```java
import components.map.Map;

/**
 * Layered implementation of secondary method {@code parse} for {@code Program}.
 *
 * @author Chloe Feller and Krish Patel
 *
 */
public final class Program1Parse1 extends Program1 {

    /*
     * Private members --------------------------------------------------------
     */

    /**
     * Parses a single BL instruction from {@code tokens} returning the
     * instruction name as the value of the function and the body of the
     * instruction in {@code body}.
     *
     * @param tokens
     *            the input tokens
     * @param body
     *            the instruction body
     * @return the instruction name
     * @replaces body
     * @updates tokens
     * @requires <pre>
     * [<"INSTRUCTION"> is a prefix of tokens]  and
     *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
     * </pre>
     * @ensures <pre>
     * if [an instruction string is a proper prefix of #tokens]  and
     *    [the beginning name of this instruction equals its ending name]  and
     *    [the name of this instruction does not equal the name of a primitive
     *     instruction in the BL language] then
     *  parseInstruction = [name of instruction at start of #tokens]  and
     *  body = [Statement corresponding to the block string that is the body of
     *          the instruction string at start of #tokens]  and
     *  #tokens = [instruction string at start of #tokens] * tokens
     * else
     *  [report an appropriate error message to the console and terminate client]
     * </pre>
     */
    private static String parseInstruction(Queue<String> tokens,
            Statement body) {
        assert tokens != null : "Violation of: tokens is not null";
        assert body != null : "Violation of: body is not null";
        assert tokens.length() > 0 && tokens.front().equals("INSTRUCTION") : ""
                + "Violation of: <\"INSTRUCTION\"> is proper prefix of tokens";

        /**
         * Check the first three tokens are correct.
         */
        tokens.dequeue(); // asserted in contract this equals "INSTRUCTION"
        String firstInstruction = tokens.dequeue();

        /**
```

```
68          * Assert instruction name is unique and not a primitive call.
69          */
70         Reporter.assertElseFatalError(Tokenizer.isIdentifier(firstInstruction),
71                 "Must be a unique name");
72
73         String[] primitive = { "move", "turnleft", "turnright", "infect",
74                 "skip" };
75         for (String p : primitive) {
76             Reporter.assertElseFatalError(!p.equals(firstInstruction),
77                     "Instruction cannot be a primtive call");
78         }
79
80         Reporter.assertElseFatalError(tokens.dequeue().equals("IS"),
81                 "Invalid token");
82
83         /**
84          * Parse block.
85          */
86         body.parseBlock(tokens);
87
88         /**
89          * Check the final two tokens are correct.
90          */
91         Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
92                 "Invalid token");
93         String secondInstruction = tokens.dequeue();
94
95         /**
96          * Check firstInstruction and secondInstruction are equal.
97          */
98         Reporter.assertElseFatalError(
99                 firstInstruction.equals(secondInstruction),
100                "Instruction names are not equal");
101
102        // This line added just to make the program compilable.
103        return secondInstruction;
104    }
105
106    /*
107     * Constructors ------------------------------------------------------------
108     */
109
110    /**
111     * No-argument constructor.
112     */
113    public Program1Parse1() {
114        super();
115    }
116
117    /*
118     * Public methods ----------------------------------------------------------
119     */
120
121    @Override
122    public void parse(SimpleReader in) {
123        assert in != null : "Violation of: in is not null";
124        assert in.isOpen() : "Violation of: in.is_open";
```

```
125         Queue<String> tokens = Tokenizer.tokens(in);
126         this.parse(tokens);
127     }
128 
129     @Override
130     public void parse(Queue<String> tokens) {
131         assert tokens != null : "Violation of: tokens is not null";
132         assert tokens.length() > 0 : ""
133                 + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
134 
135         // TODO - fill in body
136         String str = tokens.dequeue();
137         Reporter.assertElseFatalError(str.equals("PROGRAM"),
138                 "Must be a unique name");
139 
140         Reporter.assertElseFatalError(tokens.length() > 0,
141                 "Token is not included.");
142 
143         String title = tokens.dequeue();
144         Reporter.assertElseFatalError(Tokenizer.isIdentifier(title),
145                 "Invalid token");
146         Reporter.assertElseFatalError(tokens.length() > 0,
147                 "Token is not included");
148 
149         String i = tokens.dequeue();
150         Reporter.assertElseFatalError(i.equals("IS"), "Invalid token");
151 
152         Reporter.assertElseFatalError(tokens.length() > 0,
153                 "Token is not included");
154 
155         Map<String, Statement> context = this.newContext();
156 
157         while (tokens.front().equals("INSTRUCTION")) {
158             Statement block = this.newBody();
159 
160             String name = parseInstruction(tokens, block);
161 
162             Reporter.assertElseFatalError(Tokenizer.isIdentifier(name),
163                     "Expecting" + name);
164             Reporter.assertElseFatalError(!context.hasKey(name),
165                     "Cannot have more than one Instruction");
166 
167             context.add(name, block);
168             Reporter.assertElseFatalError(tokens.length() > 0,
169                     "Program did not end correctly");
170         }
171 
172         String begin = tokens.dequeue();
173         Reporter.assertElseFatalError(begin.equals("BEGIN"), "Invalid token");
174 
175         Statement body = this.newBody();
176         body.parseBlock(tokens);
177 
178         Reporter.assertElseFatalError(tokens.length() > 0,
179                 "Token is not included");
180 
181         String end = tokens.dequeue();
```

```
182         Reporter.assertElseFatalError(end.equals("END"), "Invalid token");
183
184         Reporter.assertElseFatalError(tokens.length() > 0,
185                 "Token is not included");
186
187         String name = tokens.dequeue();
188         Reporter.assertElseFatalError(name.equals(title), "Invalid token");
189
190         Reporter.assertElseFatalError(
191                 tokens.length() == 1
192                         && tokens.front().equals(Tokenizer.END_OF_INPUT),
193                 "Token is not included");
194
195         this.setName(name);
196         this.swapContext(context);
197         this.swapBody(body);
198
199     }
200
201     /*
202      * Main test method ----------------------------------------------------
203      */
204
205     /**
206      * Main method.
207      *
208      * @param args
209      *            the command line arguments
210      */
211     public static void main(String[] args) {
212         SimpleReader in = new SimpleReader1L();
213         SimpleWriter out = new SimpleWriter1L();
214         /*
215          * Get input file name
216          */
217         out.print("Enter valid BL program file name: ");
218         String fileName = in.nextLine();
219         /*
220          * Parse input file
221          */
222         out.println("*** Parsing input file ***");
223         Program p = new Program1Parse1();
224         SimpleReader file = new SimpleReader1L(fileName);
225         Queue<String> tokens = Tokenizer.tokens(file);
226         file.close();
227         p.parse(tokens);
228         /*
229          * Pretty print the program
230          */
231         out.println("*** Pretty print of parsed program ***");
232         p.prettyPrint(out);
233
234         in.close();
235         out.close();
236     }
237
238 }
```

239