```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 Shyam Sai Bethina and Yihone Chu
 *
 */
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.
         *
```

```
47          * @param k
48          *            the kind of statement
49          */
50         private StatementLabel(Kind k) {
51             assert k == Kind.BLOCK : "Violation of: k = BLOCK";
52             this.kind = k;
53         }
54
55         /**
56          * Constructor for IF, IF_ELSE, WHILE.
57          *
58          * @param k
59          *            the kind of statement
60          * @param c
61          *            the statement condition
62          */
63         private StatementLabel(Kind k, Condition c) {
64             assert k == Kind.IF || k == Kind.IF_ELSE || k ==
    Kind.WHILE : ""
65                     + "Violation of: k = IF or k = IF_ELSE or k
    = WHILE";
66             this.kind = k;
67             this.condition = c;
68         }
69
70         /**
71          * Constructor for CALL.
72          *
73          * @param k
74          *            the kind of statement
75          * @param i
76          *            the instruction name
77          */
78         private StatementLabel(Kind k, String i) {
79             assert k == Kind.CALL : "Violation of: k = CALL";
80             assert i != null : "Violation of: i is not null";
81             assert Tokenizer
82                     .isIdentifier(i) : "Violation of: i is an
    IDENTIFIER";
83             this.kind = k;
84             this.instruction = i;
85         }
86
87         @Override
```

```
 88         public String toString() {
 89             String condition = "?", instruction = "?";
 90             if ((this.kind == Kind.IF) || (this.kind ==
    Kind.IF_ELSE)
 91                     || (this.kind == Kind.WHILE)) {
 92                 condition = this.condition.toString();
 93             } else if (this.kind == Kind.CALL) {
 94                 instruction = this.instruction;
 95             }
 96             return "(" + this.kind + "," + condition + "," +
    instruction + ")";
 97         }
 98 
 99     }
100 
101     /**
102      * The tree representation field.
103      */
104     private Tree<StatementLabel> rep;
105 
106     /**
107      * Creator of initial representation.
108      */
109     private void createNewRep() {
110         /*
111          * Creates a new representation by setting the root of
    the rep to
112          * Kind.Block, and the children are empty Trees of
    statement labels.
113          */
114         this.rep = new Tree1<StatementLabel>();
115         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
116         StatementLabel s = new StatementLabel(Kind.BLOCK);
117         this.rep.assemble(s, children);
118     }
119 
120     /*
121      * Constructors
    ------------------------------------------------------------
122      */
123 
124     /**
125      * No-argument constructor.
```

```
126     */
127    public Statement2() {
128        /*
129         * Creates a new representation by
    calling .createNewRep()
130         */
131        this.createNewRep();
132    }
133
134    /*
135     * Standard methods
    ---------------------------------------------------------
136     */
137
138    @Override
139    public final Statement2 newInstance() {
140        try {
141            return
    this.getClass().getConstructor().newInstance();
142        } catch (ReflectiveOperationException e) {
143            throw new AssertionError(
144                    "Cannot construct object of type " +
    this.getClass());
145        }
146    }
147
148    @Override
149    public final void clear() {
150        this.createNewRep();
151    }
152
153    @Override
154    public final void transferFrom(Statement source) {
155        assert source != null : "Violation of: source is not
    null";
156        assert source != this : "Violation of: source is not
    this";
157        assert source instanceof Statement2 : ""
158                + "Violation of: source is of dynamic type
    Statement2";
159        /*
160         * This cast cannot fail since the assert above would
    have stopped
161         * execution in that case: source must be of dynamic
```

```
   type Statement2.
162          */
163         Statement2 localSource = (Statement2) source;
164         this.rep = localSource.rep;
165         localSource.createNewRep();
166     }
167 
168     /*
169      * Kernel methods
   ------------------------------------------------------------
170      */
171 
172     @Override
173     public final Kind kind() {
174         /*
175          * Returns the kind of this' representation's root.
176          */
177         return this.rep.root().kind;
178     }
179 
180     @Override
181     public final void addToBlock(int pos, Statement s) {
182         assert s != null : "Violation of: s is not null";
183         assert s instanceof Statement2 : "Violation of: s is a
   Statement2";
184         assert this.kind() == Kind.BLOCK : ""
185                 + "Violation of: [this is a BLOCK statement]";
186         assert 0 <= pos : "Violation of: 0 <= pos";
187         assert pos <= this.lengthOfBlock() : ""
188                 + "Violation of: pos <= [length of this
   BLOCK]";
189         assert s.kind() != Kind.BLOCK : "Violation of: [s is
   not a BLOCK statement]";
190         //use this to access s sequence
191         Statement2 locals = (Statement2) s;
192 
193         /*
194          * Creates an empty sequence of trees of
   StatementLabels, and
195          * disassembles the representation.
