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

/**
 * JUnit test fixture for {@code Statement}'s constructor and kernel methods.
 *
 * @author Wayne Heym
 * @author Zhuoyang Li
 *
 */
public abstract class StatementTest {

    /**
     * The name of a file containing a sequence of BL statements.
     */
    private static final String FILE_NAME_1 = "data/statement-sample.bl";

    // TODO - define file names for additional test inputs
    /**
     * The name of a file containing a sequence of BL statements.
     */
    private static final String FILE_NAME_2 = "data/statement-sampleTest.bl";

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

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

    /**
     *
     * Creates and returns a block {@code Statement}, of the type of the
     * implementation under test, from the file with the given name.
     *
     * @param filename
     *            the name of the file to be parsed for the sequence of
     *            statements to go in the block statement
     * @return the constructed block statement
     * @ensures <pre>
     * createFromFile = [the block statement containing the statements
     * parsed from the file]
     * </pre>
     */
    private Statement createFromFileTest(String filename) {
        Statement s = this.constructorTest();
        SimpleReader file = new SimpleReader1L(filename);
        Queue<String> tokens = Tokenizer.tokens(file);
        s.parseBlock(tokens);
        file.close();
        return s;
    }

    /**
     *
     * Creates and returns a block {@code Statement}, of the reference
     * implementation type, from the file with the given name.
     *
     * @param filename
     *            the name of the file to be parsed for the sequence of
     *            statements to go in the block statement
     * @return the constructed block statement
     * @ensures <pre>
     * createFromFile = [the block statement containing the statements
     * parsed from the file]
     * </pre>
     */
    private Statement createFromFileRef(String filename) {
        Statement s = this.constructorRef();
        SimpleReader file = new SimpleReader1L(filename);
        Queue<String> tokens = Tokenizer.tokens(file);
        s.parseBlock(tokens);
        file.close();
        return s;
    }

    /**
     * Test constructor.
     */
    @Test
    public final void testConstructor() {
        /*
         * Setup
         */
        Statement sRef = this.constructorRef();

        /*
         * The call
         */
        Statement sTest = this.constructorTest();

        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
    }

    /**
     * Test kind of a WHILE statement.
     */
    @Test
    public final void testKindWhile() {
        /*
         * Setup
         */
        final int whilePos = 3;
        Statement sourceTest =
  this.createFromFileTest(FILE_NAME_1);
        Statement sourceRef =
  this.createFromFileRef(FILE_NAME_1);
        Statement sTest =
  sourceTest.removeFromBlock(whilePos);
        Statement sRef = sourceRef.removeFromBlock(whilePos);
        Kind kRef = sRef.kind();

        /*
```

```java
         * The call
         */
        Kind kTest = sTest.kind();

        /*
         * Evaluation
         */
        assertEquals(kRef, kTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test addToBlock at an interior position.
     */
    @Test
    public final void testAddToBlockInterior() {
        /*
         * Setup
         */
        Statement sTest = this.createFromFileTest(FILE_NAME_1);
        Statement sRef = this.createFromFileRef(FILE_NAME_1);
        Statement emptyBlock = sRef.newInstance();
        Statement nestedTest = sTest.removeFromBlock(1);
        Statement nestedRef = sRef.removeFromBlock(1);
        sRef.addToBlock(2, nestedRef);

        /*
         * The call
         */
        sTest.addToBlock(2, nestedTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, nestedTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test removeFromBlock at the front leaving a non-empty block behind.
     */
    @Test
    public final void testRemoveFromBlockFrontLeavingNonEmpty() {
        /*
```

```
         * Setup
         */
        Statement sTest =
this.createFromFileTest(FILE_NAME_1);
        Statement sRef = this.createFromFileRef(FILE_NAME_1);
        Statement nestedRef = sRef.removeFromBlock(0);

        /*
         * The call
         */
        Statement nestedTest = sTest.removeFromBlock(0);

        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
        assertEquals(nestedRef, nestedTest);
    }

