THE OHIO STATE UNIVERSITY

PROJECT 7: IMPLEMENTATION OF PROGRAM AND STATEMENT KERNELS

Daniil Gofman

Ansh Pachauri

SW 2: Dev & Dsgn

Paolo Bucci

Yiyang Chen

Shivam Gupta

November 1, 2023

```
 1 import components.map.Map;
 2 import components.map.Map1L;
 3 import components.program.Program;
 4 import components.program.ProgramSecondary;
 5 import components.statement.Statement;
 6 import components.statement.Statement1;
 7 import components.statement.StatementKernel.Kind;
 8 import components.utilities.Tokenizer;
 9
10 /**
11  * {@code Program} represented the obvious way with
   implementations of primary
12  * methods.
13  *
14  * @convention [$this.name is an IDENTIFIER] and [$this.context
   is a CONTEXT]
15  *             and [$this.body is a BLOCK statement]
16  * @correspondence this = ($this.name, $this.context,
   $this.body)
17  *
18  * @author Ansh Pachauri and Daniil Gofman
19  *
20  */
21 public class Program2 extends ProgramSecondary {
22
23     /*
24      * Private members
   --------------------------------------------------------
25      */
26
27     /**
28      * The program name.
29      */
30     private String name;
31
32     /**
33      * The program context.
34      */
35     private Map<String, Statement> context;
```

```
36
37      /**
38       * The program body.
39       */
40      private Statement body;
41
42      /**
43       * Reports whether all the names of instructions in {@code
    c} are valid
44       * IDENTIFIERs.
45       *
46       * @param c
47       *            the context to check
48       * @return true if all instruction names are identifiers;
    false otherwise
49       * @ensures <pre>
50       * allIdentifiers =
51       *   [all the names of instructions in c are valid
    IDENTIFIERs]
52       * </pre>
53       */
54      private static boolean allIdentifiers(Map<String,
    Statement> c) {
55          for (Map.Pair<String, Statement> pair : c) {
56              if (!Tokenizer.isIdentifier(pair.key())) {
57                  return false;
58              }
59          }
60          return true;
61      }
62
63      /**
64       * Reports whether no instruction name in {@code c} is the
    name of a
65       * primitive instruction.
66       *
67       * @param c
68       *            the context to check
69       * @return true if no instruction name is the name of a
```

```
   primitive
 70     *         instruction; false otherwise
 71     * @ensures <pre>
 72     * noPrimitiveInstructions =
 73     *   [no instruction name in c is the name of a primitive
   instruction]
 74     * </pre>
 75     */
 76    private static boolean noPrimitiveInstructions(Map<String,
   Statement> c) {
 77        return !c.hasKey("move") && !c.hasKey("turnleft")
 78                && !c.hasKey("turnright") && !
   c.hasKey("infect")
 79                && !c.hasKey("skip");
 80    }
 81
 82    /**
 83     * Reports whether all the bodies of instructions in {@code
   c} are BLOCK
 84     * statements.
 85     *
 86     * @param c
 87     *            the context to check
 88     * @return true if all instruction bodies are BLOCK
   statements; false
 89     *         otherwise
 90     * @ensures <pre>
 91     * allBlocks =
 92     *   [all the bodies of instructions in c are BLOCK
   statements]
 93     * </pre>
 94     */
 95    private static boolean allBlocks(Map<String, Statement> c)
   {
 96        for (Map.Pair<String, Statement> pair : c) {
 97            if (pair.value().kind() != Kind.BLOCK) {
 98                return false;
 99            }
100        }
```

```
101         return true;
102     }
103 
104     /**
105      * Creator of initial representation.
106      */
107     private void createNewRep() {
108 
109         this.name = "Unnamed";
110         this.context = new Map1L<String, Statement>();
111         this.body = new Statement1();
112 
113     }
114 
115     /*
116      * Constructors
    ------------------------------------------------------------
117      */
118 
119     /**
120      * No-argument constructor.
