Goal

1. Become familiar with the `CLEmitter`, an abstraction for generating JVM bytecode (see Appendix D of our text).
2. Extend the base *j--* language by adding some basic Java operations (on primitive integers) to the language. Supporting these operations requires studying the *j--* compiler in its entirety, if only cursorily, and then making slight modifications to it. Notice that many of the operations have different levels of precedence, just as `*` has a different level of precedence in *j--* than does `+`. These levels of precedence are captured in the Java grammar (see appendix at the end); for example, the parser uses one method to parse expressions involving `*` and `/`, and another to parse expressions involving `+` and `-`.

Download and Test the *j--* Compiler

Download and unzip the base *j--* compiler under some directory[1] (we'll refer to this directory as `$j`). See Appendix A for information on what's in the *j--* distribution.

Run the following command inside the `$j/j--` directory to compile the *j--* compiler.

>_ ~/workspace/j--

```
$ ant
```

Run the following command to compile a *j--* program `$j/j--/tests/pass/HelloWorld.java` using the *j--* compiler, which produces the JVM target program `HelloWorld.class` under `./pass`.

>_ ~/workspace/j--

```
$ sh ./bin/j-- tests/pass/HelloWorld.java
```

Run the following command to run `HelloWorld.class`.

>_ ~/workspace/j--

```
$ java pass.HelloWorld
```

Download the Project Tests

Download and unzip the tests for this project under `$j/j--`.

Problem 1. (*Using `CLEmitter`*) Consider the following program `IsPrime.java` that receives an integer n as command-line argument and prints whether or not n is a prime number.

IsPrime.java

```
public class IsPrime {
    // Returns true if n is prime, and false otherwise.
    private static boolean isPrime(int n) {
        if (n < 2) {
            return false;
        }
        for (int i = 2; i <= n / i; i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }

    // Entry point.
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        boolean result = isPrime(n);
        if (result) {
            System.out.println(n + " is a prime number");
        } else {
            System.out.println(n + " is not a prime number");
        }
    }
}
```

Using the annotated program `GenFactorial.java` under `$j/j--/tests/clemitter` as a model, complete the implementation of the program `$j/j--/project1/GenIsPrime.java` that uses the `CLEmitter` interface to programmatically generate `IsPrime.class`, ie, the JVM bytecode for the program `IsPrime.java` above.

[1] We recommend `~/workspace`.

```
>_ ~/workspace/j--

$ javac -d . -cp .:./lib/j--.jar project1/GenIsPrime.java
$ java -cp .:./lib/j--.jar GenIsPrime
$ java IsPrime 42
42 is not a prime number
$ java IsPrime 31
31 is a prime number
```

Hints: There are two ways to approach this problem, the first being more intellectually rewarding.

1. The bytecode for `GenIsPrime.main()` is similar to the bytecode for `GenFactorial.main()`. Here are some hints for generating bytecode for the `isPrime()` method:

    ```
        if n >= 2 goto A:
        return false
    A:  i = 2
    D:  if i > n / i goto B:
        if n % i != 0 goto C:
        return false
    C:  increment i by 1
        goto D:
    B:  return True
    ```

2. Compile `IsPrime.java` using `javac`, and decompile (using `javap`) `IsPrime.class` to get the bytecode `javac` generated and mimic the same in `GenIsPrime`.

Problem 2. (*Division Operation*) Follow the process outlined in Section 1.5 of our text to implement the Java division operator `/`.

AST representation(s):

- `JDivideOp` in `JBinaryExpression.java`

```
>_ ~/workspace/j--

$ sh ./bin/j-- project1/tests/Division.java
$ java Division 42 6
7
```

Problem 3. (*Remainder Operation*) Implement the Java remainder operator `%`.

AST representation(s):

- `JRemainderOp` in `JBinaryExpression.java`

```
>_ ~/workspace/j--

$ sh ./bin/j-- project1/tests/Remainder.java
$ java Remainder 42 13
3
```

Problem 4. (*Shift Operations*) Implement the Java shift operators: arithmetic left shift `<<`, arithmetic right shift `>>`, logical right shift `>>>`.

