Assignment 2: The Interpreter

CSC 413 Summer 2019

Student name: Tsun Ming Lee

ID: 918541473

Github Link:

https://github.com/csc413-02-summer2019/csc413-p2-hkmatthew711

1. Introduction

1.1 Project Overview

This project is an interpreter program for the mock language X, and it interprets byte codes of programs written in mock language X.

1.2 Technical Overview

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 will work together to run a program written in the Language X. There are two sample programs in recursive version: one is for computing the nth Fibonacci number and another is for finding the factorial of a number. Both program files have the extension x.cod.

1.3 Summary of Work Completed

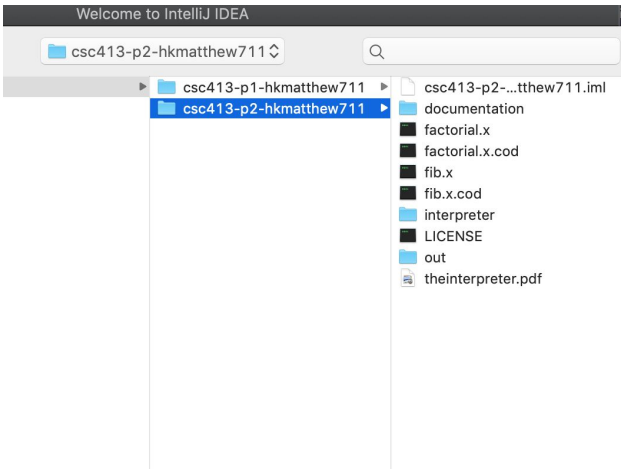
In this project, we needed to create 15 sub classes of byte codes for ByteCodeLoader class and VirtualMachine class to load the byte codes and execute the X program. We also implemented Program class for storing all the byte codes, and implemented the RunTimeStack class for processing the stack of active frames.

2. Development Environment

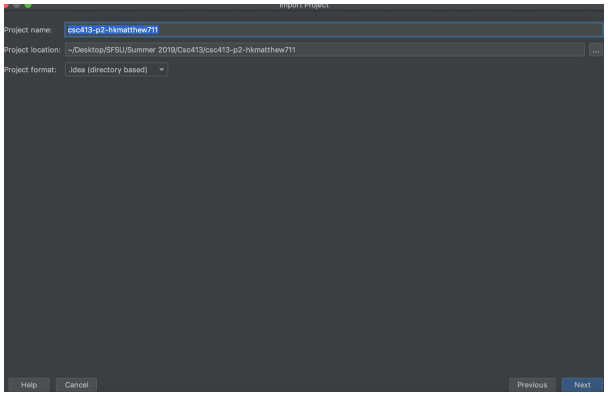
For this project, I use IntelliJ IDEA CE to compile it, java version "2019.1.3".

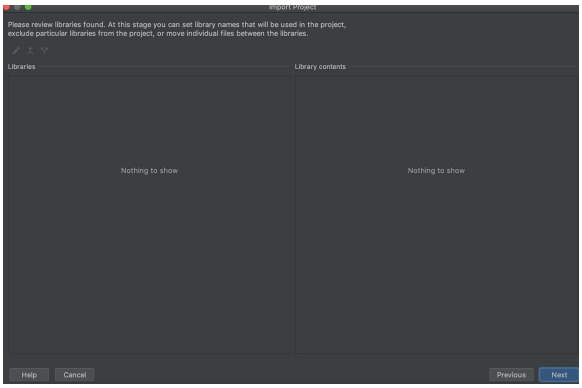
3. How to Build/Import your Project

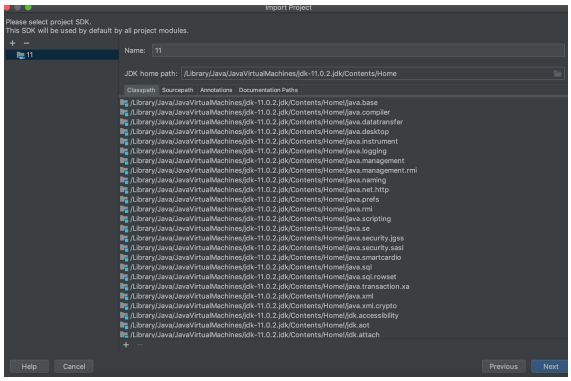
The environment for the project is IntelliJ. First, open IntelliJ. Then click the Import Project on the welcome screen. Find the project location and click open.

