```java
import components.sequence.Sequence;

/**
 * {@code Statement} represented as a {@code Tree<StatementLabel>} with
 * implementations of primary methods.
 *
 * @convention [$this.rep is a valid representation of a Statement]
 * @correspondence this = $this.rep
 *
 * @author Zhuoyang Li + Xinci Ma
 *
 */
public class Statement2 extends StatementSecondary {

    /*
     * Private members
    --------------------------------------------------------------
     */

    /**
     * Label class for the tree representation.
     */
    private static final class StatementLabel {

        /**
         * Statement kind.
         */
        private Kind kind;

        /**
         * IF/IF_ELSE/WHILE statement condition.
         */
        private Condition condition;

        /**
         * CALL instruction name.
         */
        private String instruction;

        /**
         * Constructor for BLOCK.
         *
         * @param k
         *            the kind of statement
         *
```

```
50          * @requires k = BLOCK
51          * @ensures this = (BLOCK, ?, ?)
52          */
53         private StatementLabel(Kind k) {
54             assert k == Kind.BLOCK : "Violation of: k =
   BLOCK";
55             this.kind = k;
56         }
57
58         /**
59          * Constructor for IF, IF_ELSE, WHILE.
60          *
61          * @param k
62          *            the kind of statement
63          * @param c
64          *            the statement condition
65          *
66          * @requires k = IF or k = IF_ELSE or k = WHILE
67          * @ensures this = (k, c, ?)
68          */
69         private StatementLabel(Kind k, Condition c) {
70             assert k == Kind.IF || k == Kind.IF_ELSE || k ==
   Kind.WHILE : ""
71                     + "Violation of: k = IF or k = IF_ELSE or
   k = WHILE";
72             this.kind = k;
73             this.condition = c;
74         }
75
76         /**
77          * Constructor for CALL.
78          *
79          * @param k
80          *            the kind of statement
81          * @param i
82          *            the instruction name
83          *
84          * @requires k = CALL and [i is an IDENTIFIER]
85          * @ensures this = (CALL, ?, i)
86          */
87         private StatementLabel(Kind k, String i) {
88             assert k == Kind.CALL : "Violation of: k = CALL";
89             assert i != null : "Violation of: i is not null";
90             assert Tokenizer
91                     .isIdentifier(i) : "Violation of: i is an
   IDENTIFIER";
92             this.kind = k;
```

```
 93              this.instruction = i;
 94          }
 95
 96          @Override
 97          public String toString() {
 98              String condition = "?", instruction = "?";
 99              if ((this.kind == Kind.IF) || (this.kind ==
    Kind.IF_ELSE)
100                      || (this.kind == Kind.WHILE)) {
101                  condition = this.condition.toString();
102              } else if (this.kind == Kind.CALL) {
103                  instruction = this.instruction;
104              }
105              return "(" + this.kind + "," + condition + "," +
    instruction + ")";
106          }
107
108      }
109
110      /**
111       * The tree representation field.
112       */
113      private Tree<StatementLabel> rep;
114
115      /**
116       * Creator of initial representation.
117       */
118      private void createNewRep() {
119
120          this.rep = new Tree1<>();
121          StatementLabel rootLabel = new
    StatementLabel(Kind.BLOCK);
122          Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
123          this.rep.assemble(rootLabel, children);
124
125      }
126
127      /*
128       * Constructors
    ------------------------------------------------------------
129       */
130
131      /**
132       * No-argument constructor.
133       */
134      public Statement2() {
```

```
135         this.createNewRep();
136     }
137
138     /*
139      * Standard methods
    ------------------------------------------------------------
140      */
141
142     @Override
143     public final Statement2 newInstance() {
144         try {
145             return
    this.getClass().getConstructor().newInstance();
146         } catch (ReflectiveOperationException e) {
147             throw new AssertionError(
148                     "Cannot construct object of type " +
    this.getClass());
149         }
150     }
151
152     @Override
153     public final void clear() {
154         this.createNewRep();
155     }
156
157     @Override
158     public final void transferFrom(Statement source) {
159         assert source != null : "Violation of: source is not
    null";
160         assert source != this : "Violation of: source is not
    this";
161         assert source instanceof Statement2 : ""
162                 + "Violation of: source is of dynamic type
    Statement2";
163         /*
164          * This cast cannot fail since the assert above would
    have stopped
165          * execution in that case: source must be of dynamic
    type Statement2.