121      */
122     public Program2() {
123         this.createNewRep();
124     }
125 
126     /*
127      * Standard methods
    -----------------------------------------------------------
128      */
129 
130     @Override
131     public final Program newInstance() {
132         try {
133             return
    this.getClass().getConstructor().newInstance();
134         } catch (ReflectiveOperationException e) {
135             throw new AssertionError(
136                     "Cannot construct object of type " +
```

```
    this.getClass());
137         }
138     }
139
140     @Override
141     public final void clear() {
142         this.createNewRep();
143     }
144
145     @Override
146     public final void transferFrom(Program source) {
147         assert source != null : "Violation of: source is not
    null";
148         assert source != this : "Violation of: source is not
    this";
149         assert source instanceof Program2 : ""
150                 + "Violation of: source is of dynamic type
    Program2";
151         /*
152          * This cast cannot fail since the assert above would
    have stopped
153          * execution in that case: source must be of dynamic
    type Program2.
154          */
155         Program2 localSource = (Program2) source;
156         this.name = localSource.name;
157         this.context = localSource.context;
158         this.body = localSource.body;
159         localSource.createNewRep();
160     }
161
162     /*
163      * Kernel methods
    ------------------------------------------------------------
164      */
165
166     @Override
167     public final void setName(String n) {
168         assert n != null : "Violation of: n is not null";
```

```
169         assert Tokenizer.isIdentifier(n) : ""
170                 + "Violation of: n is a valid IDENTIFIER";
171         // Set name of this to a n.
172         this.name = n;
173 
174     }
175 
176     @Override
177     public final String name() {
178 
179         // Return the name of program.
180         return this.name;
181     }
182 
183     @Override
184     public final Map<String, Statement> newContext() {
185 
186         // Return the variable of the same type of context.
187         return this.context.newInstance();
188     }
189 
190     @Override
191     public final void swapContext(Map<String, Statement> c) {
192         assert c != null : "Violation of: c is not null";
193         assert c instanceof Map1L<?, ?> : "Violation of: c is a
    Map1L<?, ?>";
194         assert allIdentifiers(
195                 c) : "Violation of: names in c are valid
    IDENTIFIERs";
196         assert noPrimitiveInstructions(
197                 c) : "Violation of: names in c do not match the
    names"
198                         + " of primitive instructions in the BL
    language";
199         assert allBlocks(c) : "Violation of: bodies in c"
200                 + " are all BLOCK statements";
201 
202         // Create a temporary Map to store the current context
203         Map<String, Statement> temp =
```

```
    this.context.newInstance();
204
205         // Transfer the contents of this.context to the temp
    Map
206         temp.transferFrom(this.context);
207
208         // Transfer the contents of Map c to this.context,
209         // effectively swapping their contents
210         this.context.transferFrom(c);
211
212         // Transfer the contents of the temporary temp Map back
    to Map c
213         c.transferFrom(temp);
214     }
215
216     @Override
217     public final Statement newBody() {
218
219         // Returns new body for the Statement
220         return this.body.newInstance();
221     }
222
223     @Override
224     public final void swapBody(Statement b) {
225         assert b != null : "Violation of: b is not null";
226         assert b instanceof Statement1 : "Violation of: b is a
    Statement1";
227         assert b.kind() == Kind.BLOCK : "Violation of: b is a
    BLOCK statement";
228
229         // Create a temporary Statement to store the current
    body
230         Statement temp = this.body.newInstance();
231
232         // Transfer the contents of this.body to the temp
    Statement
233         temp.transferFrom(this.body);
234
235         // Transfer the contents of Statement b to this.body,
```

```
236         // effectively swapping their contents
237         this.body.transferFrom(b);
238 
239         // Transfer the contents of the temporary temp
    Statement back to Statement b
240         b.transferFrom(temp);
241     }
242 
243 }
244 
```

```
 1 import components.program.Program;
 2 import components.program.Program1;
 3 import components.simplereader.SimpleReader;
 4 import components.simplereader.SimpleReader1L;
 5 import components.simplewriter.SimpleWriter;
 6 import components.simplewriter.SimpleWriter1L;
 7
 8 /**
 9  * BL program parser and pretty-printer to test Program2 kernel
   student
10  * implementation against Program1 kernel library
   implementation.
11  *
12  * @author Ansh Pachauri and Daniil Gofman
13  *
14  */
15 public final class ProgramTester {
16
17     /**
18      * Private constructor so this utility class cannot be
   instantiated.
19      */
20     private ProgramTester() {
21     }
22
23     /**
24      * Main method.