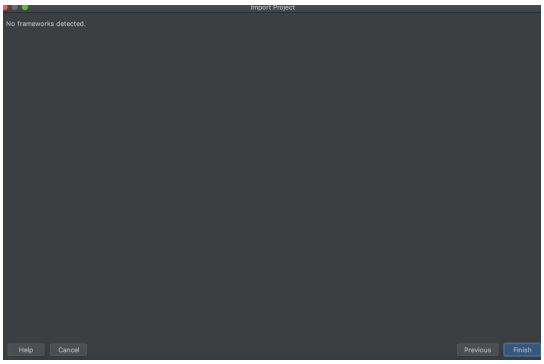


Then following the screen and keep click on Next, until Finish.



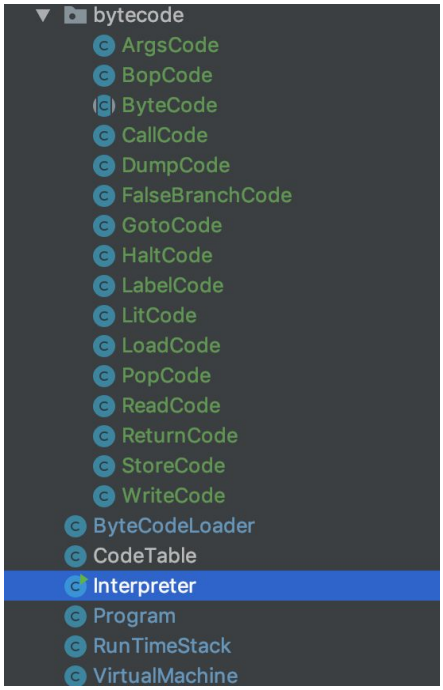






4. How to Run your Project

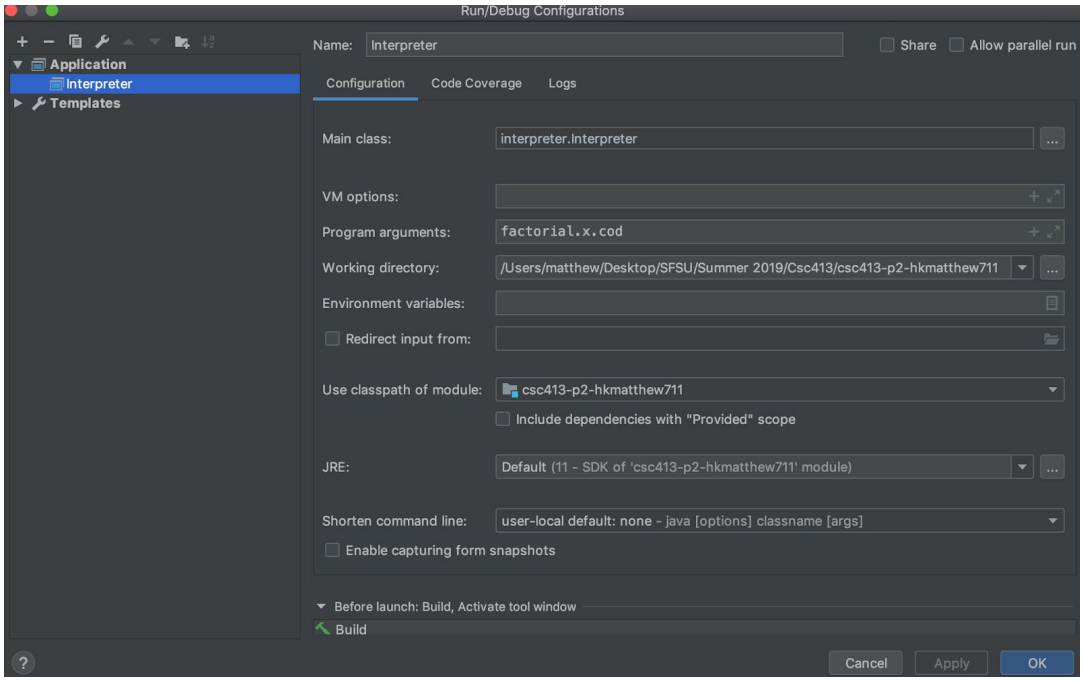
After importing the project, it comes to the screen that lists all the project files on the left side



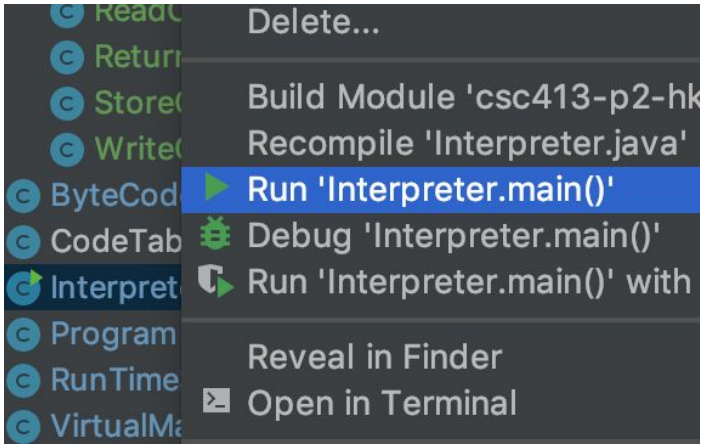
Before running this project, it is necessary to set up the configuration because the project needs an argument to execute. The argument for this project is the X program file.

