

```

1 import components.sequence.Sequence;
2
3 /**
4  * {@code Statement} represented as a {@code Tree<StatementLabel>} with
5  * implementations of primary methods.
6  *
7  * @convention [$this.rep is a valid representation of a Statement]
8  * @correspondence this = $this.rep
9  *
10 * @author Krish Patel and Chloe Feller
11 */
12 public class Statement2 extends StatementSecondary {
13
14     /*
15      * Private members -----
16      */
17
18     /**
19      * Label class for the tree representation.
20      */
21     private static final class StatementLabel {
22
23         /**
24          * Statement kind.
25          */
26         private Kind kind;
27
28         /**
29          * IF/IF_ELSE/WHILE statement condition.
30          */
31         private Condition condition;
32
33         /**
34          * CALL instruction name.
35          */
36         private String instruction;
37
38         /**
39          * Constructor for BLOCK.
40          *
41          * @param k
42          *         the kind of statement
43          *
44          * @requires k = BLOCK
45          * @ensures this = (BLOCK, ?, ?)
46          */
47         private StatementLabel(Kind k) {
48             assert k == Kind.BLOCK : "Violation of: k = BLOCK";
49             this.kind = k;
50         }
51
52         /**
53          * Constructor for IF, IF_ELSE, WHILE.
54          *
55          * @param k
56          *         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<StatementLabel>();
121     StatementLabel emptyBlock = new StatementLabel(Kind.BLOCK);
122     Sequence<Tree<Statement2.StatementLabel>> emptyTree = this.rep
123         .newSequenceOfTree();
124     this.rep.assemble(emptyBlock, emptyTree);
125
126 }
127
128 /*
129  * Constructors -----
130  */
131
132 /**
133  * No-argument constructor.
134  */
135 public Statement2() {
136     this.createNewRep();
137 }
138
139 /*
140  * Standard methods -----
141  */
142
143 @Override
144 public final Statement2 newInstance() {
145     try {
146         return this.getClass().getConstructor().newInstance();
147     } catch (ReflectiveOperationException e) {
148         throw new AssertionError(
149             "Cannot construct object of type " + this.getClass());
150     }
151 }
152
153 @Override
154 public final void clear() {
155     this.createNewRep();
156 }
157
158 @Override
159 public final void transferFrom(Statement source) {
160     assert source != null : "Violation of: source is not null";
161     assert source != this : "Violation of: source is not this";
162     assert source instanceof Statement2 : ""
163         + "Violation of: source is of dynamic type Statement2";
164     /*
165      * This cast cannot fail since the assert above would have stopped
166      * execution in that case: source must be of dynamic type Statement2.
167      */
168     Statement2 localSource = (Statement2) source;
169     this.rep = localSource.rep;
170     localSource.createNewRep();
171 }
172
173 /*
174  * Kernel methods -----
175  */
176

```

```

177     @Override
178     public final Kind kind() {
179
180         StatementLabel s = this.rep.root();
181         Kind k = s.kind;
182
183         return k;
184     }
185
186     @Override
187     public final void addToBlock(int pos, Statement s) {
188         assert s != null : "Violation of: s is not null";
189         assert s != this : "Violation of: s is not this";
190         assert s instanceof Statement2 : "Violation of: s is a Statement2";
191         assert this.kind() == Kind.BLOCK : ""
192             + "Violation of: [this is a BLOCK statement]";
193         assert 0 <= pos : "Violation of: 0 <= pos";
194         assert pos <= this.lengthOfBlock() : ""
195             + "Violation of: pos <= [length of this BLOCK]";
196         assert s.kind() != Kind.BLOCK : "Violation of: [s is not a BLOCK statement]";
197
198         Sequence<Tree<StatementLabel>> child = this.rep.newSequenceOfTree();
199         StatementLabel label = this.rep.disassemble(child);
200         Statement2 local = (Statement2) s;
201
202         child.add(pos, local.rep);
203
204         this.rep.assemble(label, child);
205         local.createNewRep();
206
207     }
208
209     @Override
210     public final Statement removeFromBlock(int pos) {
211         assert 0 <= pos : "Violation of: 0 <= pos";
212         assert pos < this.lengthOfBlock() : ""
213             + "Violation of: pos < [length of this BLOCK]";
214         assert this.kind() == Kind.BLOCK : ""
215             + "Violation of: [this is a BLOCK statement]";
216         /*
217          * The following call to Statement newInstance method is a violation of
218          * the kernel purity rule. However, there is no way to avoid it and it
219          * is safe because the convention clearly holds at this point in the
220          * code.
221          */
222         Statement2 s = this.newInstance();
223
224         Sequence<Tree<StatementLabel>> sequence = this.rep.newSequenceOfTree();
225         StatementLabel kind = this.rep.disassemble(sequence);
226
227         Tree<StatementLabel> removed = sequence.remove(pos);
228         s.rep.transferFrom(removed);
229         this.rep.assemble(kind, sequence);
230
231         return s;
232     }
233

