```java
import components.queue.Queue;

/**
 * Layered implementation of secondary methods {@code parse} and
 * {@code parseBlock} for {@code Statement}.
 *
 * @author Zhuoyang Li + Xinci Ma
 *
 */
public final class Statement1Parse1 extends Statement1 {

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

    /**
     * Converts {@code c} into the corresponding {@code Condition}.
     *
     * @param c
     *            the condition to convert
     * @return the {@code Condition} corresponding to {@code c}
     * @requires [c is a condition string]
     * @ensures parseCondition = [Condition corresponding to c]
     */
    private static Condition parseCondition(String c) {
        assert c != null : "Violation of: c is not null";
        assert Tokenizer
                .isCondition(c) : "Violation of: c is a condition string";
        return Condition.valueOf(c.replace('-', '_').toUpperCase());
    }

    /**
     * Parses an IF or IF_ELSE statement from {@code tokens} into {@code s}.
     *
     * @param tokens
     *            the input tokens
     * @param s
     *            the parsed statement
     * @replaces s
```

```
48      * @updates tokens
49      * @requires <pre>
50      * [<"IF"> is a prefix of tokens]  and
51      *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
52      * </pre>
53      * @ensures <pre>
54      * if [an if string is a proper prefix of #tokens] then
55      *  s = [IF or IF_ELSE Statement corresponding to if
    string at start of #tokens]  and
56      *  #tokens = [if string at start of #tokens] * tokens
57      * else
58      *  [reports an appropriate error message to the console
    and terminates client]
59      * </pre>
60      */
61     private static void parseIf(Queue<String> tokens,
    Statement s) {
62         assert tokens != null : "Violation of: tokens is not
    null";
63         assert s != null : "Violation of: s is not null";
64         assert tokens.length() > 0 && tokens.front().equals(
65                 "IF") : "Violation of: <\"IF\"> is proper
    prefix of tokens";
66
67         // Remove "IF" token from the queue
68         tokens.dequeue();
69         // Ensure the next token is a valid condition
70         Reporter.assertElseFatalError(
71                 tokens.length() > 0 &&
    Tokenizer.isCondition(tokens.front()),
72                 "Expected condition after IF");
73
74         String conditionToken = tokens.dequeue();
75         Condition condition = parseCondition(conditionToken);
76
77         // Ensure "THEN" follows the condition
78         String then = tokens.dequeue();
79         Reporter.assertElseFatalError(
80                 tokens.length() + 1 > 0 &&
    then.equals("THEN"),
81                 "Expected THEN after condition");
82
83         // Parse the block of statements for the IF branch
84         Statement ifBranch = s.newInstance();
85         Reporter.assertElseFatalError(tokens.length() > 0,
86                 "Unexpected end of input in IF branch");
87
```

```
 88          ifBranch.parseBlock(tokens);
 89
 90          // Check for the presence of an ELSE branch or the
    end of the IF statement
 91          Reporter.assertElseFatalError(
 92                  tokens.length() > 0 &&
    (tokens.front().equals("ELSE")
 93                          || tokens.front().equals("END")),
 94                  "Expected ELSE or END after IF branch");
 95
 96          if (tokens.front().equals("ELSE")) {
 97              tokens.dequeue();
 98              // Parse the block of statements for the ELSE
    branch
 99              Statement elseBranch = s.newInstance();
100              Reporter.assertElseFatalError(tokens.length() >
    0,
101                      "Unexpected end of input in ELSE
    branch");
102              elseBranch.parseBlock(tokens);
103
104              // Confirm the correct closure of an IF_ELSE
    statement with "END IF"
105              Reporter.assertElseFatalError(
106                      tokens.length() > 1 &&
    tokens.dequeue().equals("END")
107                              && tokens.dequeue().equals("IF"),
108                      "Expected END IF to close IF_ELSE
    statement");
109              s.assembleIfElse(condition, ifBranch,
    elseBranch);
110          } else {
111              // No ELSE branch, just ensure the IF statement
    is properly closed
112              tokens.dequeue();
113              Reporter.assertElseFatalError(
114                      tokens.length() > 0 &&
    tokens.dequeue().equals("IF"),
115                      "Expected END IF to close IF statement");
116              s.assembleIf(condition, ifBranch);
117          }
118
119      }
120
121      /**
122       * Parses a WHILE statement from the input token queue
    into the provided
```

```
123      * statement object. This involves identifying and
    consuming a "WHILE"
124      * keyword, followed by a condition, and a block of
    statements to execute
125      * while the condition is true. The process concludes
    with expecting and
126      * consuming an "END WHILE" to properly close the loop.
127      *
128      * @param tokens
129      *            The queue of tokens representing the source
    code, which is
130      *            being parsed.
131      * @param s
132      *            The statement object to populate based on
    the WHILE statement
133      *            parsed from the tokens.