Click on Add Configuration(located at the upper right tab) from the above screen and choose Application below Templates.

• The main class for this project is Interpreter(or just enter: **interpreter.Interpreter**).
• Enter the X program file name for Program Arguments. For example: **factorial.x.cod**
• Set the working directory as the path for the project folder.
After adding the required configurations, click on Apply and OK.



To run the project, right click Interpreter and choose Run ‘Interpreter.main()’



The following is the output for executing **factorial.x.cod:**

```
GOTO 8
[]
GOTO 33
[]

ARGS 0
[][]
CALL 1
[][]
Enter an integer: 1
READ
[][1]
RETURN
[1]

ARGS 1
[][1]
CALL 10
[][1]
LOAD 0 n
[][1,1]
LIT 2
[][1,1,2]
BOP  <
[][1,1]
FALSEBRANCH 19
[][1]
LIT 1
[][1,1]
RETURN factorial  exit
factorial<<2>>
[1]
```

```
ARGS 1
[][1]
CALL 4
[][1]
LOAD 0 dummyFormal
[][1,1]
WRITE
1
WRITE
[][1,1]
RETURN
[1]
POP 3
[]
HALT
[]
```

The following is the output for executing fib.x.cod:

```
GOTO 8
[]
LIT 0       int x
[][0]
GOTO 49
[0]
LIT 0       int k
[][0,0]
LIT 5
[][0,0,5]
STORE 0 x x = 0
[][5,0]

ARGS 0
[][5,0]
CALL 1
[][5,0]
Enter an integer: 1
READ
[][5,0] [1]
RETURN
[5,0,1]

ARGS 1
[][5,0] [1]
CALL 11
[][5,0] [1]
LOAD 0 n
[][5,0] [1,1]
LIT 1
[][5,0] [1,1,1]
BOP  <=
[][5,0] [1,1]
FALSEBRANCH 20
[][5,0] [1]
LIT 1
[][5,0] [1,1]
RETURN factorial  exit
fib<<2>>
[5,0,1]
```

```
ARGS 1
[][5,0] [1]
CALL 4
[][5,0] [1]
LOAD 0 dummyFormal
[][5,0] [1,1]
WRITE
1
WRITE
[][5,0] [1,1]
RETURN
[5,0,1]
STORE 1 k k = 1
[][5,1]
LIT 0       int x
[][5,1,0]
LIT 7
[][5,1,0,7]
STORE 2 x x = 7
[][5,1,7]
LIT 8
[][5,1,7,8]
STORE 2 x x = 8
[][5,1,8]
POP 1
[5,1]
POP 2
[5]
HALT
[5]

Process finished with exit code 0
```

5. Assumptions Made

-- I assumed that the project will first execute the X program file then do the interpretation of the byte codes.
-- I assumed that the project can display the X program output.

6. Implementation Discussion

Package Class:

1. **ByteCodeLoader.java:** The only function that needs to be implemented is loadCodes(). It is a function for setting the byte codes in an ArrayList, and reads the ArrayList.
2. **CodeTable.java:** It has a hash map for the byte codes. No implementation is needed
3. **Interpreter.java:** It is the main class of the project. No implementation is needed
4. **Program.java:** The only function that needs to be implemented is resolveAddrs(). This function resolves all the addresses of particular byte codes.
5. **RunTimeStack.java:** There are 10 functions in this class. dump() is to dump the current stack and print the arranged brackets; peek() returns the first element of the stack, which is an ArrayList; pop() removes the first element of the stack; push() adds a new element in the stack; newFrameAt() creates a new frame with the parameter offset; store() stores the an element in the stack; load() loads an element in the stack; popFrame() pops the last element in the frame; stackSize() returns the size of the stack; Integer push() loads literals onto the stack.
6. **VirtualMachine.java:** This class executes the X program, calling the functions in RunTimeStack.java

