```
 1 import components.queue.Queue;
 2 import components.simplereader.SimpleReader;
 3 import components.simplereader.SimpleReader1L;
 4 import components.simplewriter.SimpleWriter;
 5 import components.simplewriter.SimpleWriter1L;
 6 import components.statement.Statement;
 7 import components.statement.Statement1;
 8 import components.utilities.Reporter;
 9 import components.utilities.Tokenizer;
10
11 /**
12  * Layered implementation of secondary methods {@code parse} and
13  * {@code parseBlock} for {@code Statement}.
14  *
15  * @author Daniil Gofman and Ansh Pachauri
16  *
17  */
18 public final class Statement1Parse1 extends Statement1 {
19
20     /*
21      * Private members
   ------------------------------------------------------------
22      */
23
24     /**
25      * Converts {@code c} into the corresponding {@code Condition}.
26      *
27      * @param c
28      *            the condition to convert
29      * @return the {@code Condition} corresponding to {@code c}
30      * @requires [c is a condition string]
31      * @ensures parseCondition = [Condition corresponding to c]
32      */
33     private static Condition parseCondition(String c) {
34         assert c != null : "Violation of: c is not null";
35         assert Tokenizer
36                 .isCondition(c) : "Violation of: c is a condition string";
37         return Condition.valueOf(c.replace('-', '_').toUpperCase());
38     }
39
40     /**
41      * Parses an IF or IF_ELSE statement from {@code tokens} into {@code s}.
42      *
43      * @param tokens
44      *            the input tokens
45      * @param s
46      *            the parsed statement
47      * @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("IF") : ""
65                 + "Violation of: <\"IF\"> is proper prefix of tokens";
66
67         String tokenIf = tokens.dequeue();
68         Reporter.assertElseFatalError(tokenIf.equals("IF"),
69                 "Error: " + tokenIf + " is not equal to \"IF\"");
70         //give error if next token is not a condition.
71         Reporter.assertElseFatalError(Tokenizer.isCondition(tokens.front()),
72                 "Error: " + tokens.front() + " is not a valid condition");
73         //parse the condition.
74         Condition condIf = parseCondition(tokens.dequeue());
75         //check for keyword THEN.
76         Reporter.assertElseFatalError(tokens.front().equals("THEN"),
77                 "Error: " + tokens.front() + " is not equal to \"THEN\"");
78         //dequeue "THEN"
79         tokens.dequeue();
80         //parse block under if
81         Statement blockIf = s.newInstance();
82         blockIf.parseBlock(tokens);
83         //check for keyword END or ELSE
84         Reporter.assertElseFatalError(
85                 tokens.front().equals("ELSE") || tokens.front().equals
  ("END"),
86                 "Error: " + tokens.front()
87                         + " is not equal to \"ELSE\" or \"END\"");
88
89         if (tokens.front().equals("ELSE")) {
90             //dequeue "ELSE"
91             tokens.dequeue();
92             //parse block under else
93             Statement blockElse = s.newInstance();
94             blockElse.parseBlock(tokens);
95             //assemble if_else
96             s.assembleIfElse(condIf, blockIf, blockElse);
```

```
 97            //check for keyword END
 98            Reporter.assertElseFatalError(tokens.front().equals("END"),
 99                    "Error: " + tokens.front() + " is not equal to \"END
   \"");
100            //dequeue "END"
101            tokens.dequeue();
102        } else {
103            //assemble if
104            s.assembleIf(condIf, blockIf);
105            //check for keyword END
106            Reporter.assertElseFatalError(tokens.front().equals("END"),
107                    "Error: " + tokens.front() + " is not equal to \"END
   \"");
108            //dequeue "END"
109            tokens.dequeue();
110        }
111        //check for keyword IF
112        Reporter.assertElseFatalError(tokens.front().equals(tokenIf),
113                "Error: " + tokens.front() + " is not equal to \"IF\"");
114        //dequeue "IF"
115        tokens.dequeue();
116
117    }
118
119    /**
120     * Parses a WHILE statement from {@code tokens} into {@code s}.
121     *
122     * @param tokens
123     *            the input tokens
124     * @param s
125     *            the parsed statement
126     * @replaces s
127     * @updates tokens
128     * @requires <pre>
129     * [<"WHILE"> is a prefix of tokens]  and
130     *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
131     * </pre>
132     * @ensures <pre>
133     * if [a while string is a proper prefix of #tokens] then
134     *  s = [WHILE Statement corresponding to while string at start of
   #tokens]  and
135     *  #tokens = [while string at start of #tokens] * tokens
136     * else
137     *  [reports an appropriate error message to the console and terminates
   client]
138     * </pre>
139     */
140    private static void parseWhile(Queue<String> tokens, Statement s) {
141        assert tokens != null : "Violation of: tokens is not null";
142        assert s != null : "Violation of: s is not null";
```

```
143         assert tokens.length() > 0 && tokens.front().equals("WHILE") : ""
144                 + "Violation of: <\"WHILE\"> is proper prefix of tokens";
145         //check for keyword WHILE and dequeue WHILE
146         String tokenWhile = tokens.dequeue();
147         Reporter.assertElseFatalError(tokenWhile.equals("WHILE"),
148                 "Error: " + tokenWhile + " is not equal to \"WHILE\"");
149         //give error if next token is not a condition
150         Reporter.assertElseFatalError(Tokenizer.isCondition(tokens.front()),
151                 "Error: " + tokens.front() + " is not a valid condition");
152         //parse the condition
153         Condition condWhile = parseCondition(tokens.dequeue());
154         //check and dequeue keyword DO
155         Reporter.assertElseFatalError(tokens.front().equals("DO"),
156                 "Error: " + tokens.front() + " is not equal to \"DO\"");
157
158         tokens.dequeue();
159         //parse block under while
160         Statement blockWhile = s.newInstance();
161         blockWhile.parseBlock(tokens);
162         //assemble while
163         s.assembleWhile(condWhile, blockWhile);
164         //check and dequeue for keyword END
165         Reporter.assertElseFatalError(tokens.front().equals("END"),
166                 "Error: " + tokens.front() + " is not equal to \"END\"");
167
168         tokens.dequeue();
169         //check for keyword WHILE and dequeue WHILE
170         Reporter.assertElseFatalError(tokens.front().equals(tokenWhile),
171                 "Error: " + tokens.front() + " is not equal to \"WHILE\"");
172
173         tokens.dequeue();
174
175     }
176
177     /**
178      * Parses a CALL statement from {@code tokens} into {@code s}.