    /**
     * Test lengthOfBlock, greater than zero.
     */
    @Test
    public final void testLengthOfBlockNonEmpty() {
        /*
         * Setup
         */
        Statement sTest =
this.createFromFileTest(FILE_NAME_1);
        Statement sRef = this.createFromFileRef(FILE_NAME_1);
        int lengthRef = sRef.lengthOfBlock();

        /*
         * The call
         */
        int lengthTest = sTest.lengthOfBlock();

        /*
         * Evaluation
         */
        assertEquals(lengthRef, lengthTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test assembleIf.
     */
```

```java
    @Test
    public final void testAssembleIf() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_1);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_1);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(1);
        Statement sRef = blockRef.removeFromBlock(1);
        Statement nestedTest = sourceTest.newInstance();
        Condition c = sourceTest.disassembleIf(nestedTest);
        Statement sTest = sourceTest.newInstance();

        /*
         * The call
         */
        sTest.assembleIf(c, nestedTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, nestedTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test disassembleIf.
     */
    @Test
    public final void testDisassembleIf() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_1);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_1);
        Statement sTest = blockTest.removeFromBlock(1);
        Statement sRef = blockRef.removeFromBlock(1);
        Statement nestedTest = sTest.newInstance();
        Statement nestedRef = sRef.newInstance();
        Condition cRef = sRef.disassembleIf(nestedRef);

        /*
```

```java
         * The call
         */
        Condition cTest = sTest.disassembleIf(nestedTest);

        /*
         * Evaluation
         */
        assertEquals(nestedRef, nestedTest);
        assertEquals(sRef, sTest);
        assertEquals(cRef, cTest);
    }

    /**
     * Test assembleIfElse.
     */
    @Test
    public final void testAssembleIfElse() {
        /*
         * Setup
         */
        final int ifElsePos = 2;
        Statement blockTest = this.createFromFileTest(FILE_NAME_1);
        Statement blockRef = this.createFromFileRef(FILE_NAME_1);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(ifElsePos);
        Statement sRef = blockRef.removeFromBlock(ifElsePos);
        Statement thenBlockTest = sourceTest.newInstance();
        Statement elseBlockTest = sourceTest.newInstance();
        Condition cTest = sourceTest.disassembleIfElse(thenBlockTest,
                elseBlockTest);
        Statement sTest = blockTest.newInstance();

        /*
         * The call
         */
        sTest.assembleIfElse(cTest, thenBlockTest, elseBlockTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, thenBlockTest);
        assertEquals(emptyBlock, elseBlockTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test disassembleIfElse.
     */
    @Test
    public final void testDisassembleIfElse() {
        /*
         * Setup
         */
        final int ifElsePos = 2;
        Statement blockTest = this.createFromFileTest(FILE_NAME_1);
        Statement blockRef = this.createFromFileRef(FILE_NAME_1);
        Statement sTest = blockTest.removeFromBlock(ifElsePos);
        Statement sRef = blockRef.removeFromBlock(ifElsePos);
        Statement thenBlockTest = sTest.newInstance();
        Statement elseBlockTest = sTest.newInstance();
        Statement thenBlockRef = sRef.newInstance();
        Statement elseBlockRef = sRef.newInstance();
        Condition cRef = sRef.disassembleIfElse(thenBlockRef, elseBlockRef);

        /*
         * The call
         */
        Condition cTest = sTest.disassembleIfElse(thenBlockTest, elseBlockTest);

        /*
         * Evaluation
         */
        assertEquals(cRef, cTest);
        assertEquals(thenBlockRef, thenBlockTest);
        assertEquals(elseBlockRef, elseBlockTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test assembleWhile.
     */
    @Test
    public final void testAssembleWhile() {
        /*
```

```java
         * Setup
         */
        Statement blockTest = this.createFromFileTest(FILE_NAME_1);
        Statement blockRef = this.createFromFileRef(FILE_NAME_1);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(1);
        Statement sourceRef = blockRef.removeFromBlock(1);
        Statement nestedTest = sourceTest.newInstance();
        Statement nestedRef = sourceRef.newInstance();
        Condition cTest = sourceTest.disassembleIf(nestedTest);
        Condition cRef = sourceRef.disassembleIf(nestedRef);
        Statement sRef = sourceRef.newInstance();
        sRef.assembleWhile(cRef, nestedRef);
        Statement sTest = sourceTest.newInstance();