25      *
26      * @param args
27      *            the command line arguments
28      */
29     public static void main(String[] args) {
30         SimpleReader in = new SimpleReader1L();
31         SimpleWriter out = new SimpleWriter1L();
32         /*
33          * Get file name
34          */
35         out.print("Enter a file name for a valid BL program: ");
36         String fileName = in.nextLine();
```

```
37         /*
38          * Input program using the library implementation
   Program1
39          */
40         out.print(
41                 "  Reading program using the library
   implementation Program1...");
42         SimpleReader file = new SimpleReader1L(fileName);
43         Program p1 = new Program1();
44         p1.parse(file);
45         file.close();
46         out.println("done!");
47         /*
48          * Input program again using the student implementation
   Program2
49          */
50         out.print(
51                 "  Reading program using the student
   implementation Program2...");
52         file = new SimpleReader1L(fileName);
53         Program p2 = new Program2();
54         p2.parse(file);
55         file.close();
56         out.println("done!");
57         /*
58          * Check for equality
59          */
60         out.print("  Checking for equality of two programs...");
61         if (p2.equals(p1)) {
62             out.println("they are equal, good!");
63         } else {
64             out.println("error: programs are not equal!");
65         }
66         /*
67          * Output program with library implementation
68          */
69         out.println("  Pretty printing program with library
   implementation...");
70         p1.prettyPrint(out);
```

```
71          out.println("done!");
72          /*
73           * Output program with student implementation
74           */
75          out.println("  Pretty printing program with student
  implementation...");
76          p2.prettyPrint(out);
77          out.println("done!");
78          /*
79           * Check for equality again
80           */
81          out.print("  Checking for equality of two programs...");
82          if (p2.equals(p1)) {
83              out.println("they are equal, good!");
84          } else {
85              out.println("error: programs are not equal!");
86          }
87          in.close();
88          out.close();
89      }
90
91  }
92
```

```
 1 import components.sequence.Sequence;
 2 import components.statement.Statement;
 3 import components.statement.StatementSecondary;
 4 import components.tree.Tree;
 5 import components.tree.Tree1;
 6 import components.utilities.Tokenizer;
 7
 8 /**
 9  * {@code Statement} represented as a {@code
   Tree<StatementLabel>} with
10  * implementations of primary methods.
11  *
12  * @convention [$this.rep is a valid representation of a
   Statement]
13  * @correspondence this = $this.rep
14  *
15  * @author Ansh Pachauri and Daniil Gofman
16  *
17  */
18 public class Statement2 extends StatementSecondary {
19
20     /*
21      * Private members
   ------------------------------------------------------------
22      */
23
24     /**
25      * Label class for the tree representation.
26      */
27     private static final class StatementLabel {
28
29         /**
30          * Statement kind.
31          */
32         private Kind kind;
33
34         /**
35          * IF/IF_ELSE/WHILE statement condition.
36          */
```

```
37          private Condition condition;
38
39          /**
40           * CALL instruction name.
41           */
42          private String instruction;
43
44          /**
45           * Constructor for BLOCK.
46           *
47           * @param k
48           *            the kind of statement
49           *
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<StatementLabel>();
121         StatementLabel start = new StatementLabel(Kind.BLOCK);
122         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
123         this.rep.assemble(start, 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 kind of a root
180         return this.rep.root().kind;
181     }
182
183     @Override
184     public final void addToBlock(int pos, Statement s) {
185         assert s != null : "Violation of: s is not null";
186         assert s != this : "Violation of: s is not this";
187         assert s instanceof Statement2 : "Violation of: s is a
    Statement2";
188         assert this
189                 .kind() == Kind.BLOCK : "Violation of: [this is
    a BLOCK statement]";
190         assert 0 <= pos : "Violation of: 0 <= pos";
191         assert pos <= this
192                 .lengthOfBlock() : "Violation of: pos <=
    [length of this BLOCK]";
193         assert s.kind() != Kind.BLOCK : "Violation of: [s is
    not a BLOCK statement]";
194
195         // Cast the input Statement 's' to Statement2 (specific
    type) as sLocal
196         Statement2 sLocal = (Statement2) s;
197
198         // Create a sequence of tree nodes to represent
199         // the children of 'this' BLOCK statement