```

```

234     @Override
235     public final int lengthOfBlock() {
236         assert this.kind() == Kind.BLOCK : ""
237             + "Violation of: [this is a BLOCK statement]";
238
239         Sequence<Tree<StatementLabel>> sequence = this.rep.newSequenceOfTree();
240         StatementLabel kind = this.rep.disassemble(sequence);
241
242         int length = sequence.length();
243
244         this.rep.assemble(kind, sequence);
245
246         return length;
247     }
248
249     @Override
250     public final void assembleIf(Condition c, Statement s) {
251         assert c != null : "Violation of: c is not null";
252         assert s != null : "Violation of: s is not null";
253         assert s != this : "Violation of: s is not this";
254         assert s instanceof Statement2 : "Violation of: s is a Statement2";
255         assert s.kind() == Kind.BLOCK : ""
256             + "Violation of: [s is a BLOCK statement]";
257         Statement2 localS = (Statement2) s;
258         StatementLabel label = new StatementLabel(Kind.IF, c);
259         Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
260         children.add(0, localS.rep);
261         this.rep.assemble(label, children);
262         localS.createNewRep(); // clears s
263     }
264
265     @Override
266     public final Condition disassembleIf(Statement s) {
267         assert s != null : "Violation of: s is not null";
268         assert s != this : "Violation of: s is not this";
269         assert s instanceof Statement2 : "Violation of: s is a Statement2";
270         assert this.kind() == Kind.IF : ""
271             + "Violation of: [this is an IF statement]";
272         Statement2 localS = (Statement2) s;
273         Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
274         StatementLabel label = this.rep.disassemble(children);
275         localS.rep = children.remove(0);
276         this.createNewRep(); // clears this
277         return label.condition;
278     }
279
280     @Override
281     public final void assembleIfElse(Condition c, Statement s1, Statement s2) {
282         assert c != null : "Violation of: c is not null";
283         assert s1 != null : "Violation of: s1 is not null";
284         assert s2 != null : "Violation of: s2 is not null";
285         assert s1 != this : "Violation of: s1 is not this";
286         assert s2 != this : "Violation of: s2 is not this";
287         assert s1 != s2 : "Violation of: s1 is not s2";
288         assert s1 instanceof Statement2 : "Violation of: s1 is a Statement2";
289         assert s2 instanceof Statement2 : "Violation of: s2 is a Statement2";
290         assert s1

```

```

291         .kind() == Kind.BLOCK : "Violation of: [s1 is a BLOCK statement]";
292     assert s2
293         .kind() == Kind.BLOCK : "Violation of: [s2 is a BLOCK statement]";
294
295     Statement2 localS = (Statement2) s1;
296     Statement2 localS2 = (Statement2) s2;
297     StatementLabel label = new StatementLabel(Kind.IF_ELSE, c);
298     Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
299     children.add(0, localS.rep);
300     children.add(1, localS2.rep);
301     this.rep.assemble(label, children);
302     localS.createNewRep(); // clears s
303     localS2.createNewRep();
304
305 }
306
307 @Override
308 public final Condition disassembleIfElse(Statement s1, Statement s2) {
309     assert s1 != null : "Violation of: s1 is not null";
310     assert s2 != null : "Violation of: s2 is not null";
311     assert s1 != this : "Violation of: s1 is not this";
312     assert s2 != this : "Violation of: s2 is not this";
313     assert s1 != s2 : "Violation of: s1 is not s2";
314     assert s1 instanceof Statement2 : "Violation of: s1 is a Statement2";
315     assert s2 instanceof Statement2 : "Violation of: s2 is a Statement2";
316     assert this.kind() == Kind.IF_ELSE : ""
317         + "Violation of: [this is an IF_ELSE statement]";
318
319     Statement2 local1 = (Statement2) s1;
320     Statement2 local2 = (Statement2) s2;
321     Sequence<Tree<StatementLabel>> child = this.rep.newSequenceOfTree();
322     StatementLabel label = this.rep.disassemble(child);
323
324     local2.rep = child.remove(1);
325     local1.rep = child.remove(0);
326
327     this.createNewRep();
328
329     return label.condition;
330 }
331
332 @Override
333 public final void assembleWhile(Condition c, Statement s) {
334     assert c != null : "Violation of: c is not null";
335     assert s != null : "Violation of: s is not null";
336     assert s != this : "Violation of: s is not this";
337     assert s instanceof Statement2 : "Violation of: s is a Statement2";
338     assert s.kind() == Kind.BLOCK : "Violation of: [s is a BLOCK statement]";
339
340     Statement2 localS = (Statement2) s;
341     StatementLabel label = new StatementLabel(Kind.WHILE, c);
342     Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
343     children.add(0, localS.rep);
344     this.rep.assemble(label, children);
345     localS.createNewRep(); // clears s
346
347 }

```

```
348
349     @Override
350     public final Condition disassembleWhile(Statement s) {
351         assert s != null : "Violation of: s is not null";
352         assert s != this : "Violation of: s is not this";
353         assert s instanceof Statement2 : "Violation of: s is a Statement2";
354         assert this.kind() == Kind.WHILE : ""
355             + "Violation of: [this is a WHILE statement]";
356
357         Sequence<Tree<StatementLabel>> child = this.rep.newSequenceOfTree();
358         StatementLabel label = this.rep.disassemble(child);
359         Condition c = label.condition;
360
361         Statement2 local = (Statement2) s;
362         local.rep = child.remove(0);
363
364         this.createNewRep();
365
366         return c;
367     }
368
369     @Override
370     public final void assembleCall(String inst) {
371         assert inst != null : "Violation of: inst is not null";
372         assert Tokenizer.isIdentifier(inst) : ""
373             + "Violation of: inst is a valid IDENTIFIER";
374
375         StatementLabel label = new StatementLabel(Kind.CALL, inst);
376         Sequence<Tree<StatementLabel>> child = this.rep.newSequenceOfTree();
377
378         this.rep.assemble(label, child);
379
380     }
381
382     @Override
383     public final String disassembleCall() {
384         assert this.kind() == Kind.CALL : ""
385             + "Violation of: [this is a CALL statement]";
386
387         Sequence<Tree<StatementLabel>> child = this.rep.newSequenceOfTree();
388         StatementLabel label = this.rep.disassemble(child);
389
390         String inst = label.instruction;
391
392         this.createNewRep();
393
394         return inst;
395     }
396
397 }
398
```