        /*
         * The call
         */
        sTest.assembleWhile(cTest, nestedTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, nestedTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test disassembleWhile.
     */
    @Test
    public final void testDisassembleWhile() {
        /*
         * Setup
         */
        final int whilePos = 3;
        Statement blockTest = this.createFromFileTest(FILE_NAME_1);
        Statement blockRef = this.createFromFileRef(FILE_NAME_1);
        Statement sTest = blockTest.removeFromBlock(whilePos);
        Statement sRef = blockRef.removeFromBlock(whilePos);
        Statement nestedTest = sTest.newInstance();
        Statement nestedRef = sRef.newInstance();
        Condition cRef = sRef.disassembleWhile(nestedRef);

        /*
         * The call
         */
        Condition cTest = sTest.disassembleWhile(nestedTest);

        /*
         * Evaluation
         */
        assertEquals(nestedRef, nestedTest);
        assertEquals(sRef, sTest);
        assertEquals(cRef, cTest);
    }

    /**
     * Test assembleCall.
     */
    @Test
    public final void testAssembleCall() {
        /*
         * Setup
         */
        Statement sRef = this.constructorRef().newInstance();
        Statement sTest = this.constructorTest().newInstance();

        String name = "look-for-something";
        sRef.assembleCall(name);

        /*
         * The call
         */
        sTest.assembleCall(name);

        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
    }

    /**
     * Test disassembleCall.
     */
    @Test
    public final void testDisassembleCall() {
        /*
         * Setup
         */
        Statement blockTest = this.createFromFileTest(FILE_NAME_1);
        Statement blockRef = this.createFromFileRef(FILE_NAME_1);
        Statement sTest = blockTest.removeFromBlock(0);
        Statement sRef = blockRef.removeFromBlock(0);
        String nRef = sRef.disassembleCall();

        /*
         * The call
         */
        String nTest = sTest.disassembleCall();

        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
        assertEquals(nRef, nTest);
    }

    // TODO - provide additional test cases to thoroughly test StatementKernel

    /**
     * Test kind of a BLOCK statement.
     */
    @Test
    public final void testKindBlock() {
        /*
         * Setup
         */
        Statement sourceTest = this.createFromFileTest(FILE_NAME_2);
        Statement sourceRef = this.createFromFileRef(FILE_NAME_2);
        Kind kRef = sourceRef.kind();

        /*
         * The call
         */
        Kind kTest = sourceTest.kind();

        /*
         * Evaluation
         */
        assertEquals(kRef, kTest);
        assertEquals(sourceRef, sourceTest);
    }

    /**
     * Test addToBlock at an interior position.
     */

    @Test
    public final void testAddToBlockInterior2() {
        /*
         * Setup
         */
        Statement sTest = this.createFromFileTest(FILE_NAME_2);
        Statement sRef = this.createFromFileRef(FILE_NAME_2);
        Statement emptyBlock = sRef.newInstance();
        Statement nestedTest = sTest.removeFromBlock(1);
        Statement nestedRef = sRef.removeFromBlock(1);
        sRef.addToBlock(2, nestedRef);

        /*
         * The call
         */
        sTest.addToBlock(2, nestedTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, nestedTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test removeFromBlock at the front leaving a non-empty block behind.
     */
    @Test
    public final void testRemoveFromBlockFrontLeavingNonEmpty2() {
        /*
         * Setup
         */
        Statement sTest = this.createFromFileTest(FILE_NAME_2);
        Statement sRef = this.createFromFileRef(FILE_NAME_2);
        Statement nestedRef = sRef.removeFromBlock(0);

        /*
         * The call
         */
        Statement nestedTest = sTest.removeFromBlock(0);

        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
        assertEquals(nestedRef, nestedTest);
    }

    /**
     * Test lengthOfBlock, greater than zero.
     */
    @Test
    public final void testLengthOfBlockNonEmpty2() {
        /*
         * Setup
         */
        Statement sTest =
this.createFromFileTest(FILE_NAME_2);
        Statement sRef = this.createFromFileRef(FILE_NAME_2);
        int lengthRef = sRef.lengthOfBlock();