200         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
201
202         // Disassemble 'this' BLOCK statement into its children
    and get its label
203         StatementLabel label = this.rep.disassemble(children);
204
205         // Add the Statement 'sLocal' to the specified position
206         // pos in the children sequence
207         children.add(pos, sLocal.rep);
208
```

```
209         // Assemble this BLOCK statement with the updated
    children
210         // sequence and the original label
211         this.rep.assemble(label, children);
212
213         // Clear the content of Statement s to prevent any
    further use or confusion
214         s.clear();
215     }
216
217     @Override
218     public final Statement removeFromBlock(int pos) {
219         assert 0 <= pos : "Violation of: 0 <= pos";
220         assert pos < this
221                 .lengthOfBlock() : "Violation of: pos < [length
    of this BLOCK]";
222         assert this
223                 .kind() == Kind.BLOCK : "Violation of: [this is
    a BLOCK statement]";
224
225         // Create a new instance of Statement2 to represent the
    removed statement
226         Statement2 s = this.newInstance();
227
228         // Create a sequence of tree nodes to represent
229         // the children of this BLOCK statement
230         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
231
232         // Disassemble this BLOCK statement into its children
    and get its label
233         StatementLabel label = this.rep.disassemble(children);
234
235         // Remove the statement at the specified position pos
    from the children sequence
236         s.rep = children.remove(pos);
237
238         // Assemble this BLOCK statement with the updated
239         // children sequence and the original label
```

```
240         this.rep.assemble(label, children);
241
242         // Return the removed statement
243         return s;
244     }
245
246     @Override
247     public final int lengthOfBlock() {
248         assert this.kind() == Kind.BLOCK : ""
249                 + "Violation of: [this is a BLOCK statement]";
250
251         // Returns length of a block
252         return this.rep.numberOfSubtrees();
253     }
254
255     @Override
256     public final void assembleIf(Condition c, Statement s) {
257         assert c != null : "Violation of: c is not null";
258         assert s != null : "Violation of: s is not null";
259         assert s != this : "Violation of: s is not this";
260         assert s instanceof Statement2 : "Violation of: s is a
    Statement2";
261         assert s.kind() == Kind.BLOCK : ""
262                 + "Violation of: [s is a BLOCK statement]";
263         Statement2 sLocal = (Statement2) s;
264         StatementLabel label = new StatementLabel(Kind.IF, c);
265         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
266         children.add(0, sLocal.rep);
267         this.rep.assemble(label, children);
268         sLocal.createNewRep();
269     }
270
271     @Override
272     public final Condition disassembleIf(Statement s) {
273         assert s != null : "Violation of: s is not null";
274         assert s != this : "Violation of: s is not this";
275         assert s instanceof Statement2 : "Violation of: s is a
    Statement2";
```

```
276         assert this.kind() == Kind.IF : ""
277                 + "Violation of: [this is an IF statement]";
278         Statement2 locals = (Statement2) s;
279         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
280         StatementLabel label = this.rep.disassemble(children);
281         locals.rep = children.remove(0);
282         this.createNewRep(); // clears this
283         return label.condition;
284     }
285
286     @Override
287     public final void assembleIfElse(Condition c, Statement s1,
    Statement s2) {
288         assert c != null : "Violation of: c is not null";
289         assert s1 != null : "Violation of: s1 is not null";
290         assert s2 != null : "Violation of: s2 is not null";
291         assert s1 != this : "Violation of: s1 is not this";
292         assert s2 != this : "Violation of: s2 is not this";
293         assert s1 != s2 : "Violation of: s1 is not s2";
294         assert s1 instanceof Statement2 : "Violation of: s1 is
    a Statement2";
295         assert s2 instanceof Statement2 : "Violation of: s2 is
    a Statement2";
296         assert s1
297                 .kind() == Kind.BLOCK : "Violation of: [s1 is a
    BLOCK statement]";
298         assert s2
299                 .kind() == Kind.BLOCK : "Violation of: [s2 is a
    BLOCK statement]";
300
301         // Cast Statement s1 and s2 to Statement2 as s1Local
    and s2Local
302         Statement2 s1Local = (Statement2) s1;
303         Statement2 s2Local = (Statement2) s2;
304
305         // Create a new StatementLabel representing
306         // the IF_ELSE kind with the provided Condition c
307         StatementLabel label = new StatementLabel(Kind.IF_ELSE,
```

```
    c);
308
