

```

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 Qinuo Shi & Yiming Cheng
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.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     // TODO - fill in body
110     this.body = new Statement1();
111     this.context = new Map1L<String, Statement>();
112     this.name = "Unnamed";
113     // Make sure to use Statement1 from the library
114     // Use Map1L for the context if you want the asserts below to match

```

```

115
116     }
117
118     /*
119     * Constructors -----
120     */
121
122     /**
123     * No-argument constructor.
124     */
125     public Program2() {
126         this.createNewRep();
127     }
128
129     /*
130     * Standard methods -----
131     */
132
133     @Override
134     public final Program newInstance() {
135         try {
136             return this.getClass().getConstructor().newInstance();
137         } catch (ReflectiveOperationException e) {
138             throw new AssertionError(
139                 "Cannot construct object of type " + this.getClass());
140         }
141     }
142
143     @Override
144     public final void clear() {
145         this.createNewRep();
146     }
147
148     @Override
149     public final void transferFrom(Program source) {
150         assert source != null : "Violation of: source is not null";
151         assert source != this : "Violation of: source is not this";
152         assert source instanceof Program2 : ""
153             + "Violation of: source is of dynamic type Program2";
154         /*
155         * This cast cannot fail since the assert above would have stopped
156         * execution in that case: source must be of dynamic type Program2.
157         */
158         Program2 localSource = (Program2) source;
159         this.name = localSource.name;
160         this.context = localSource.context;
161         this.body = localSource.body;
162         localSource.createNewRep();
163     }
164
165     /*
166     * Kernel methods -----
167     */
168
169     @Override
170     public final void setName(String n) {
171         assert n != null : "Violation of: n is not null";

```

```
172         assert Tokenizer.isIdentifier(n) : ""
173             + "Violation of: n is a valid IDENTIFIER";
174
175         // TODO - fill in body
176         this.name = n;
177
178     }
179
180     @Override
181     public final String name() {
182
183         // TODO - fill in body
184
185         // Fix this line to return the result.
186         return this.name;
187     }
188
189     @Override
190     public final Map<String, Statement> newContext() {
191
192         // TODO - fill in body
193
194         // Fix this line to return the result.
195         return this.context.newInstance();
196     }
197
198     @Override
199     public final void swapContext(Map<String, Statement> c) {
200         assert c != null : "Violation of: c is not null";
201         assert c instanceof Map1L<?, ?> : "Violation of: c is a Map1L<?, ?>";
202         assert allIdentifiers(
203             c) : "Violation of: names in c are valid IDENTIFIERS";
204         assert noPrimitiveInstructions(c) : ""
205             + "Violation of: names in c do not match the names"
206             + " of primitive instructions in the BL language";
207         assert allBlocks(c) : "Violation of: bodies in c"
208             + " are all BLOCK statements";
209
210         // TODO - fill in body
211         Map<String, Statement> tool = this.context.newInstance();
212         tool.transferFrom(c);
213
214         c.transferFrom(this.context);
215         this.context.transferFrom(tool);
216
217     }
218
219     @Override
220     public final Statement newBody() {
221
222         // TODO - fill in body
223
224         // Fix this line to return the result.
225         return this.body.newInstance();
226     }
227
228     @Override
```

```
229     public final void swapBody(Statement b) {
230         assert b != null : "Violation of: b is not null";
231         assert b instanceof Statement1 : "Violation of: b is a Statement1";
232         assert b.kind() == Kind.BLOCK : "Violation of: b is a BLOCK statement";
233
234         // TODO - fill in body
235         Statement tool = this.body.newInstance();
236         tool.transferFrom(b);
237
238         b.transferFrom(this.body);
239         this.body.transferFrom(tool);
240
241     }
242
243 }
244
```