134      * @replaces s
135      * @updates tokens
136      * @requires <pre>
137      * ["WHILE" is a prefix of tokens] and
138      * [Tokenizer.END_OF_INPUT is a suffix of tokens]
139      * </pre>
140      * @ensures <pre>
141      * if [a while string is a proper prefix of #tokens] then
142      * s = [WHILE Statement corresponding to the while string
    at the start of #tokens] and
143      * #tokens = [while string at the start of #tokens] *
    tokens
144      * else
145      * [reports an appropriate error message to the console
    and terminates the client]
146      * </pre>
147      */
148     private static void parseWhile(Queue<String> tokens,
    Statement s) {
149         assert tokens != null : "Violation of: tokens is not
    null";
150         assert s != null : "Violation of: s is not null";
151         assert tokens.length() > 0 && tokens.front().equals(
152                 "WHILE") : "Violation of: <\"WHILE\"> is a
    proper prefix of tokens";
153
154         // Remove "WHILE" keyword to proceed with condition
    parsing
155         tokens.dequeue();
156
157         // Check for a valid condition following "WHILE"
```

```
158         Reporter.assertElseFatalError(
159                 tokens.length() > 0 &&
    Tokenizer.isCondition(tokens.front()),
160                 "Expected a condition after WHILE");
161
162         // Extract and parse condition token
163         String conditionToken = tokens.dequeue();
164         Condition condition = parseCondition(conditionToken);
165
166         // Expect "DO" indicating the start of the loop's
    body
167         Reporter.assertElseFatalError(
168                 tokens.length() > 0 &&
    tokens.front().equals("DO"),
169                 "Expected DO after condition");
170
171         tokens.dequeue();
172
173         // Parse the loop body
174         Statement loopBody = s.newInstance();
175         Reporter.assertElseFatalError(tokens.length() > 0,
176                 "Unexpected end of input when parsing WHILE
    block");
177         loopBody.parseBlock(tokens);
178
179         // Ensure loop closure with "END WHILE"
180         Reporter.assertElseFatalError(
181                 tokens.length() > 1 &&
    tokens.dequeue().equals("END")
182                         && tokens.dequeue().equals("WHILE"),
183                 "Expected END WHILE to properly close the
    loop");
184
185         // Assemble the while loop statement
186         s.assembleWhile(condition, loopBody);
187     }
188
189     /**
190      * Parses a CALL statement from {@code tokens} into
    {@code s}.
191      *
192      * @param tokens
193      *            the input tokens
194      * @param s
195      *            the parsed statement
196      * @replaces s
197      * @updates tokens
```

```
198      * @requires [identifier string is a proper prefix of
    tokens]
199      * @ensures <pre>
200      * s =
201      *   [CALL Statement corresponding to identifier string
    at start of #tokens]  and
202      *  #tokens = [identifier string at start of #tokens] *
    tokens
203      * </pre>
204      */
205     private static void parseCall(Queue<String> tokens,
    Statement s) {
206         assert tokens != null : "Violation of: tokens is not
    null";
207         assert s != null : "Violation of: s is not null";
208         assert tokens.length() > 0
209                 && Tokenizer.isIdentifier(tokens.front()) :
    ""
210                         + "Violation of: identifier string is
    proper prefix of tokens";
211
212         String identifier = tokens.dequeue();
213         s.assembleCall(identifier);
214     }
215
216     /*
217      * Constructors
    ------------------------------------------------------------
218      */
219
220     /**
221      * No-argument constructor.
222      */
223     public Statement1Parse1() {
224         super();
225     }
226
227     /*
228      * Public methods
    -------------------------------------------------------------
229      */
230
231     @Override
232     public void parse(Queue<String> tokens) {
233         assert tokens != null : "Violation of: tokens is not
    null";
234         assert tokens.length() > 0 : ""
```

```
235                 + "Violation of: Tokenizer.END_OF_INPUT is a
    suffix of tokens";
236
237         if (tokens.front().equals("IF")) {
238             parseIf(tokens, this);
239         } else if (tokens.front().equals("WHILE")) {
240             parseWhile(tokens, this);
241         } else if (Tokenizer.isIdentifier(tokens.front())) {
242             parseCall(tokens, this);
243         } else {
244             Reporter.fatalErrorToConsole("Expected
    statement");
245         }
246
247     }
248
249     @Override
250     public void parseBlock(Queue<String> tokens) {
251         assert tokens != null : "Violation of: tokens is not
    null";
252         assert tokens.length() > 0 : ""
253                 + "Violation of: Tokenizer.END_OF_INPUT is a
    suffix of tokens";
254
255         String token = tokens.front();
256         int i = 0;
257         while (token.equals("IF") || token.equals("WHILE")
258                 || Tokenizer.isIdentifier(token)) {
259             Statement tempt = this.newInstance();
260
261             tempt.parse(tokens);
262             this.addToBlock(i, tempt);
263             token = tokens.front();
264             i++;
265         }
266
267     }
268
269     /*
270      * Main test method
    ------------------------------------------------------
271      */
272
273     /**
274      * Main method.
275      *
276      * @param args
```

```
277      *            the command line arguments
278      */
279     public static void main(String[] args) {
280         SimpleReader in = new SimpleReader1L();
281         SimpleWriter out = new SimpleWriter1L();
282         /*
283          * Get input file name
284          */
285         out.print("Enter valid BL statement(s) file name: ");
286         String fileName = in.nextLine();
287         /*
288          * Parse input file
289          */
290         out.println("*** Parsing input file ***");
291         Statement s = new Statement1Parse1();
292         SimpleReader file = new SimpleReader1L(fileName);
293         Queue<String> tokens = Tokenizer.tokens(file);
294         file.close();
295         s.parse(tokens); // replace with parseBlock to test
    other method
296         /*
297          * Pretty print the statement(s)
298          */
299         out.println("*** Pretty print of parsed statement(s)
    ***");
300         s.prettyPrint(out, 0);
301
302         in.close();
303         out.close();
304     }
305
306 }
307
```