309         // Create a sequence of tree nodes to represent the
    children of this statement
310         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
311
312         // Add s2Local to the children sequence at position 0
313         children.add(0, s2Local.rep);
314
315         // Add s1Local to the children sequence at position 0,
    pushing s2Local down
316         children.add(0, s1Local.rep);
317
318         // Assemble this statement with the new label and the
    updated children sequence
319         this.rep.assemble(label, children);
320
321         // Create a new representation for s1Local and s2Local
322         s1Local.createNewRep();
323         s2Local.createNewRep();
324     }
325
326     @Override
327     public final Condition disassembleIfElse(Statement s1,
    Statement s2) {
328         assert s1 != null : "Violation of: s1 is not null";
329         assert s2 != null : "Violation of: s1 is not null";
330         assert s1 != this : "Violation of: s1 is not this";
331         assert s2 != this : "Violation of: s2 is not this";
332         assert s1 != s2 : "Violation of: s1 is not s2";
333         assert s1 instanceof Statement2 : "Violation of: s1 is
    a Statement2";
334         assert s2 instanceof Statement2 : "Violation of: s2 is
    a Statement2";
335         assert this
336                 .kind() == Kind.IF_ELSE : "Violation of: [this
    is an IF_ELSE statement]";
337
```

```
338         // Cast Statement s1 and s2 to Statement2 as localIf
    and localElse
339         Statement2 localIf = (Statement2) s1;
340         Statement2 localElse = (Statement2) s2;
341
342         // Create a sequence of tree nodes to represent the
    children of this statement
343         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
344
345         // Disassemble 'this' IF_ELSE statement into its
    children and get its root label
346         StatementLabel root = this.rep.disassemble(children);
347
348         // Assign the first child (if-branch) to localIf.rep
349         localIf.rep = children.remove(0);
350
351         // Assign the second child (else-branch) to
    localElse.rep
352         localElse.rep = children.remove(0);
353
354         // Assemble IF_ELSE statement with the updated root
    label and children sequence
355         this.rep.assemble(root, children);
356
357         // Create a new representation for this IF_ELSE
    statement
358         this.createNewRep();
359
360         // Return the Condition associated with the root label
361         return root.condition;
362     }
363
364     @Override
365     public final void assembleWhile(Condition c, Statement s) {
366         assert c != null : "Violation of: c is not null";
367         assert s != null : "Violation of: s is not null";
368         assert s != this : "Violation of: s is not this";
369         assert s instanceof Statement2 : "Violation of: s is a
```

```
    Statement2";
370         assert s.kind() == Kind.BLOCK : "Violation of: [s is a
    BLOCK statement]";
371
372         // Cast Statement s to Statement2 as sLocal
373         Statement2 sLocal = (Statement2) s;
374
375         // Create a new StatementLabel representing
376         // the WHILE kind with the provided Condition c
377         StatementLabel label = new StatementLabel(Kind.WHILE,
    c);
378
379         // Create a sequence of tree nodes to represent the
    children of this statement
380         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
381
382         // Add sLocal to the children sequence at position 0
383         children.add(0, sLocal.rep);
384
385         // Assemble this statement with the new label and the
    updated children sequence
386         this.rep.assemble(label, children);
387
388         // Create a new representation for sLocal
389         sLocal.createNewRep();
390     }
391
392     @Override
393     public final Condition disassembleWhile(Statement s) {
394         assert s != null : "Violation of: s is not null";
395         assert s != this : "Violation of: s is not this";
396         assert s instanceof Statement2 : "Violation of: s is a
    Statement2";
397         assert this
398                 .kind() == Kind.WHILE : "Violation of: [this is
    a WHILE statement]";
399
400         // Cast Statement s to Statement2 as sLocal
```

```
401         Statement2 sLocal = (Statement2) s;
402
403         // Create a sequence of tree nodes to represent the
    children of this statement
404         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
405
406         // Disassemble this WHILE statement into its children
    and get its label
407         StatementLabel label = this.rep.disassemble(children);
408
409         // Assign the child (the BLOCK statement) to sLocal.rep
410         sLocal.rep = children.remove(0);
411
412         // Create a new representation for this WHILE statement
413         this.createNewRep();
414
415         // Return the Condition associated with the label
416         return label.condition;
417     }
418
419     @Override
