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

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

    /**
     * The names of a files containing a (possibly invalid) BL programs.
     */
    private static final String FILE_NAME_1 = "test/program/program1.bl",
            FILE_NAME_2 = "test/program/program2.bl",
            FILE_NAME_3 = "test/program/program3.bl",
            FILE_NAME_4 = "test/program/program4.bl",
            FILE_NAME_5 = "test/program/program5.bl",
            FILE_NAME_6 = "test/program/program6.bl",
            FILE_NAME_7 = "test/program/program7.bl";

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

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

    /**
     * Test of parse on syntactically valid input.
     */
    @Test
    public final void testParseValidExample() {
        /*
         * Setup
         */
        Program pRef = this.constructorRef();
        SimpleReader file = new SimpleReader1L(FILE_NAME_1);
        pRef.parse(file);
        file.close();
        Program pTest = this.constructorTest();
        file = new SimpleReader1L(FILE_NAME_1);
        Queue<String> tokens = Tokenizer.tokens(file);
        file.close();
        /*
         * The call
```

```
66          */
67         pTest.parse(tokens);
68         /*
69          * Evaluation
70          */
71         assertEquals(pRef, pTest);
72     }
73
74     /**
75      * Test of parse on syntactically invalid input (instruction is not unique).
76      */
77     @Test(expected = RuntimeException.class)
78     public final void testParseErrorExample() {
79         /*
80          * Setup
81          */
82         Program pTest = this.constructorTest();
83         SimpleReader file = new SimpleReader1L(FILE_NAME_2);
84         Queue<String> tokens = Tokenizer.tokens(file);
85         file.close();
86         /*
87          * The call--should result in a syntax error being found
88          */
89         pTest.parse(tokens);
90     }
91
92     /**
93      * Test of parse with a valid input.
94      */
95     @Test
96     public final void testParseValidExample2() {
97         /*
98          * Setup
99          */
100        Program pRef = this.constructorRef();
101        SimpleReader file = new SimpleReader1L(FILE_NAME_3);
102        pRef.parse(file);
103        file.close();
104        Program pTest = this.constructorTest();
105        file = new SimpleReader1L(FILE_NAME_3);
106        Queue<String> tokens = Tokenizer.tokens(file);
107        file.close();
108        /*
109         * The call
110         */
111        pTest.parse(tokens);
112        /*
113         * Evaluation
114         */
115        assertEquals(pRef, pTest);
116    }
117
118    /**
119     * Test of parse with a valid input.
120     */
121    @Test
122    public final void testParseValidExample3() {
```

```
123         /*
124          * Setup
125          */
126         Program pRef = this.constructorRef();
127         SimpleReader file = new SimpleReader1L(FILE_NAME_4);
128         pRef.parse(file);
129         file.close();
130         Program pTest = this.constructorTest();
131         file = new SimpleReader1L(FILE_NAME_4);
132         Queue<String> tokens = Tokenizer.tokens(file);
133         file.close();
134         /*
135          * The call
136          */
137         pTest.parse(tokens);
138         /*
139          * Evaluation
140          */
141         assertEquals(pRef, pTest);
142     }
143 
144     /**
145      * Test of parse with an invalid input (instruction is not unique).
146      */
147     @Test(expected = RuntimeException.class)
148     public final void testParseErrorExample3() {
149         /*
150          * Setup
151          */
152         Program pTest = this.constructorTest();
153         SimpleReader file = new SimpleReader1L(FILE_NAME_5);
154         Queue<String> tokens = Tokenizer.tokens(file);
155         file.close();
156         /*
157          * The call--should result in a syntax error being found
158          */
159         pTest.parse(tokens);
160     }
161 
162     /**
163      * Test of parse with an invalid input (instruction is a primitive call).
164      */
165     @Test(expected = RuntimeException.class)
166     public final void testParseErrorExample4() {
167         /*
168          * Setup
169          */
170         Program pTest = this.constructorTest();
171         SimpleReader file = new SimpleReader1L(FILE_NAME_6);
172         Queue<String> tokens = Tokenizer.tokens(file);
173         file.close();
174         /*
175          * The call--should result in a syntax error being found
176          */
177         pTest.parse(tokens);
178     }
179 
```

```
180     /**
181      * Test of parse with an invalid input (instruction names at beginning and
182      * end are different).
183      */
184     @Test(expected = RuntimeException.class)
185     public final void testParseErrorExample6() {
186         /*
187          * Setup
188          */
189         Program pTest = this.constructorTest();
190         SimpleReader file = new SimpleReader1L(FILE_NAME_7);
191         Queue<String> tokens = Tokenizer.tokens(file);
192         file.close();
193         /*
194          * The call--should result in a syntax error being found
195          */
196         pTest.parse(tokens);
197     }
198
199 }
200
```