Goal

1. Become familiar with `CLEmitter`.
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.

Grammars

The lexical and syntactic grammars for *j*-- and Java can be found at `https://www.cs.umb.edu/j--/grammar.pdf` ↗.

Download and Test the *j*-- Compiler

Download and unzip the base *j*-- compiler ↗ under some directory[1] (we'll refer to this directory as `$j`). Run the following command inside the `$j/j--` directory to compile the *j*-- compiler.

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

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

```
>_ ~/workspace/j--
$ bash ./bin/j-- tests/jvm/HelloWorld.java
```

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

```
>_ ~/workspace/j--
$ java HelloWorld
Hello, World
```

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 accepts n (int) as command-line argument, and writes whether or not n is a prime number.

```java
IsPrime.java
public class IsPrime {
    // 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");
        }
    }

    // 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;
    }
}
```

[1] We recommend `~/workspace`.

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` such that it uses the `CLEmitter` interface to programmatically generate `IsPrime.class`, ie, the JVM bytecode for the `IsPrime.java` program listed above.

```
>_ ~/workspace/j--
$ bash ./bin/clemitter project1/GenIsPrime.java
$ java IsPrime 42
42 is not a prime number
$ java IsPrime 31
31 is a prime number
```

Directions: The bytecode for `GenIsPrime.main()` is similar to the bytecode for `GenFactorial.main()`. Here is the pseudocode 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
```

Problem 2. (*Arithmetic Operations*) Implement the Java arithmetic operators: division `/`, remainder `%`, and unary plus `+`.

AST representations:

- `JDivideOp` in `JBinaryExpression.java`
- `JRemainderOp` in `JBinaryExpression.java`
- `JUnaryPlusOp` in `JUnaryExpression.java`

Semantics:

- The LHS and RHS operands of `/` and `%` must be ints.
- The operand of `+` must be an int.

```
>_ ~/workspace/j--
$ bash ./bin/j-- project1/Division.java
$ java Division 60 13
4
$ bash ./bin/j-- project1/Remainder.java
$ java Remainder 60 13
8
$ bash ./bin/j-- project1/UnaryPlus.java
$ java UnaryPlus 60
60
```

Directions:

- Define tokens for `/` and `%` in `TokenInfo.java`.
- Modify `Scanner.java` to scan `/` and `%`.
- Modify `Parser.java` to parse `/` and `%`, correctly capturing the precedence rules by parsing the operators in the right places.
- Implement the `analyze()` and `codegen()` methods in `JDivideOp`, `JRemainderOp`, and `JUnaryPlusOp`.

Problem 3. (*Bitwise Operations*) Implement the Java bitwise operators: unary complement `~`, inclusive or `|`, exclusive or `^`, and `&`.

AST representations:

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

Semantics:

- The operand of `~` must be an int.
- The LHS and RHS operands of `|`, `^`, and `&` must be ints.

```
>_ ~/workspace/j--
$ bash ./bin/j-- project1/BitwiseNot.java
$ java BitwiseNot 60
11111111111111111111111111000011
$ bash ./bin/j-- project1/BitwiseInclusiveOr.java
 java BitwiseInclusiveOr 60 13
111101
$ bash ./bin/j-- project1/BitwiseExclusiveOr.java
$ java BitwiseExclusiveOr 60 13
110001
$ bash ./bin/j-- project1/BitwiseAnd.java
$ java BitwiseAnd 60 13
1100
```

Directions:

- Define tokens for `~`, `|`, `^`, and `&` in `TokenInfo.java`.
- Modify `Scanner.java` to scan `~`, `|`, `^`, and `&`.
- Modify `Parser.java` to parse `~`, `|`, `^`, and `&`, capturing the precedence rules by parsing the operators in the right places.
- Implement the `analyze()` and `codegen()` methods in `JComplementOp`, `JInclusiveOrOp`, `JExclusiveOrOp`, and `JAndOp`.

Note: there are JVM instructions for `|`, `^`, and `&`, but not for `~`, which must be computed as the "exclusive or" of the operand and -1.

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

AST representations:

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

Semantics:

- The LHS and RHS operands of `<<`, `>>`, and `>>>` must be ints.

```
>_ ~/workspace/j--
$ bash ./bin/j-- project1/ALeftShift.java
$ java ALeftShift -1 16
11111111111111110000000000000000
$ bash ./bin/j-- project1/ARightShift.java
$ java ARightShift -1 16
11111111111111111111111111111111
$ bash ./bin/j-- project1/LRightShift.java
$ java LRightShift -1 16
1111111111111111
```

Directions:

- Define tokens for `<<`, `>>`, and `>>>` in `TokenInfo.java`.
- Modify `Scanner.java` to scan `<<`, `>>`, and `>>>`.
- Modify `Parser.java` to parse `<<`, `>>`, and `>>>`, capturing the precedence rules by parsing the operators in the right places.
- Implement the `analyze()` and `codegen()` methods in `JALeftShiftOp`, `JARightShiftOp`, and `JLRightShiftOp`.

Before you submit your files, make sure:

- Your code is adequately commented and follows good programming principles.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.

Files to submit:

1. `GenIsPrime.java`
2. `TokenInfo.java`
3. `Scanner.java`
4. `Parser.java`
5. `JBinaryExpression.java`
6. `JUnaryExpression.java`
7. `report.txt`