```java
import components.map.Map;

/**
 * Layered implementation of secondary method {@code parse} for {@code Program}.
 *
 * @author Zhuoyang Li + Xinci Ma
 *
 */
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
 49      *  #tokens = [instruction string at start of #tokens] *
   tokens
 50      * else
 51      *  [report an appropriate error message to the console
   and terminate client]
 52      * </pre>
 53      */
 54     private static String parseInstruction(Queue<String>
   tokens,
 55             Statement body) {
 56         assert tokens != null : "Violation of: tokens is not
   null";
 57         assert body != null : "Violation of: body is not
   null";
 58         assert tokens.length() > 0 &&
   tokens.front().equals("INSTRUCTION") : ""
 59                 + "Violation of: <\"INSTRUCTION\"> is proper
   prefix of tokens";
 60
 61         //get rid of instruction
 62         String ins = tokens.dequeue();
 63         //get access to identifier
 64         String id = tokens.dequeue();
 65         String pri = "turnleft,turnright,turnback,skip,move";
 66         boolean isPrim = pri.indexOf(id) != -1;
 67         //get rid of is
 68         String is = tokens.dequeue();
 69         //save to body
 70         body.parseBlock(tokens);
 71         //check if the instruction is complete
 72         String end = tokens.dequeue();
 73         String checkEnd = tokens.dequeue();
 74
   Reporter.assertElseFatalError(ins.equals("INSTRUCTION"),
 75                 "Error: Expect String INSTRUCTION");
 76
   Reporter.assertElseFatalError(Tokenizer.isIdentifier(id),
 77                 "Error: Expect a unique instrcution name");
 78         Reporter.assertElseFatalError(!isPrim,
 79                 "Error: you can not put a primitive call as
   an instruction name");
 80         Reporter.assertElseFatalError(is.equals("IS"),
 81                 "Error: Expect String IS");
 82         Reporter.assertElseFatalError(end.equals("END"),
 83                 "Error: Expect String END");
 84         Reporter.assertElseFatalError(checkEnd.equals(id),
```

```
 85                 "Error: Expect a match instrcution name");
 86
 87         return id;
 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         //check header
120         String pro = tokens.dequeue();
121         String pre = tokens.dequeue();
122         this.setName(pre);
123         String is = tokens.dequeue();
124         Reporter.assertElseFatalError(pro.equals("PROGRAM"),
125                 "Error: Expect String PROGRAM");
126         Reporter.assertElseFatalError(is.equals("IS"),
127                 "Error: Expect String IS");
```

```
128         //check header//
129 
130         //Instruction
131         Map<String, Statement> context = this.newContext();
132         while (tokens.front().equals("INSTRUCTION")) {
133             Statement body = this.newBody();
134             String ins = parseInstruction(tokens, body);
135             context.add(ins, body);
136         }
137         this.swapContext(context);
138         //Instruction//
139 
140         //check end
141 
    Reporter.assertElseFatalError(tokens.front().equals("BEGIN"),
142                 "Error: Expect String BEGIN");
143         tokens.dequeue();
144 
145         Statement newB = this.newBody();
146         newB.parseBlock(tokens);
147         this.swapBody(newB);
148 
149         String end = tokens.dequeue();
150         String backid = tokens.dequeue();
151         Reporter.assertElseFatalError(end.equals("END"),
152                 "Error: Expect String END");
153         Reporter.assertElseFatalError(backid.equals(pre),
154                 "Error: Expect A match program name");
155         Reporter.assertElseFatalError(
156                 tokens.front().equals("### END OF INPUT
    ###"), "Error");
157         //check end//
158     }
159 
160     /*
161      * Main test method
    -------------------------------------------------------
162      */
163 
164     /**
165      * Main method.
166      *
167      * @param args
168      *            the command line arguments
169      */
170     public static void main(String[] args) {
171         SimpleReader in = new SimpleReader1L();
```

```
172         SimpleWriter out = new SimpleWriter1L();
173         /*
174          * Get input file name
175          */
176         out.print("Enter valid BL program file name: ");
177         String fileName = in.nextLine();
178         /*
179          * Parse input file
180          */
181         out.println("*** Parsing input file ***");
182         Program p = new Program1Parse1();
183         SimpleReader file = new SimpleReader1L(fileName);
184         Queue<String> tokens = Tokenizer.tokens(file);
185         file.close();
186         p.parse(tokens);
187         /*
188          * Pretty print the program
189          */
190         out.println("*** Pretty print of parsed program
    ***");
191         p.prettyPrint(out);
192
193         in.close();
194         out.close();
195     }
196
197 }
198
```