```java
import static org.junit.Assert.assertEquals;

/**
 * JUnit test fixture for {@code Statement}'s constructor and kernel methods.
 *
 * @author Put your name here
 *
 */
public abstract class StatementTest {

    /**
     * The name of a file containing a sequence of BL statements.
     */
    private static final String FILE_NAME_1 = "test/statement1.bl",
            FILE_NAME_2 = "test/statement2.bl", FILE_NAME_3 = "test/statement3.bl",
            FILE_NAME_4 = "test/statement4.bl", FILE_NAME_5 = "test/statement5.bl",
            FILE_NAME_6 = "test/statement6.bl";

    /**
     * Invokes the {@code Statement} constructor for the implementation under
     * test and returns the result.
     *
     * @return the new statement
     * @ensures constructorTest = compose((BLOCK, ?, ?), <>)
     */
    protected abstract Statement constructorTest();

    /**
     * Invokes the {@code Statement} constructor for the reference
     * implementation and returns the result.
     *
     * @return the new statement
     * @ensures constructorRef = compose((BLOCK, ?, ?), <>)
     */
    protected abstract Statement constructorRef();

    /**
     * Test of parse on syntactically valid input.
     */
    @Test
    public final void testParseValid1() {
        /*
         * Setup
         */
        Statement sRef = this.constructorRef();
        SimpleReader file = new SimpleReader1L(FILE_NAME_1);
        Queue<String> tokens = Tokenizer.tokens(file);
        sRef.parse(tokens);
        file.close();
        Statement sTest = this.constructorTest();
        file = new SimpleReader1L(FILE_NAME_1);
        tokens = Tokenizer.tokens(file);
        file.close();
        /*
         * The call
         */
        sTest.parse(tokens);
        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
    }

    /**
     * Test of parse on syntactically invalid input.
     */
    @Test(expected = RuntimeException.class)
    public final void testParseError2() {
        /*
         * Setup
         */
        Statement sTest = this.constructorTest();
        SimpleReader file = new SimpleReader1L(FILE_NAME_2);
        Queue<String> tokens = Tokenizer.tokens(file);
        file.close();
        /*
         * The call--should result in an error being caught
         */
        sTest.parse(tokens);
    }

    /**
     * Test of parse on syntactically invalid input.
```

```
 92      */
 93     @Test(expected = RuntimeException.class)
 94     public final void testParseError3() {
 95         /*
 96          * Setup
 97          */
 98         Statement sTest = this.constructorTest();
 99         SimpleReader file = new SimpleReader1L(FILE_NAME_3);
100         Queue<String> tokens = Tokenizer.tokens(file);
101         file.close();
102         /*
103          * The call--should result in an error being caught
104          */
105         sTest.parse(tokens);
106     }
107
108     /**
109      * Test of parse on syntactically invalid input.
110      */
111     @Test(expected = RuntimeException.class)
112     public final void testParseError4() {
113         /*
114          * Setup
115          */
116         Statement sTest = this.constructorTest();
117         SimpleReader file = new SimpleReader1L(FILE_NAME_4);
118         Queue<String> tokens = Tokenizer.tokens(file);
119         file.close();
120         /*
121          * The call--should result in an error being caught
122          */
123         sTest.parse(tokens);
124     }
125
126     /**
127      * Test of parse on syntactically invalid input.
128      */
129     @Test(expected = RuntimeException.class)
130     public final void testParseError5() {
131         /*
132          * Setup
133          */
134         Statement sTest = this.constructorTest();
135         SimpleReader file = new SimpleReader1L(FILE_NAME_5);
136         Queue<String> tokens = Tokenizer.tokens(file);
137         file.close();
138         /*
139          * The call--should result in an error being caught
140          */
141         sTest.parse(tokens);
142     }
143
144     /**
145      * Test of parse on syntactically valid input.
146      */
147     @Test
148     public final void testParseValid6() {
149         /*
150          * Setup
151          */
152         Statement sRef = this.constructorRef();
153         SimpleReader file = new SimpleReader1L(FILE_NAME_6);
154         Queue<String> tokens = Tokenizer.tokens(file);
155         sRef.parse(tokens);
156         file.close();
157         Statement sTest = this.constructorTest();
158         file = new SimpleReader1L(FILE_NAME_6);
159         tokens = Tokenizer.tokens(file);
160         file.close();
161         /*
162          * The call
163          */
164         sTest.parse(tokens);
165         /*
166          * Evaluation
167          */
168         assertEquals(sRef, sTest);
169     }
170
171     /**
172      *
173      * Test parseBlock with valid input
174      *
```

```
175      */
176
177     @Test
178
179     public final void testParseBlockValid() {
180
181         Statement sRef = this.constructorRef();
182
183         Statement sTest = this.constructorTest();
184
185         SimpleReader inFile = new SimpleReader1L(FILE_NAME_1);
186
187         Queue<String> tokens = Tokenizer.tokens(inFile);
188
189         sRef.parseBlock(tokens);
190
191         inFile.close();
192
193         inFile = new SimpleReader1L(FILE_NAME_1);
194
195         tokens = Tokenizer.tokens(inFile);
196
197         inFile.close();
198
199         sTest.parseBlock(tokens);
200
201         assertEquals(sRef, sTest);
202
203     }
204
205     /**
206      *
207      * Test parseBlock with invalid input.
208      *
209      */
210
211     @Test(expected = RuntimeException.class)
212
213     public final void testParseBlockError() {
214
215         Statement sTest = this.constructorTest();
216
217         SimpleReader file = new SimpleReader1L(FILE_NAME_2);
218
219         Queue<String> tokens = Tokenizer.tokens(file);
220
221         file.close();
222
223         /*
224          *
225          * The call--should result in an error being caught
226          *
227          */
228
229         sTest.parseBlock(tokens);
230
231     }
232
233     /**
234      *
235      * Test of parse on valid nested input.
236      *
237      */
238
239     @Test
240
241     public final void testParseNestedIfElse() {
242
243         Statement sRef = this.constructorRef();
244
245         Statement sTest = this.constructorTest();
246
247         SimpleReader file = new SimpleReader1L(FILE_NAME_1);
248
249         Queue<String> tokens = Tokenizer.tokens(file);
250
251         sRef.parse(tokens);
252
253         file.close();
254
255         file = new SimpleReader1L(FILE_NAME_1);
256
257         tokens = Tokenizer.tokens(file);
```

```
258
259         file.close();
260
261         sTest.parse(tokens);
262
263         assertEquals(sRef, sTest);
264
265     }
266
267     // TODO - add more test cases for valid inputs for both parse and parseBlock
268     // TODO - add more test cases for as many distinct syntax errors as possible
269     //        for both parse and parseBlock
270
271 }
272
```