

```

1 import components.queue.Queue;
10
11 /**
12  * Layered implementation of secondary methods {@code parse} and
13  * {@code parseBlock} for {@code Statement}.
14  *
15  * @author Qinuo Shi & Yiming Cheng
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     // TODO - fill in body
68     tokens.dequeue();
69
70     /*
71     * If the BL format is not found, report an error here
72     */
73     String con = tokens.dequeue();
74     Reporter.assertElseFatalError(Tokenizer.isCondition(con),
75         "Cannot find CONDITION.");
76
77     Condition ifCon = parseCondition(con);
78
79     Reporter.assertElseFatalError(tokens.dequeue().equals("THEN"),
80         "Cannot find THEN.");
81
82     Statement tool = s.newInstance();
83     tool.transferFrom(s);
84
85     /*
86     * Parse IF without either END or ELSE
87     */
88     while (!tokens.front().equals("END")
89         && !tokens.front().equals("ELSE")) {
90         tool.parseBlock(tokens);
91     }
92
93     /*
94     * Parse IF with ELSE
95     */
96     if (tokens.front().equals("ELSE")) {
97         tokens.dequeue();
98         Statement elseBlock = s.newInstance();
99
100         while (!tokens.front().equals("END")) {
101             elseBlock.parseBlock(tokens);
102         }
103         /*
104         * Assemble IF and ELSE
105         */
106         s.assembleIfElse(ifCon, tool, elseBlock);
107     } else {
108         /*
109         * Assemble IF
110         */
111         s.assembleIf(ifCon, tool);
112     }
113
114     tokens.dequeue();
115
116     /*
117     * Reporting an error if IF cannot be found.
118     */
119     String testForIF = tokens.dequeue();
120     Reporter.assertElseFatalError(testForIF.equals("IF"), "Cannot find IF");
121
122 }
```

```

123
124  /**
125   * Parses a WHILE statement from {@code tokens} into {@code s}.
126   *
127   * @param tokens
128   *         the input tokens
129   * @param s
130   *         the parsed statement
131   * @replaces s
132   * @updates tokens
133   * @requires <pre>
134   *   [<"WHILE"> is a prefix of tokens] and
135   *   [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
136   * </pre>
137   * @ensures <pre>
138   *   if [a while string is a proper prefix of #tokens] then
139   *     s = [WHILE Statement corresponding to while string at start of #tokens] and
140   *     #tokens = [while string at start of #tokens] * tokens
141   *   else
142   *     [reports an appropriate error message to the console and terminates client]
143   * </pre>
144   */
145  private static void parseWhile(Queue<String> tokens, Statement s) {
146      assert tokens != null : "Violation of: tokens is not null";
147      assert s != null : "Violation of: s is not null";
148      assert tokens.length() > 0 && tokens.front().equals("WHILE") : ""
149          + "Violation of: <\\"WHILE\\"> is proper prefix of tokens";
150
151      // TODO - fill in body
152      tokens.dequeue();
153
154      /*
155       * If the BL format is not found, report an error here
156       */
157      String con = tokens.dequeue();
158      Reporter.assertElseFatalError(Tokenizer.isCondition(con),
159          "Cannot find CONDITION.");
160
161      Condition whileCon = parseCondition(con);
162
163      Reporter.assertElseFatalError(tokens.dequeue().equals("DO"),
164          "Cannot find DO");
165
166      /*
167       * Parse WHILE
168       */
169      s.parseBlock(tokens);
170
171      Statement tool = s.newInstance();
172      tool.transferFrom(s);
173      /*
174       * Assemble WHILE
175       */
176      s.assembleWhile(whileCon, tool);
177
178      /*
179       * Report an error if END and WHILE cannot be found

```

```

180     */
181     Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
182         "Cannot find END");
183     Reporter.assertElseFatalError(tokens.dequeue().equals("WHILE"),
184         "Cannot find WHILE");
185 }
186
187 /**
188  * Parses a CALL statement from {@code tokens} into {@code s}.
189  *
190  * @param tokens
191  *     the input tokens
192  * @param s
193  *     the parsed statement
194  * @replaces s
195  * @updates tokens
196  * @requires [identifier string is a proper prefix of tokens]
197  * @ensures <pre>
198  *     s =
199  *     [CALL Statement corresponding to identifier string at start of #tokens] and
200  *     #tokens = [identifier string at start of #tokens] * tokens
201  * </pre>
202  */
203 private static void parseCall(Queue<String> tokens, Statement s) {
204     assert tokens != null : "Violation of: tokens is not null";
205     assert s != null : "Violation of: s is not null";
206     assert tokens.length() > 0
207         && Tokenizer.isIdentifier(tokens.front()) : ""
208         + "Violation of: identifier string is proper prefix of tokens";
209
210     // TODO - fill in body
211     s.assembleCall(tokens.dequeue());
212 }
213
214
215 /**
216  * Constructors -----
217  */
218
219 /**
220  * No-argument constructor.
221  */
222 public Statement1Parse1() {
223     super();
224 }
225
226 /**
227  * Public methods -----
228  */
229
230 @Override
231 public void parse(Queue<String> tokens) {
232     assert tokens != null : "Violation of: tokens is not null";
233     assert tokens.length() > 0 : ""
234         + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
235
236     // TODO - fill in body

```

```

237     /*
238     * Run them in different method by identifying their kind
239     */
240     if (tokens.front().equals("IF")) {
241         parseIf(tokens, this);
242     } else if (tokens.front().equals("WHILE")) {
243         parseWhile(tokens, this);
244     } else {
245         parseCall(tokens, this);
246     }
247
248 }
249
250 @Override
251 public void parseBlock(Queue<String> tokens) {
252     assert tokens != null : "Violation of: tokens is not null";
253     assert tokens.length() > 0 : ""
254         + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
255
256     // TODO - fill in body
257     /*
258     * Block as a special condition, running here
259     */
260     while ((!tokens.front().equals("ELSE") && !tokens.front().equals("END")
261         && !tokens.front().equals(Tokenizer.END_OF_INPUT))) {
262         Statement tool = this.newInstance();
263         tool.parse(tokens);
264         this.addToBlock(this.lengthOfBlock(), tool);
265         tool.clear();
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
```