```java
import components.map.Map;

/**
 * Layered implementation of secondary method {@code parse} for {@code Program}.
 *
 * @author Elizabeth Tisdale, Harker LeCroy
 *
 */
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";

        // remove "INSTRUCTION"
        tokens.dequeue();

        String identifier = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isIdentifier(identifier),
                "No identifier.");

        // check for is
        Reporter.assertElseFatalError(tokens.dequeue().equals("IS"),
                "IS token not found.");

        // parse block
        body.parseBlock(tokens);

        // check for end
        Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
                "END token not found.");

        // check for identifier
        Reporter.assertElseFatalError(tokens.dequeue().equals(identifier),
                "No matching closing identifier.");

        // This line added just to make the program compilable.
```

```
 87         return identifier;
 88     }
 89
 90     /*
 91      * Constructors ------------------------------------------------------------
 92      */
 93
 94     /**
 95      * No-argument constructor.
 96      */
 97     public Program1Parse1() {
 98         super();
 99     }
100
101     /*
102      * Public methods ----------------------------------------------------------
103      */
104
105     @Override
106     public void parse(SimpleReader in) {
107         assert in != null : "Violation of: in is not null";
108         assert in.isOpen() : "Violation of: in.is_open";
109         Queue<String> tokens = Tokenizer.tokens(in);
110         this.parse(tokens);
111     }
112
113     @Override
114     public void parse(Queue<String> tokens) {
115         assert tokens != null : "Violation of: tokens is not null";
116         assert tokens.length() > 0
117                 : "" + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
118
119         // dequeue program word
120         Reporter.assertElseFatalError(tokens.dequeue().equals("PROGRAM"),
121                 "PROGRAM token not found.");
122
123         // dequeue identifier, keep
124         String identifier = tokens.dequeue();
125         Reporter.assertElseFatalError(Tokenizer.isIdentifier(identifier),
126                 "No identifier.");
127         this.setName(identifier);
128
129         // check for is
130         Reporter.assertElseFatalError(tokens.dequeue().equals("IS"),
131                 "IS token not found.");
132
133         Set<String> instrNames = new Set1L<>();
134
135         Set<String> primInstr = new Set1L<>();
136         primInstr.add("move");
137         primInstr.add("turnleft");
138         primInstr.add("turnright");
139         primInstr.add("infect");
140         primInstr.add("skip");
141         primInstr.add("halt");
142
143         Map<String, Statement> context = new Map1L<>();
144         while (tokens.front().equals("INSTRUCTION")) {
145
146             Statement instructionBody = this.newBody();
147
148             String instrName = parseInstruction(tokens, instructionBody);
149             Reporter.assertElseFatalError(!instrNames.contains(instrName),
150                     "Repeated instruction name.");
151             Reporter.assertElseFatalError(!primInstr.contains(instrName),
152                     "Use of primitive instruction name.");
153
154             context.add(instrName, instructionBody);
155             instrNames.add(instrName);
156         }
157
158         this.swapContext(context);
```

```
159
160         // check for begin
161         Reporter.assertElseFatalError(tokens.dequeue().equals("BEGIN"),
162                 "BEGIN token not found.");
163
164         Statement body = new Statement1();
165
166         body.parseBlock(tokens);
167         this.swapBody(body);
168
169         // check for end
170         Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
171                 "END token not found.");
172
173         // check for identifier
174         Reporter.assertElseFatalError(tokens.dequeue().equals(identifier),
175                 "No matching closing identifier.");
176
177         // check for identifier
178         Reporter.assertElseFatalError(tokens.length() == 1, "Unexpected extra tokens.");
179
180     }
181
182     /*
183      * Main test method ----------------------------------------------------
184      */
185
186     /**
187      * Main method.
188      *
189      * @param args
190      *            the command line arguments
191      */
192     public static void main(String[] args) {
193         SimpleReader in = new SimpleReader1L();
194         SimpleWriter out = new SimpleWriter1L();
195         /*
196          * Get input file name
197          */
198         out.print("Enter valid BL program file name: ");
199         String fileName = in.nextLine();
200         /*
201          * Parse input file
202          */
203         out.println("*** Parsing input file ***");
204         Program p = new Program1Parse1();
205         SimpleReader file = new SimpleReader1L(fileName);
206         Queue<String> tokens = Tokenizer.tokens(file);
207         file.close();
208         p.parse(tokens);
209         /*
210          * Pretty print the program
211          */
212         out.println("*** Pretty print of parsed program ***");
213         p.prettyPrint(out);
214
215         in.close();
216         out.close();
217     }
218
219 }
220
```