196          */
197         Sequence<Tree<StatementLabel>> children =
   this.rep.newSequenceOfTree();
198         StatementLabel label = this.rep.disassemble(children);
```

```
199
200         /*
201          * Add the representation of locals to the position
    from the parameters.
202          */
203         children.add(pos, locals.rep);
204
205         /*
206          * Gets rid of the locals, and reassembles the
    representation with the
207          * new block.
208          */
209         locals.createNewRep(); //gets rid of locals
210         this.rep.assemble(label, children);
211     }
212
213     @Override
214     public final Statement removeFromBlock(int pos) {
215         assert 0 <= pos : "Violation of: 0 <= pos";
216         assert pos < this.lengthOfBlock() : ""
217                 + "Violation of: pos < [length of this BLOCK]";
218         assert this.kind() == Kind.BLOCK : ""
219                 + "Violation of: [this is a BLOCK statement]";
220         /*
221          * The following call to Statement newInstance method
    is a violation of
222          * the kernel purity rule. However, there is no way to
    avoid it and it
223          * is safe because the convention clearly holds at this
    point in the
224          * code.
225          */
226         Statement2 temp = new Statement2();
227
228         /*
229          * Removes the subtree's representation at the position
    from the
230          * parameter list, and returns it.
231          */
232         temp.rep = this.rep.removeSubtree(pos);
233
234         return temp;
235     }
236
```

```
237     @Override
238     public final int lengthOfBlock() {
239         assert this.kind() == Kind.BLOCK : ""
240                 + "Violation of: [this is a BLOCK statement]";
241
242         /*
243          * Returns the number of subtrees in the
    representation.
244          */
245         return this.rep.numberOfSubtrees();
246     }
247
248     @Override
249     public final void assembleIf(Condition c, Statement s) {
250         assert c != null : "Violation of: c is not null";
251         assert s != null : "Violation of: s is not null";
252         assert s instanceof Statement2 : "Violation of: s is a
    Statement2";
253         assert s.kind() == Kind.BLOCK : ""
254                 + "Violation of: [s is a BLOCK statement]";
255
256         Statement2 locals = (Statement2) s;
257         StatementLabel newLabel = new StatementLabel(Kind.IF,
    c);
258
259         /*
260          * Creates an empty sequence of trees of
    StatementLabels, and
261          * disassembles the representation.
262          */
263         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
264
265         /*
266          * Add the representation of locals to the position
    from the parameters.
267          */
268         children.add(0, locals.rep);
269
270         /*
271          * Gets rid of the locals, and reassembles the
    representation with the
272          * new if block.
273          */
```

```
274         this.rep.assemble(newLabel, children);
275         locals.createNewRep(); // clears s
276     }
277 
278     @Override
279     public final Condition disassembleIf(Statement s) {
280         assert s != null : "Violation of: s is not null";
281         assert s instanceof Statement2 : "Violation of: s is a
    Statement2";
282         assert this.kind() == Kind.IF : "Violation of: [s is an
    IF statement]";
283 
284         Statement2 localS = (Statement2) s;
285 
286         /*
287          * Creates an empty sequence of trees of
    StatementLabels, and
288          * disassembles the representation.
289          */
290         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
291         StatementLabel label = this.rep.disassemble(children);
292 
293         /*
294          * Removes the first block from the representation.
295          */
296         localS.rep = children.remove(0);
297 
298         this.createNewRep(); // clears this
299 
300         /*
301          * Returns the condition of the label.
302          */
303         return label.condition;
304     }
305 
306     @Override
307     public final void assembleIfElse(Condition c, Statement s1,
    Statement s2) {
308         assert c != null : "Violation of: c is not null";
309         assert s1 != null : "Violation of: s1 is not null";
310         assert s2 != null : "Violation of: s2 is not null";
311         assert s1 instanceof Statement2 : "Violation of: s1 is
    a Statement2";
```

```
312         assert s2 instanceof Statement2 : "Violation of: s2 is
    a Statement2";
313         assert s1
314                 .kind() == Kind.BLOCK : "Violation of: [s1 is a
    BLOCK statement]";
315         assert s2
316                 .kind() == Kind.BLOCK : "Violation of: [s2 is a
    BLOCK statement]";
317
318         Statement2 locals1 = (Statement2) s1;
319         Statement2 locals2 = (Statement2) s2;
320
321         /*
322          * Creates an empty sequence of trees of
    StatementLabels, and
323          * disassembles the representation.
