

```

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 Chloe Feller and Krish Patel
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         /**
68          * Check the keyword and condition.
69          */
70         String start = tokens.dequeue();
71         Reporter.assertElseFatalError(Tokenizer.isKeyword(start),
72             "Invalid token");
73
74         String condition = tokens.dequeue();
75         Reporter.assertElseFatalError(
76             tokens.length() > 0 && Tokenizer.isCondition(condition),
77             "Invalid condition");
78         Condition c = parseCondition(condition);
79
80         Reporter.assertElseFatalError(tokens.dequeue().equals("THEN"),
81             "Invalid token");
82
83         /**
84          * Parse and assemble "IF" or "IF_ELSE".
85          */
86         Statement s1 = s.newInstance();
87         s1.parseBlock(tokens);
88         if (tokens.front().equals("ELSE")) {
89             tokens.dequeue();
90             Statement s2 = s.newInstance();
91             s2.parseBlock(tokens);
92             s.assembleIfElse(c, s1, s2);
93         } else {
94             s.assembleIf(c, s1);
95         }
96
97         /**
98          * Check the end.
99          */
100        Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
101            "Invalid token");
102        Reporter.assertElseFatalError(Tokenizer.isKeyword(tokens.dequeue()),
103            "Invalid token");
104    }
105
106    /**
107     * Parses a WHILE statement from {@code tokens} into {@code s}.
108     *
109     * @param tokens
110     *     the input tokens
111     * @param s
112     *     the parsed statement
113     * @replaces s
114     * @updates tokens

```

```

115     * @requires <pre>
116     * [<"WHILE"> is a prefix of tokens] and
117     * [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
118     * </pre>
119     * @ensures <pre>
120     * if [a while string is a proper prefix of #tokens] then
121     *   s = [WHILE Statement corresponding to while string at start of #tokens] and
122     *   #tokens = [while string at start of #tokens] * tokens
123     * else
124     *   [reports an appropriate error message to the console and terminates client]
125     * </pre>
126     */
127     private static void parseWhile(Queue<String> tokens, Statement s) {
128         assert tokens != null : "Violation of: tokens is not null";
129         assert s != null : "Violation of: s is not null";
130         assert tokens.length() > 0 && tokens.front().equals("WHILE") : ""
131             + "Violation of: <\\"WHILE\\"> is proper prefix of tokens";
132
133         String start = tokens.dequeue();
134         Reporter.assertElseFatalError(Tokenizer.isKeyword(start),
135             "Invalid token");
136
137         Reporter.assertElseFatalError(Tokenizer.isCondition(tokens.front()),
138             "Invalid condition");
139         Condition c = parseCondition(tokens.dequeue());
140
141         Reporter.assertElseFatalError(tokens.dequeue().equals("DO"),
142             "Invalid token");
143
144         Statement s1 = s.newInstance();
145         s1.parseBlock(tokens);
146         s.assembleWhile(c, s1);
147
148         Reporter.assertElseFatalError(start.equals("WHILE"),
149             "expecting 'WHILE'");
150         start = tokens.dequeue();
151
152         Reporter.assertElseFatalError(start.equals("END"), "Invalid token");
153         Reporter.assertElseFatalError(Tokenizer.isKeyword(tokens.dequeue()),
154             "Invalid token");
155     }
156 }
157
158 /**
159  * Parses a CALL statement from {@code tokens} into {@code s}.
160  *
161  * @param tokens
162  *     the input tokens
163  * @param s
164  *     the parsed statement
165  * @replaces s
166  * @updates tokens
167  * @requires [identifier string is a proper prefix of tokens]
168  * @ensures <pre>
169  *   s =
170  *   [CALL Statement corresponding to identifier string at start of #tokens] and
171  *   #tokens = [identifier string at start of #tokens] * tokens

```

```

172     * </pre>
173     */
174     private static void parseCall(Queue<String> tokens, Statement s) {
175         assert tokens != null : "Violation of: tokens is not null";
176         assert s != null : "Violation of: s is not null";
177         assert tokens.length() > 0
178             && Tokenizer.isIdentifier(tokens.front()) : ""
179             + "Violation of: identifier string is proper prefix of tokens";
180
181         String name = tokens.dequeue();
182         Reporter.assertElseFatalError(Tokenizer.isIdentifier(name),
183             "Invalid token");
184         s.assembleCall(name);
185     }
186
187     /*
188     * Constructors -----
189     */
190
191     /**
192     * No-argument constructor.
193     */
194     public Statement1Parse1() {
195         super();
196     }
197
198     /*
199     * Public methods -----
200     */
201
202     @Override
203     public void parse(Queue<String> tokens) {
204         assert tokens != null : "Violation of: tokens is not null";
205         assert tokens.length() > 0 : ""
206             + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
207
208         String token = tokens.front();
209
210         Reporter.assertElseFatalError(
211             Tokenizer.isIdentifier(token) || Tokenizer.isKeyword(token),
212             "Invalid token");
213
214         if (token.equals("WHILE")) {
215             parseWhile(tokens, this);
216         } else if (token.equals("IF")) {
217             parseIf(tokens, this);
218         } else {
219             parseCall(tokens, this);
220         }
221     }
222
223     @Override
224     public void parseBlock(Queue<String> tokens) {
225         assert tokens != null : "Violation of: tokens is not null";
226         assert tokens.length() > 0 : ""
227             + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";

```

```

229
230     this.clear();
231
232     String token = tokens.front();
233
234     for (int i = 0; Tokenizer.isIdentifier(token) || token.equals("IF")
235           || token.equals("WHILE"); i++) {
236         Statement s = this.newInstance();
237
238         s.parse(tokens);
239         this.addToBlock(i, s);
240
241         token = tokens.front();
242     }
243
244 }
245
246 /*
247  * Main test method -----
248  */
249
250 /**
251  * Main method.
252  *
253  * @param args
254  *     the command line arguments
255  */
256 public static void main(String[] args) {
257     SimpleReader in = new SimpleReader1L();
258     SimpleWriter out = new SimpleWriter1L();
259     /*
260      * Get input file name
261      */
262     out.print("Enter valid BL statement(s) file name: ");
263     String fileName = in.nextLine();
264     /*
265      * Parse input file
266      */
267     out.println("*** Parsing input file ***");
268     Statement s = new Statement1Parse1();
269     SimpleReader file = new SimpleReader1L(fileName);
270     Queue<String> tokens = Tokenizer.tokens(file);
271     file.close();
272     s.parse(tokens); // replace with parseBlock to test other method
273     /*
274      * Pretty print the statement(s)
275      */
276     out.println("*** Pretty print of parsed statement(s) ***");
277     s.prettyPrint(out, 0);
278
279     in.close();
280     out.close();
281 }
282
283 }
284

```