179      *
180      * @param tokens
181      *            the input tokens
182      * @param s
183      *            the parsed statement
184      * @replaces s
185      * @updates tokens
186      * @requires [identifier string is a proper prefix of tokens]
187      * @ensures <pre>
188      * s =
189      *   [CALL Statement corresponding to identifier string at start of
   #tokens]  and
190      *  #tokens = [identifier string at start of #tokens] * tokens
191      * </pre>
```

```
192     */
193    private static void parseCall(Queue<String> tokens, Statement s) {
194        assert tokens != null : "Violation of: tokens is not null";
195        assert s != null : "Violation of: s is not null";
196        assert tokens.length() > 0
197                && Tokenizer.isIdentifier(tokens.front()) : ""
198                        + "Violation of: identifier string is proper prefix
   of tokens";
199        //dequeue call and assemble the call
200        String call = tokens.dequeue();
201        s.assembleCall(call);
202
203    }
204
205    /*
206     * Constructors
   ------------------------------------------------------------
207     */
208
209    /**
210     * No-argument constructor.
211     */
212    public Statement1Parse1() {
213        super();
214    }
215
216    /*
217     * Public methods
   ------------------------------------------------------------
218     */
219
220    @Override
221    public void parse(Queue<String> tokens) {
222        assert tokens != null : "Violation of: tokens is not null";
223        assert tokens.length() > 0 : ""
224                + "Violation of: Tokenizer.END_OF_INPUT is a suffix of
   tokens";
225        //check if next token is not "IF", "WHILE", or an identifier
226        Reporter.assertElseFatalError(
227                Tokenizer.isIdentifier(tokens.front())
228                        || tokens.front().equals("IF")
229                        || tokens.front().equals("WHILE"),
230                tokens.front() + " is not IF, IF_ELSE, WHILE, or CALL");
231
232        if (tokens.front().equals("IF")) {
233            //parse the if
234            parseIf(tokens, this);
235        } else if (tokens.front().equals("WHILE")) {
236            //parse the while
237            parseWhile(tokens, this);
```

```
238         } else {
239             //parse call
240             parseCall(tokens, this);
241         }
242
243     }
244
245     @Override
246     public void parseBlock(Queue<String> tokens) {
247         assert tokens != null : "Violation of: tokens is not null";
248         assert tokens.length() > 0 : ""
249                 + "Violation of: Tokenizer.END_OF_INPUT is a suffix of
    tokens";
250
251         Statement stmt = this.newInstance();
252         //check if next token is not "IF", "WHILE", or an identifier
253         Reporter.assertElseFatalError(Tokenizer.isIdentifier(tokens.front())
254                 || tokens.front().equals("IF") || tokens.front().equals
    ("WHILE")
255                 || tokens.front().equals("END"),
256                 tokens.front() + " is not IF, IF_ELSE, WHILE, END, or
    CALL");
257
258         int i = 0;
259         while (Tokenizer.isIdentifier(tokens.front())
260                 || tokens.front().equals("IF")
261                 || tokens.front().equals("WHILE")) {
262             //parse the statement
263             stmt.parse(tokens);
264             this.addToBlock(i, stmt);
265             i++;
266         }
267
268     }
269
270     /*
271      * Main test method
    ------------------------------------------------------
272      */
273
274     /**
275      * Main method.
276      *
277      * @param args
278      *            the command line arguments
279      */
280     public static void main(String[] args) {
281         SimpleReader in = new SimpleReader1L();
282         SimpleWriter out = new SimpleWriter1L();
283         /*
```

```
284         * Get input file name
285         */
286        out.print("Enter valid BL statement(s) file name: ");
287        String fileName = in.nextLine();
288        /*
289         * Parse input file
290         */
291        out.println("*** Parsing input file ***");
292        Statement s = new Statement1Parse1();
293        SimpleReader file = new SimpleReader1L(fileName);
294        Queue<String> tokens = Tokenizer.tokens(file);
295        file.close();
296        s.parse(tokens); // replace with parseBlock to test other method
297        /*
298         * Pretty print the statement(s)
299         */
300        out.println("*** Pretty print of parsed statement(s) ***");
301        s.prettyPrint(out, 0);
302
303        in.close();
304        out.close();
305    }
306
307}
308
```