1 Exercises

Exercise 1. Consider the \textit{j-} program \texttt{Sum} shown below:

```java
package pass;
import java.lang.Integer;
import java.lang.System;

public class Sum {
    private static String MSG = "SUM = ";
    private int n;

    public Sum(int n) {
        this.n = n;
    }

    public int compute() {
        int sum = 0, i = n;
        while (i > 0) {
            sum += i --;
        }
        return sum;
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        Sum sum = new Sum(n);
        System.out.println(MSG + sum.compute());
    }
}
```

How does JVM bytecode generation (\texttt{JCompilationUnit.codegen()}) for the program work?

Exercise 2. Suppose \texttt{lhs} and \texttt{rhs} are boolean expressions. How does \textit{j-} generate code for the following statements?

- a. \texttt{boolean x = lhs && rhs;}
- b. \texttt{if (lhs && rhs) {
  then_statement
  } else {
  else_statement
  }}
- c. \texttt{while (lhs && rhs) {
  statement
  }}

Exercise 3. Suppose \(x\) is an object and \(y\) is an integer field within.

- a. What is the JVM bytecode generated for the following statement? How does the runtime stack evolve as the instructions are executed?
  \texttt{++x.y;}
- b. If \(z\) is also an integer, what is the JVM bytecode generated for the following statement? How does the runtime stack evolve as the instructions are executed?
  \texttt{z = ++x.y;}

Exercise 4. How is code generated for the expression "The first perfect number is " + 6?

Exercise 5. How is code generated for casts?

Exercise 6. How would you generate JVM bytecode for the do-while statement, ie, implement \texttt{codegen()} in \texttt{JDoWhileStatement.java}?
2 Solutions to Exercises

Solution 1. Consult sections 5.2 – 5.6 of our text.

Solution 2.

a. lhs code
   branch to Target on false
rhs code
   branch to Target on false
push 1 on stack
goto End
Target: push 0 on stack
End: ...

b. lhs code
   branch to Target on false
rhs code
   branch to Target on false
then_statement code
goto End
Target: else_statement code
End: ...

C. Test: lhs code
   branch to Target on false
rhs code
   branch to Target on false
body code
goto Test
Target: ...

Solution 3. We use table on slide 26 from the JVM Code Generation chapter.

a. Bytecode:
   aload x'
dup
getfield y
iconst_1
iadd
putfield y

Runtime stack (right to left is top to bottom):

| x |
| x | x
| x | y
| x | y | 1
| x | y+1
| ...

b. Bytecode:
   aload x
dup
getfield y
iconst_1
iadd
dup_x1
putfield y

Runtime stack (right to left is top to bottom):

| x |
| x | x
| x | y
| x | y | 1
| x | y+1
| y+1 | x | y+1
| y+1

Solution 4. Since the left-hand-side expression of + is a string, the operation denotes string concatenation, and is represented in the AST as a JStringConcatenationOp object. The codegen() method therein does the following:

1. Creates an empty string buffer, ie, a StringBuffer object, and initializes it.
2. Appends the string “The first perfect number is “ to the buffer using StringBuffer’s append(String x) method.
3. Appends the integer value 6 to the buffer using StringBuffer’s append(int x) method.
4. Invokes the `toString()` method on the buffer to produce a string on the runtime stack.

**Solution 5.** Analysis determines both the validity of a cast and the necessary converter, which encapsulates the code generated for the particular cast. Each Converter implements a method `codegen()`, which generates any code necessary to the cast. Code is first generated for the expression being cast, and then for the cast, using the appropriate converter.

**Solution 6.**

```java
public void codegen(CLEmitter output) {
    String bodyStart = output.createLabel();
    output.addLabel(bodyStart);
    body.codegen(output);
    condition.codegen(output, bodyStart, true);
}
```