```java
import components.statement.Statement;

/**
 * {@code Queue} represented as a {@code Sequence} of entries, with
 * implementations of primary methods.
 *
 * @param <T>
 *            type of {@code Queue} entries
 * @correspondence this = $this.entries
 */
public class HelloWorld {

    public static void main(String[] args) {

    }

    /**
     * Generates the sequence of virtual machine instructions ("byte codes")
     * corresponding to {@code s} and appends it at the end of {@code cp}.
     *
     * @param s
     *            the {@code Statement} for which to generate code
     * @param context
     *            the {@code Context} in which to find user defined instructions
     * @param cp
     *            the {@code Sequence} containing the generated code
     * @updates cp
     * @ensures <pre>
     * if [all instructions called in s are either primitive or
     *     defined in context]  and
     *    [context does not include any calling cycles, i.e., recursion] then
     *  cp = #cp * [sequence of virtual machine "byte codes" corresponding to s]
     * else
     *  [reports an appropriate error message to the console and terminates client]
     * </pre>
     */
```

```
37     private static void generateCodeForStatement(Statement s,
38             Map<String, Statement> context, Sequence<Integer> cp)
   {
39
40         final int dummy = 0;
41
42         switch (s.kind()) {
43             case BLOCK: {
44
45                 Statement current = s.newInstance();
46                 for (int i = 0; i < s.lengthOfBlock(); i++) {
47                     current = s.removeFromBlock(i);
48                     generateCodeForStatement(current, context,
   cp);
49                     s.addToBlock(i, current);
50                 }
51
52                 break;
53             }
54             case IF: {
55                 Statement b = s.newInstance();
56                 Condition c = s.disassembleIf(b);
57                 cp.add(cp.length(),
   conditionalJump(c).byteCode());
58                 int jump = cp.length();
59                 cp.add(cp.length(), dummy);
60                 generateCodeForStatement(b, context, cp);
61                 cp.replaceEntry(jump, cp.length());
62                 s.assembleIf(c, b);
63                 break;
64             }
65             case IF_ELSE: {
66                 Statement ifS = s.newInstance();
67                 Statement elseS = s.newInstance();
68                 Condition c = s.disassembleIfElse(ifS, elseS);
69                 cp.add(cp.length(),
   conditionalJump(c).byteCode());
70                 int jump = cp.length();
71                 cp.add(cp.length(), dummy);
72                 generateCodeForStatement(ifS, context, cp);
73                 generateCodeForStatement(elseS, context, cp);
74                 cp.replaceEntry(jump, cp.length());
75                 s.assembleIfElse(c, ifS, elseS);
76
```

```
 77                 break;
 78             }
 79             case WHILE: {
 80 
 81                 Statement whileS = s.newInstance();
 82                 Condition c = s.disassembleWhile(whileS);
 83                 cp.add(cp.length(),
    conditionalJump(c).byteCode());
 84                 int jump = cp.length();
 85                 cp.add(cp.length(), dummy);
 86                 generateCodeForStatement(whileS, context, cp);
 87                 cp.replaceEntry(jump, cp.length());
 88                 s.assembleWhile(c, whileS);
 89 
 90                 break;
 91             }
 92 
 93             case CALL: {
 94 
 95                 String label = s.disassembleCall();
 96                 if (context.hasKey(label)) {
 97                     generateCodeForStatement(context.value(label),
 98                             context.newInstance(), cp);
 99                 } else {
100                     label = label.toUpperCase();
101                     cp.add(cp.length(),
    conditionalJump(label).byteCode());
102                     label = label.toLowerCase();
103                 }
104                 s.assembleCall(label);
105 
106                 break;
107             }
108 
109         }
110     }
111 }
```