AST representation(s):

- `JALeftShiftOp` in `JBinaryExpression.java`
- `JARightShiftOp` in `JBinaryExpression.java`
- `JLRightShiftOp` in `JBinaryExpression.java`

```
>_ ~/workspace/j--
$ sh ./bin/j-- project1/tests/ArithmeticLeftShift.java
$ java ArithmeticLeftShift 1 5
32
$ sh ./bin/j-- project1/tests/ArithmeticRightShift.java
$ java ArithmeticRightShift 32 5
1
$ java ArithmeticRightShift -32 5
-1
$ sh ./bin/j-- project1/tests/LogicalRightShift.java
$ java LogicalRightShift 32 5
1
$ java LogicalRightShift -32 5
134217727
```

Problem 5. (*Bitwise Operations*) Implement the Java bitwise operators: unary complement `~`, inclusive or `|`, exclusive or `^`, and `&`. Note: there are JVM instructions for `|`, `^`, and `&`, but not for `~`, which must be computed as the "exclusive or" of the operand and -1.

AST representation(s):

- `JComplementOp` in `JUnaryExpression.java`
- `JOrOp` in `JBinaryExpression.java`
- `JXorOp` in `JBinaryExpression.java`
- `JAndOp` in `JBinaryExpression.java`

```
>_ ~/workspace/j--
$ sh ./bin/j-- project1/tests/BitwiseNot.java
$ java BitwiseNot 42
-43
$ sh ./bin/j-- project1/tests/BitwiseInclusiveOr.java
$ java BitwiseInclusiveOr 3 5
7
$ sh ./bin/j-- project1/tests/BitwiseExclusiveOr.java
$ java BitwiseExclusiveOr 3 5
6
$ sh ./bin/j-- project1/tests/BitwiseAnd.java
$ java BitwiseAnd 3 5
1
```

Problem 6. (*Unary Plus Operation*) Implement the Java unary plus operaor `+`.

AST representation(s):

- `JUnaryPlusOp` in `JUnaryExpression.java`

```
>_ ~/workspace/j--
$ sh ./bin/j-- project1/tests/UnaryPlus.java
$ java UnaryPlus -42
-42
```

Files to Submit

1. `$j/j--/project1/GenIsPrime.java`
2. `$j/j--/src/jminusminus/TokenInfo.java`
3. `$j/j--/src/jminusminus/Scanner.java`
4. `$j/j--/src/jminusminus/Parser.java`
5. `$j/j--/src/jminusminus/JBinaryExpression.java`
6. `$j/j--/src/jminusminus/JUnaryExpression.java`
7. `$j/j--/project1/report.txt`

Before You Submit

- Make sure you name the classes and files you create exactly as suggested in this writeup. Remember, names are case-sensitive.
- Make sure your report uses the given template, isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling mistakes.