166          */
167         Statement2 localSource = (Statement2) source;
168         this.rep = localSource.rep;
169         localSource.createNewRep();
170     }
171
172     /*
173      * Kernel methods
```

```
174      */
175
176     @Override
177     public final Kind kind() {
178
179         return this.rep.root().kind;
180     }
181
182     @Override
183     public final void addToBlock(int pos, Statement s) {
184         assert s != null : "Violation of: s is not null";
185         assert s != this : "Violation of: s is not this";
186         assert s instanceof Statement2 : "Violation of: s is
    a Statement2";
187         assert this.kind() == Kind.BLOCK : ""
188                 + "Violation of: [this is a BLOCK
    statement]";
189         assert 0 <= pos : "Violation of: 0 <= pos";
190         assert pos <= this.lengthOfBlock() : ""
191                 + "Violation of: pos <= [length of this
    BLOCK]";
192         assert s.kind() != Kind.BLOCK : "Violation of: [s is
    not a BLOCK statement]";
193
194         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
195         Statement2 localS = (Statement2) s;
196         StatementLabel label =
    this.rep.disassemble(children);
197         children.add(pos, localS.rep);
198         this.rep.assemble(label, children);
199         localS.createNewRep(); // clear s
200
201     }
202
203     @Override
204     public final Statement removeFromBlock(int pos) {
205         assert 0 <= pos : "Violation of: 0 <= pos";
206         assert pos < this.lengthOfBlock() : ""
207                 + "Violation of: pos < [length of this
    BLOCK]";
208         assert this.kind() == Kind.BLOCK : ""
209                 + "Violation of: [this is a BLOCK
    statement]";
210         /*
211          * The following call to Statement newInstance method
```

```
    is a violation of
212          * the kernel purity rule. However, there is no way
    to avoid it and it
213          * is safe because the convention clearly holds at
    this point in the
214          * code.
215          */
216         Statement2 s = this.newInstance();
217
218         Tree<StatementLabel> removedSubtree =
    this.rep.removeSubtree(pos);
219         Statement2 removedStatement = new Statement2();
220         removedStatement.rep = removedSubtree;
221         return removedStatement;
222     }
223
224     @Override
225     public final int lengthOfBlock() {
226         assert this.kind() == Kind.BLOCK : ""
227                 + "Violation of: [this is a BLOCK
    statement]";
228
229         return this.rep.numberOfSubtrees();
230     }
231
232     @Override
233     public final void assembleIf(Condition c, Statement s) {
234         assert c != null : "Violation of: c is not null";
235         assert s != null : "Violation of: s is not null";
236         assert s != this : "Violation of: s is not this";
237         assert s instanceof Statement2 : "Violation of: s is
    a Statement2";
238         assert s.kind() == Kind.BLOCK : ""
239                 + "Violation of: [s is a BLOCK statement]";
240         Statement2 localS = (Statement2) s;
241         StatementLabel label = new StatementLabel(Kind.IF,
    c);
242         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
243         children.add(0, localS.rep);
244         this.rep.assemble(label, children);
245         localS.createNewRep(); // clears s
246     }
247
248     @Override
249     public final Condition disassembleIf(Statement s) {
250         assert s != null : "Violation of: s is not null";
```

```
251         assert s != this : "Violation of: s is not this";
252         assert s instanceof Statement2 : "Violation of: s is
    a Statement2";
253         assert this.kind() == Kind.IF : ""
254                 + "Violation of: [this is an IF statement]";
255         Statement2 localS = (Statement2) s;
256         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
257         StatementLabel label =
    this.rep.disassemble(children);
258         localS.rep = children.remove(0);
259         this.createNewRep(); // clears this
260         return label.condition;
261     }
262
263     @Override
264     public final void assembleIfElse(Condition c, Statement
    s1, Statement s2) {
265         assert c != null : "Violation of: c is not null";
266         assert s1 != null : "Violation of: s1 is not null";
267         assert s2 != null : "Violation of: s2 is not null";
268         assert s1 != this : "Violation of: s1 is not this";
269         assert s2 != this : "Violation of: s2 is not this";
270         assert s1 != s2 : "Violation of: s1 is not s2";
271         assert s1 instanceof Statement2 : "Violation of: s1
    is a Statement2";
272         assert s2 instanceof Statement2 : "Violation of: s2
    is a Statement2";
273         assert s1
274                 .kind() == Kind.BLOCK : "Violation of: [s1 is
    a BLOCK statement]";
275         assert s2
276                 .kind() == Kind.BLOCK : "Violation of: [s2 is
    a BLOCK statement]";
277
278         Statement2 thenStatement = (Statement2) s1;
279         Statement2 elseStatement = (Statement2) s2;
280         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
