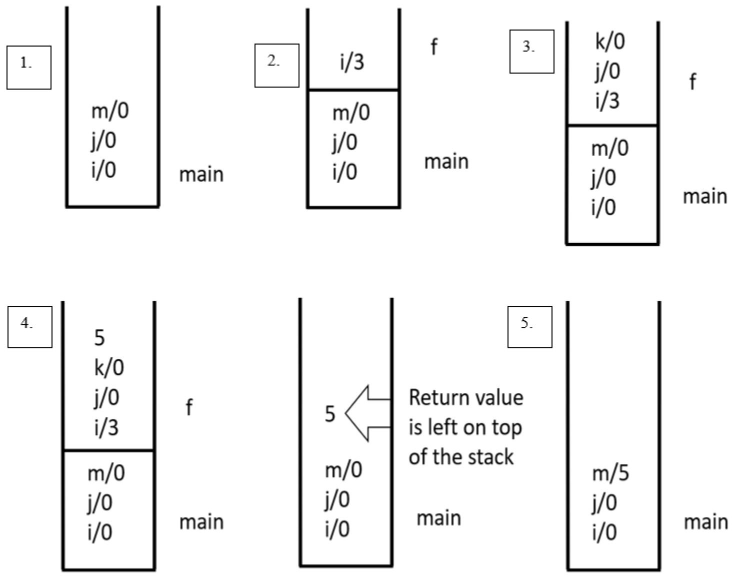


Figure 4: View of Runtime Stack as code is executed.

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

```
/**
 * Used for dumping the current state of the runTimeStack.
 * It will print portions of the stack based on respective
 * frame markers.
 * Example [1,2,3] [4,5,6] [7,8]
 * Frame pointers would be 0,3,6
 */
public void dump() { }

/**
 * returns the top of the runtime stack, but does not remove
 * @return copy of the top of the stack.
 */
public int peek() { }

```

```java
/**
 * push the value i to the top of the stack.
 * @param i value to be pushed.
 * @return value pushed
 */
public int push(int i) { }

/**
 * removes to the top of the runtime stack.
 * @return the value popped.
 */
public int pop() { }
/**
 * Takes the top item of the run time stack, and stores
 * it into a offset starting from the current frame.
 * @param offset number of slots above current frame marker
 * @return the item just stored
 */
public int store(int offset) { }
/**
 * Takes a value from the run time stack that is at offset
 * from the current frame marker and pushes it onto the top of
 * the stack.
 * @param offset number of slots above current frame marker
 * @return item just loaded into the offset
 */
public int load(int offset) { }

/**
 * create a new frame pointer at the index offset slots down
 * from the top of the runtime stack.
 * @param offset slots down from the top of the runtime stack
 */
public void newFrameAt(int offset) { }

/**
 * pop the current frame off the runtime stack. Also removes
 * the frame pointer value from the FramePointer Stack.
 */
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):

```java
    // Used to store all variables in program.
    private RunTimeStack   runTimeStack;
    //Used to store return addresses for each called
       function(excluding main)
    private Stack<Integer> returnAddress;
    //Reference to the program object where all ByteCodes are
       stored.
    private Program program;
    //the program counter (current ByteCode being executed).
    private int            programCounter;
    //Used to determine whether the VirtualMachine should be
       executing ByteCodes.
    private boolean        isRunning;
```

The VirtualMachine 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 VirtualMachines 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 VirtualMachine. For example, if you are executing the ReadCode ByteCode, the process of reading is not done by the VirtualMachine, it is done by the ReadCode class and then the ReadCode class requests the VirtualMachine to push the read value onto the RuntimeStack.

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

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

```java
public void executeProgram(){
    programCounter = 0;
    runTimeStack = new RunTimeStack();
    returnAddress = new Stack<Integer>();
    isRunning = true;

    while(isRunning){
        ByteCode code = program.getCode(programCounter);
        code.execute(this);
        programCounter++;
    }
}
```

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 VirtualMachine 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 current 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>
                <value> 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 j>
```

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 produce:
   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 > from functions names for dumping. 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 VirtualMachine 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 implementation 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 VirtualMachine 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 VirtualMachine.runStack.pop(); however, this DOES break encapsulation. To avoid this, you will need to have a corresponding set of methods within the VirtualMachine that do nothing more than pass the call to the runStack. For example, a pop function can be implemented in the VirtualMachine in the following way

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

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

```java
int temp = VirtualMachine.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.

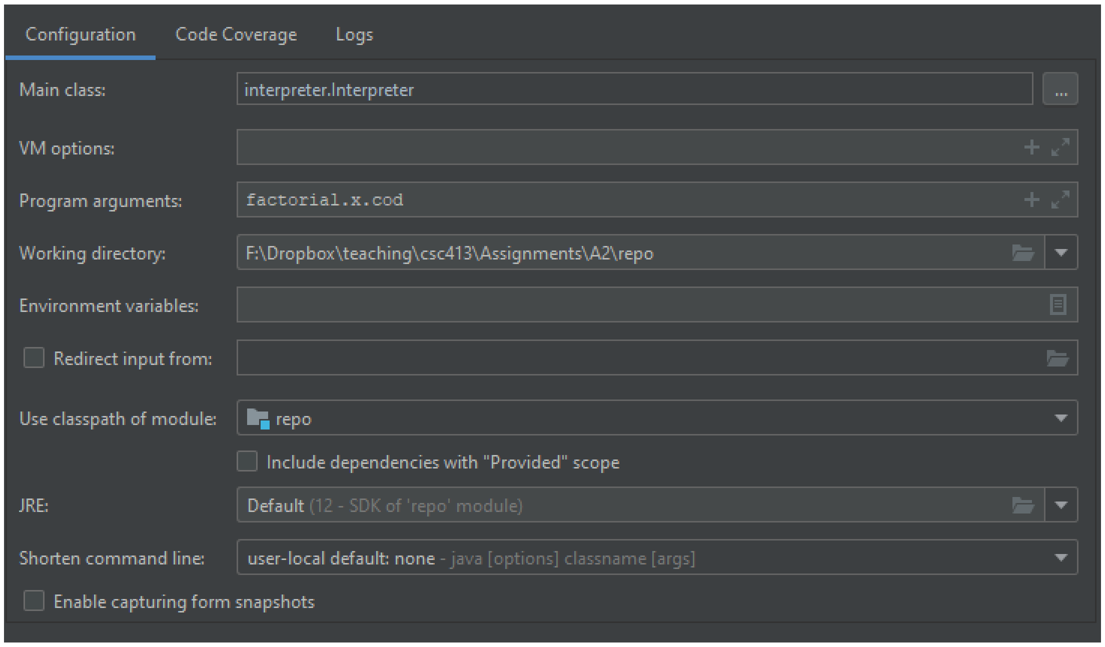
13 Setup

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 command 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:



Main class: interpreter.Interpreter

VM options:

Program arguments: factorial.x.cod

Working directory: F:\Dropbox\teaching\csc413\Assignments\A2\repo

Environment variables:

☐ Redirect input from:

Use classpath of module: repo

☐ Include dependencies with "Provided" scope

JRE: Default (12 - SDK of 'repo' module)

Shorten command line: user-local default: none - java [options] classname [args]

☐ Enable capturing form snapshots

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 the Supported ByteCodes 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 sub-folders 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.

```
                    $ git add .
                    $ git commit -m "message"
                    $ git push origin master
```

2. Complete the required documentation as outlined in the documentation guidelines. This is stored in the documentation 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**