420     public final void assembleCall(String inst) {
421         assert inst != null : "Violation of: inst is not null";
422         assert Tokenizer.isIdentifier(
423                 inst) : "Violation of: inst is a valid
    IDENTIFIER";
424
425         // Create a new StatementLabel representing
426         // the CALL kind with the provided instruction inst
427         StatementLabel label = new StatementLabel(Kind.CALL,
    inst);
428
429         // Create a sequence of tree nodes to represent the
    children of this statement
430         Sequence<Tree<StatementLabel>> children =
    this.rep.newSequenceOfTree();
431
432         // Assemble this statement with the new label and an
```

```
   empty children sequence
433         this.rep.assemble(label, children);
434     }
435
436     @Override
437     public final String disassembleCall() {
438         assert this
439                 .kind() == Kind.CALL : "Violation of: [this is
   a CALL statement]";
440
441         // Create a sequence of tree nodes to represent the
   children of this statement
442         Sequence<Tree<StatementLabel>> children =
   this.rep.newSequenceOfTree();
443
444         // Disassemble this CALL statement into its children
   and get its label
445         StatementLabel label = this.rep.disassemble(children);
446
447         // Create a new representation for this CALL statement
448         this.createNewRep();
449
450         // Return the instruction associated with the label
451         return label.instruction;
452     }
453
454 }
455
```

```
 1 import components.queue.Queue;
 2 import components.simplereader.SimpleReader;
 3 import components.simplereader.SimpleReader1L;
 4 import components.simplewriter.SimpleWriter;
 5 import components.simplewriter.SimpleWriter1L;
 6 import components.statement.Statement;
 7 import components.statement.Statement1;
 8 import components.utilities.Tokenizer;
 9
10 /**
11  * BL statement parser and pretty-printer to test Statement2
   kernel student
12  * implementation against Statement1 kernel library
   implementation.
13  *
14  * @author Ansh Pachauri and Daniil Gofman
15  *
16  */
17 public final class StatementTester {
18
19     /**
20      * Private constructor so this utility class cannot be
   instantiated.
21      */
22     private StatementTester() {
23     }
24
25     /**
26      * Main method.
27      *
28      * @param args
29      *            the command line arguments
30      */
31     public static void main(String[] args) {
32         SimpleReader in = new SimpleReader1L();
33         SimpleWriter out = new SimpleWriter1L();
34         /*
35          * Get file name
36          */
```

```
37         out.print("Enter a file name for a valid BL statement or
   "
38                 + "sequence of statements: ");
39         String fileName = in.nextLine();
40         /*
41          * Input statement(s) using the library implementation
   Statement1
42          */
43         out.print("  Reading statement(s) using the library "
44                 + "implementation Statement1...");
45         SimpleReader file = new SimpleReader1L(fileName);
46         Statement s1 = new Statement1();
47         Queue<String> tokens = Tokenizer.tokens(file);
48         s1.parseBlock(tokens);
49         file.close();
50         out.println("done!");
51         /*
52          * Input statement(s) again using the student
   implementation Statement2
53          */
54         out.print("  Reading statement(s) using the student "
55                 + "implementation Statement2...");
56         file = new SimpleReader1L(fileName);
57         Statement s2 = new Statement2();
58         tokens = Tokenizer.tokens(file);
59         s2.parseBlock(tokens);
60         file.close();
61         out.println("done!");
62         /*
63          * Check for equality
64          */
65         out.print("  Checking for equality of two
   statements...");
66         if (s2.equals(s1)) {
67             out.println("they are equal, good!");
68         } else {
69             out.println("error: statements are not equal!");
70         }
71         /*
```

```
72          * Output statement(s) with library implementation
73          */
74         out.println(
75                 "  Pretty printing statement with library
  implementation...");
76         s1.prettyPrint(out, 0);
77         out.println("done!");
78         /*
79          * Output statement(s) with student implementation
80          */
81         out.println(
82                 "  Pretty printing statement with student
  implementation...");
83         s2.prettyPrint(out, 0);
84         out.println("done!");
85         /*
86          * Check for equality again
87          */
88         out.print("  Checking for equality of two
  statements...");
89         if (s2.equals(s1)) {
90             out.println("they are equal, good!");
91         } else {
92             out.println("error: statements are not equal!");
93         }
94         in.close();
95         out.close();
96     }
97
98 }
99
```