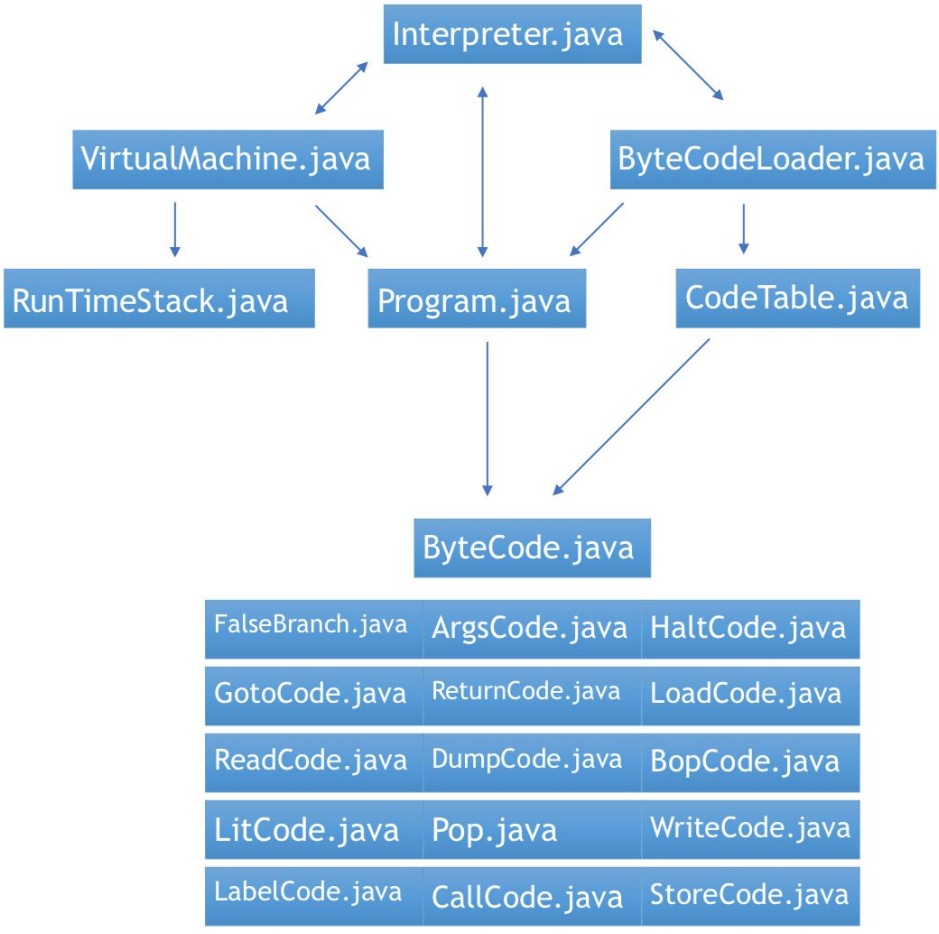
Bytecode Class:

1. **ByteCode.java:** This abstract class initialize the functions for byte codes
2. **HaltCode.java:** Stops execution
3. **PopCode.java:** Pops the top of the stack
4. **FalseBranchCode.java:** Pops the top of stack; if there is nothing to pop, then sets branch, else execute the next byte code
5. **GotoCode.java:** Initializes execution
6. **StoreCode.java:** Pops the top of the stack; stores the value into the offset from the start of the frame
7. **LoadCode.java:** Pushes the value in the slot which is offset from the start of the frame onto the top of the stack

8. **LitCode.java:** Loads the literal value
9. **ArgsCode.java:** Used prior to calling a function
10. **CallCode.java:** Transfers control to the indicated function
11. **ReturnCode.java:** Returns from the current function
12. **BopCode.java:** Pops top 2 levels of the stack and perform the indicated operation
13. **ReadCode.java:** Reads a integer; prompts the user for input and push the value to the stack
14. **WriteCode.java:** Writes the value of the top of the stack to output
15. **LabelCode.java:** Target for branches
16. **DumpCode.java:** Used to set the state of dumping in the virtual machine

6.1 Class Diagram

The following is a class diagram shows the relationship among classes:



7. Project Reflection

There are a lot of coding needed for this project, especially the four big classes that I needed to implement. There are flaws in some classes because when I get the output, the output is not matched with the expected output. I did spend some time to figure out how to implement some classes, such as RunTimeStack.java and Program.java. While I was trying to build, I encounter several errors, such as the out-of-bound error. Overall, this is a tough project, but I also learned what each byte code means in a program.

8. Project Conclusion/Results

In conclusion, the project works as an interpreter of mock language X that displays how each byte code operate in the X program. The result matches my assumptions that it displays the actually output of the X program, and interprets every byte code appears in the program.