281
282         children.add(0, thenStatement.rep);
283         children.add(1, elseStatement.rep);
284         this.rep.assemble(new StatementLabel(Kind.IF_ELSE,
    c), children);
285
286         thenStatement.createNewRep(); // clear input
    statement
```

```
287         elseStatement.createNewRep();
288
289     }
290
291     @Override
292     public final Condition disassembleIfElse(Statement s1,
    Statement s2) {
293         assert s1 != null : "Violation of: s1 is not null";
294         assert s2 != null : "Violation of: s1 is not null";
295         assert s1 != this : "Violation of: s1 is not this";
296         assert s2 != this : "Violation of: s2 is not this";
297         assert s1 != s2 : "Violation of: s1 is not s2";
298         assert s1 instanceof Statement2 : "Violation of: s1
    is a Statement2";
299         assert s2 instanceof Statement2 : "Violation of: s2
    is a Statement2";
300         assert this.kind() == Kind.IF_ELSE : ""
301                 + "Violation of: [this is an IF_ELSE
    statement]";
302
303         Statement2 thenStatement = (Statement2) s1;
304         Statement2 elseStatement = (Statement2) s2;
305         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
306
307         StatementLabel label =
    this.rep.disassemble(children);
308         thenStatement.rep = children.remove(0);
309         elseStatement.rep = children.remove(0);
310         this.createNewRep();
311
312         return label.condition;
313     }
314
315     @Override
316     public final void assembleWhile(Condition c, Statement s)
    {
317         assert c != null : "Violation of: c is not null";
318         assert s != null : "Violation of: s is not null";
319         assert s != this : "Violation of: s is not this";
320         assert s instanceof Statement2 : "Violation of: s is
    a Statement2";
321         assert s.kind() == Kind.BLOCK : "Violation of: [s is
    a BLOCK statement]";
322
323         // casting s to Statement2 to work with
    representation
```

```
324         Statement2 sAsStatement2 = (Statement2) s;
325
326         // creating new sequence for children of WHILE
    statement
327         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
328
329         // adding representation of s as only child of WHILE
    statement
330         children.add(0, sAsStatement2.rep);
331
332         // assembling WHILE statement with condition and
    single child
333         this.rep.assemble(new StatementLabel(Kind.WHILE, c),
    children);
334
335         // clearing original statement s to ensure solely
    part of WHILE structure
336         sAsStatement2.createNewRep();
337     }
338
339     @Override
340     public final Condition disassembleWhile(Statement s) {
341         assert s != null : "Violation of: s is not null";
342         assert s != this : "Violation of: s is not this";
343         assert s instanceof Statement2 : "Violation of: s is
    a Statement2";
344         assert this.kind() == Kind.WHILE : ""
345                 + "Violation of: [this is a WHILE
    statement]";
346
347         // preparing to extract children (body) of WHILE
    statement
348         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
349
350         // extracting label (contains condition) and body
351         StatementLabel label =
    this.rep.disassemble(children);
352
353         // casting s to Statement2 to modify to represent
    body of WHILE
354         Statement2 bodyStatement = (Statement2) s;
355
356         // assuming WHILE has one body statement
357         bodyStatement.rep = children.remove(0);
358         this.createNewRep();
```

```
359
360         // returning condition part of WHILE statement
361         return label.condition;
362     }
363
364     @Override
365     public final void assembleCall(String inst) {
366         assert inst != null : "Violation of: inst is not
    null";
367         assert Tokenizer.isIdentifier(inst) : ""
368                 + "Violation of: inst is a valid IDENTIFIER";
369
370         // create label for CALL with instruction name
371         StatementLabel label = new StatementLabel(Kind.CALL,
    inst);
372
373         // as CALL statements have no children, create empty
    sequence for children
374         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
375
376         // assembling CALL statement with label and no
    children
377         this.rep.assemble(label, children);
378
379     }
380
381     @Override
382     public final String disassembleCall() {
383         assert this.kind() == Kind.CALL : ""
384                 + "Violation of: [this is a CALL statement]";
385
386         // preparing to extract potential children
387         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
388
389         // extracting label which contains the instruction
    name
390         StatementLabel label =
    this.rep.disassemble(children);
391
392         // clearing representation to return only instruction
    name
393         this.createNewRep();
394
395         // returning instruction name part of CALL statement
396         return label.instruction;
```

```
397     }
398
399 }
400
```