

```
1 import components.map.Map;
9
10 /**
11  * {@code Program} represented the obvious way with
12  * implementations of primary
13  * methods.
14  * @convention [$this.name is an IDENTIFIER] and
15  * [$this.context is a CONTEXT]
16  * and [$this.body is a BLOCK statement]
17  * @correspondence this = ($this.name, $this.context,
18  * $this.body)
19  *
20  * @author Zhuoyang Li + Xinci Ma
21  */
22
23 public class Program2 extends ProgramSecondary {
24     /*
25      * Private members
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
44      * {@code c} are valid
45      * IDENTIFIERS.
46      *
47      * @param c
48      * the context to check
49      * @return true if all instruction names are identifiers;
50      * false otherwise
51      */
52     public boolean areNamesValid() {
53         return true;
54     }
55 }
```

```
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.containsKey("move") && !c.containsKey("turnleft")
78             && !c.containsKey("turnright") && !
c.containsKey("infect")
79             && !c.containsKey("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<>();
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
134             this.getClass().getConstructor().newInstance();
135         } catch (ReflectiveOperationException e) {
136             throw new AssertionError(
137                 "Cannot construct object of type " +
138                 this.getClass());
139         }
140     }
141
142     @Override
143     public final void clear() {
144         this.createNewRep();
145     }
146
147     @Override
148     public final void transferFrom(Program source) {
149         assert source != null : "Violation of: source is not
150         null";
151         assert source != this : "Violation of: source is not
152         this";
153         assert source instanceof Program2 : ""
154         + "Violation of: source is of dynamic type
155         Program2";
156         /*
157          * This cast cannot fail since the assert above would
158          * have stopped
159          * execution in that case: source must be of dynamic
160          * type Program2.
161          */
162         Program2 localSource = (Program2) source;
163         this.name = localSource.name;
164         this.context = localSource.context;
165         this.body = localSource.body;
166         localSource.createNewRep();
167     }
168
169     /*
170     * Kernel methods
171     */
172
173     @Override
174     public final void setName(String n) {
175         assert n != null : "Violation of: n is not null";
176     }
```

```
169         assert Tokenizer.isIdentifier(n) : ""
170             + "Violation of: n is a valid IDENTIFIER";
171
172         this.name = n;
173
174     }
175
176     @Override
177     public final String name() {
178
179         return this.name;
180     }
181
182     @Override
183     public final Map<String, Statement> newContext() {
184
185         return this.context.newInstance();
186     }
187
188     @Override
189     public final void swapContext(Map<String, Statement> c) {
190         assert c != null : "Violation of: c is not null";
191         assert c instanceof Map1L<?, ?> : "Violation of: c is
192 a Map1L<?, ?>";
193         assert allIdentifiers(
194             c) : "Violation of: names in c are valid
195 IDENTIFIERS";
196         assert noPrimitiveInstructions(c) : ""
197             + "Violation of: names in c do not match the
198 names"
199             + " of primitive instructions in the BL
200 language";
201         assert allBlocks(c) : "Violation of: bodies in c"
202             + " are all BLOCK statements";
203
204         Map<String, Statement> temp =
205             this.context.newInstance();
206         temp.transferFrom(c);
207         c.transferFrom(this.context);
208         this.context.transferFrom(temp);
209     }
210
211     @Override
212     public final Statement newBody() {
213
214         return this.body.newInstance();
215     }
216 }
```

```
211     }
212
213     @Override
214     public final void swapBody(Statement b) {
215         assert b != null : "Violation of: b is not null";
216         assert b instanceof Statement1 : "Violation of: b is
a Statement1";
217         assert b.kind() == Kind.BLOCK : "Violation of: b is a
BLOCK statement";
218
219         Statement temp = new Statement1();
220         temp.transferFrom(b);
221         b.transferFrom(this.body);
222         this.body.transferFrom(temp);
223
224     }
225
226 }
227
```