324          */
325         StatementLabel newLabel = new
    StatementLabel(Kind.IF_ELSE, c);
326         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
327
328         /*
329          * Add the representation of locals to the positions
    from the
330          * parameters.
331          */
332         children.add(0, locals1.rep);
333         children.add(1, locals2.rep);
334
335         /*
336          * Gets rid of the locals, and reassembles the
    representation with the
337          * new if else block.
338          */
339         this.rep.assemble(newLabel, children);
340
341         //do this to get rid of the local statements
342         locals1.createNewRep();
343         locals2.createNewRep();
344     }
345
346     @Override
347     public final Condition disassembleIfElse(Statement s1,
```

```
    Statement s2) {
348         assert s1 != null : "Violation of: s1 is not null";
349         assert s2 != null : "Violation of: s1 is not null";
350         assert s1 instanceof Statement2 : "Violation of: s1 is
    a Statement2";
351         assert s2 instanceof Statement2 : "Violation of: s2 is
    a Statement2";
352         assert this
353                 .kind() == Kind.IF_ELSE : "Violation of: [s is
    an IF_ELSE statement]";
354
355         Statement2 localS1 = (Statement2) s1;
356         Statement2 localS2 = (Statement2) s2;
357
358         /*
359          * Creates an empty sequence of trees of
    StatementLabels, and
360          * disassembles the representation.
361          */
362         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
363         StatementLabel label = this.rep.disassemble(children);
364
365         /*
366          * Removes the first block from the representation.
367          */
368         localS1.rep = children.remove(0);
369         localS2.rep = children.remove(0);
370
371         this.createNewRep(); // clears this
372
373         /*
374          * Returns the condition of the label.
375          */
376         return label.condition;
377     }
378
379     @Override
380     public final void assembleWhile(Condition c, Statement s) {
381         assert c != null : "Violation of: c is not null";
382         assert s != null : "Violation of: s is not null";
383         assert s instanceof Statement2 : "Violation of: s is a
    Statement2";
384         assert s.kind() == Kind.BLOCK : "Violation of: [s is a
```

```
    BLOCK statement]";
385
386         Statement2 localS = (Statement2) s;
387         StatementLabel newLabel = new
    StatementLabel(Kind.WHILE, c);
388
389         /*
390          * Creates an empty sequence of trees of
    StatementLabels, and
391          * disassembles the representation.
392          */
393         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
394
395         /*
396          * Add the representation of locals to the positions
    from the
397          * parameters.
398          */
399         children.add(0, localS.rep);
400
401         /*
402          * Gets rid of the locals, and reassembles the
    representation with the
403          * new while block.
404          */
405         this.rep.assemble(newLabel, children);
406
407         localS.createNewRep(); // clears
408
409     }
410
411     @Override
412     public final Condition disassembleWhile(Statement s) {
413         assert s != null : "Violation of: s is not null";
414         assert s instanceof Statement2 : "Violation of: s is a
    Statement2";
415         assert this
416                 .kind() == Kind.WHILE : "Violation of: [s is a
    WHILE statement]";
417
418         Statement2 localS = (Statement2) s;
419
420         /*
```

```
421          * Creates an empty sequence of trees of
    StatementLabels, and
422          * disassembles the representation.
423          */
424         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
425         StatementLabel label = this.rep.disassemble(children);
426
427         /*
428          * Removes the first block from the representation.
429          */
430         localS.rep = children.remove(0);
431
432         this.createNewRep(); // clears this
433
434         /*
435          * Returns the condition of the label.
436          */
437         return label.condition;
438     }
439
440     @Override
441     public final void assembleCall(String inst) {
442         assert inst != null : "Violation of: inst is not null";
443         assert Tokenizer.isIdentifier(inst) : ""
444                 + "Violation of: inst is a valid IDENTIFIER";
445
446         StatementLabel newLabel = new StatementLabel(Kind.CALL,
    inst);
447         /*
448          * Creates an empty sequence of trees of
    StatementLabels, and
449          * disassembles the representation.
450          */
451         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
452
453         /*
454          * Reassembles the representation with the new call.
455          */
456         this.rep.assemble(newLabel, children);
457     }
458
459     @Override
```

```
460     public final String disassembleCall() {
461         assert this
462                 .kind() == Kind.CALL : "Violation of: [s is a
    CALL statement]";
463 
464         /*
465          * Creates an empty sequence of trees of
    StatementLabels, and
466          * disassembles the representation.
467          */
468         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
469         StatementLabel label = this.rep.disassemble(children);
470 
471         this.createNewRep(); //clears this
472 
473         /*
474          * Returns the condition of the label.
475          */
476         return label.instruction;
477 
478     }
479 
480 }
481 
```