Appendix: Java Syntax

```
compilationUnit ::= [ package qualifiedIdentifier ; ]
                    { import qualifiedIdentifier ; }
                    { typeDeclaration }
                    EOF

qualifiedIdentifier ::= <identifier> { . <identifier> }

typeDeclaration ::= typeDeclarationModifiers ( classDeclaration | interfaceDeclaration )
                  | ;

typeDeclarationModifiers ::= { public | protected | private | static | abstract | final }

classDeclaration ::= class <identifier> [ extends qualifiedIdentifier ]
                         [ implements qualifiedIdentifier { , qualifiedIdentifier } ]
                             classBody

interfaceDeclaration ::= interface <identifier> // can't be final
                             [ extends qualifiedIdentifier { , qualifiedIdentifier } ]
                                 interfaceBody

modifiers ::= { public | protected | private | static | abstract | final }

classBody ::= { { ;
                | static block
                | block
                | modifiers memberDecl
                }
              }

interfaceBody ::= { { ;
                    | modifiers interfaceMemberDecl
                    }
                  }

memberDecl ::= <identifier> // constructor
                   formalParameters
                       [ throws qualifiedIdentifier { , qualifiedIdentifier } ] block
             | ( void | type ) <identifier> // method
                   formalParameters
                       [ throws qualifiedIdentifier { , qualifiedIdentifier } ] ( block | ; )
             | type variableDeclarators ; // fields
```

```
interfaceMemberDecl ::= ( void | type ) <identifier> // method
                              formalParameters
                                  [ throws qualifiedIdentifier { , qualifiedIdentifier } ] ;
                          | type variableDeclarators ; // fields; must have inits

block ::= { { blockStatemnt } }

blockStatement ::= localVariableDeclarationStatement
                 | statement

statement ::= block
            | if parExpression statement [ else statement ]
            | for ( [ forInit ] ; [ expression ] ; [ forUpdate ] ) statement
            | while parExpression statement
            | do statement while parExpression ;
            | try block
                  { catch ( formalParameter ) block }
                      [ finally block ] // must be present if no catches
            | switch parExpression { { switchBlockStatementGroup } }
            | return [ expression ] ;
            | throw expression ;
            | break [ <identifier> ] ;
            | continue [ <identifier> ] ;
            | ;
            | <identifier> : statement
            | statementExpression ;

formalParameters ::= ( [ formalParameter { , formalParameter } ] )

formalParameter ::= [ final ] type <identifier>

parExpression ::= ( expression )

forInit ::= statementExpression { , statementExpression }
          | [ final ] type variableDeclarators

forUpdate ::= statementExpression { , statementExpression }

switchBlockStatementGroup ::= switchLabel { switchLabel } { blockStatement }

switchLabel ::= case expression : // must be constant
              | default :

localVariableDeclarationStatement ::= [ final ] type variableDeclarators ;

variableDeclarators ::= variableDeclarator { , variableDeclarator }

variableDeclarator ::= <identifier> [ = variableInitializer ]

variableInitializer ::= arrayInitializer | expression
```

```
arrayInitializer ::= { [ variableInitializer { , variableInitializer } ] }

arguments ::= ( [ expression { , expression } ] )

type ::= basicType | referenceType

basicType ::= boolean | byte | char | short | int | float | long | double

referenceType ::= basicType [ ] { [ ] }
                | qualifiedIdentifier { [ ] }

statementExpression ::= expression // but must have side-effect, eg, i++

expression ::= assignmentExpression

assignmentExpression ::= conditionalExpression // must be a valid lhs
                         [
                           ( =
                           | +=
                           | -=
                           | *=
                           | /=
                           | %=
                           | >>=
                           | >>>=
                           | <<=
                           | &=
                           | |=
                           | ^=
                           ) assignmentExpression ]

conditionalExpression ::= conditionalOrExpression [ ? assignmentExpression : conditionalExpression ]

conditionalOrExpression ::= conditionalAndExpression { || conditionalAndExpression }

conditionalAndExpression ::= inclusiveOrExpression { && inclusiveOrExpression }

inclusiveOrExpression ::= exclusiveOrExpression { | exclusiveOrExpression }

exclusiveOrExpression ::= andExpression { ^ andExpression }

andExpression ::= equalityExpression { & equalityExpression }

equalityExpression ::= relationalExpression { ( == | != ) relationalExpression }

relationalExpression ::= shiftExpression ( { ( < | > | <= | >= ) shiftExpression } | instanceof referenceType )

shiftExpression ::= additiveExpression { ( << | >> | >>> ) additiveExpression }
```

```
additiveExpression ::= multiplicativeExpression { ( + | - ) multiplicativeExpression }

multiplicativeExpression ::= unaryExpression { ( * | / | % ) unaryExpression }

unaryExpression ::= ++ unaryExpression
                  | -- unaryExpression
                  | ( + | - ) unaryExpression
                  | simpleUnaryExpression

simpleUnaryExpression ::= ~ unaryExpression
                        | ! unaryExpression
                        | ( basicType ) unaryExpression // basic cast
                        | ( referenceType ) simpleUnaryExpression // reference cast
                        | postfixExpression

postfixExpression ::= primary { selector } { ++ | -- }

selector ::= . qualifiedIdentifier [ arguments ]
           | [ expression ]

primary ::= parExpression
          | this [ arguments ]
          | supper ( arguments | . <identifier> [ arguments ] )
          | literal
          | new creator
          | qualifiedIdentifer [ arguments]

creator ::= ( basicType | qualifiedIdentifier )
              ( arguments
              | [ ] { [ ] } [ arrayInitializer ]
              | newArrayDeclarator
              )

newArrayDeclarator ::= [ [ expression ] ] { [ [ expression ] ] }

literal ::= <int_literal> | <char_literal> | <string_literal> | <float_literal>
          | <long_literal> | <double_literal> | true | false | null
```