        /*
         * The call
         */
        int lengthTest = sTest.lengthOfBlock();

        /*
         * Evaluation
         */
        assertEquals(lengthRef, lengthTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test assembleIf.
     */
    @Test
    public final void testAssembleIf2() {
        /*
         * Setup
         */
```

```java
        Statement blockTest =
this.createFromFileTest(FILE_NAME_2);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_2);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(1);
        Statement sRef = blockRef.removeFromBlock(1);
        Statement nestedTest = sourceTest.newInstance();
        Condition c = sourceTest.disassembleIf(nestedTest);
        Statement sTest = sourceTest.newInstance();

        /*
         * The call
         */
        sTest.assembleIf(c, nestedTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, nestedTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test disassembleIf.
     */
    @Test
    public final void testDisassembleIf2() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_2);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_2);
        Statement sTest = blockTest.removeFromBlock(1);
        Statement sRef = blockRef.removeFromBlock(1);
        Statement nestedTest = sTest.newInstance();
        Statement nestedRef = sRef.newInstance();
        Condition cRef = sRef.disassembleIf(nestedRef);

        /*
         * The call
         */
        Condition cTest = sTest.disassembleIf(nestedTest);

        /*
```

```
         * Evaluation
         */
        assertEquals(nestedRef, nestedTest);
        assertEquals(sRef, sTest);
        assertEquals(cRef, cTest);
    }

    /**
     * Test assembleIfElse.
     */
    @Test
    public final void testAssembleIfElse2() {
        /*
         * Setup
         */
        final int ifElsePos = 2;
        Statement blockTest =
this.createFromFileTest(FILE_NAME_2);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_2);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest =
blockTest.removeFromBlock(ifElsePos);
        Statement sRef = blockRef.removeFromBlock(ifElsePos);
        Statement thenBlockTest = sourceTest.newInstance();
        Statement elseBlockTest = sourceTest.newInstance();
        Condition cTest =
sourceTest.disassembleIfElse(thenBlockTest,
                elseBlockTest);
        Statement sTest = blockTest.newInstance();

        /*
         * The call
         */
        sTest.assembleIfElse(cTest, thenBlockTest,
elseBlockTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, thenBlockTest);
        assertEquals(emptyBlock, elseBlockTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test disassembleIfElse.
     */
    @Test
    public final void testDisassembleIfElse2() {
        /*
         * Setup
         */
        final int ifElsePos = 2;
        Statement blockTest =
this.createFromFileTest(FILE_NAME_2);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_2);
        Statement sTest =
blockTest.removeFromBlock(ifElsePos);
        Statement sRef = blockRef.removeFromBlock(ifElsePos);
        Statement thenBlockTest = sTest.newInstance();
        Statement elseBlockTest = sTest.newInstance();
        Statement thenBlockRef = sRef.newInstance();
        Statement elseBlockRef = sRef.newInstance();
        Condition cRef = sRef.disassembleIfElse(thenBlockRef,
elseBlockRef);

        /*
         * The call
         */
        Condition cTest =
sTest.disassembleIfElse(thenBlockTest, elseBlockTest);

        /*
         * Evaluation
         */
        assertEquals(cRef, cTest);
        assertEquals(thenBlockRef, thenBlockTest);
        assertEquals(elseBlockRef, elseBlockTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test assembleWhile.
     */
    @Test
    public final void testAssembleWhile2() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_2);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_2);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(1);
        Statement sourceRef = blockRef.removeFromBlock(1);
        Statement nestedTest = sourceTest.newInstance();
        Statement nestedRef = sourceRef.newInstance();
        Condition cTest =
sourceTest.disassembleIf(nestedTest);
        Condition cRef = sourceRef.disassembleIf(nestedRef);
        Statement sRef = sourceRef.newInstance();
        sRef.assembleWhile(cRef, nestedRef);
        Statement sTest = sourceTest.newInstance();

        /*
         * The call
         */
        sTest.assembleWhile(cTest, nestedTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, nestedTest);
        assertEquals(sRef, sTest);
    }

    /**
     * Test disassembleWhile.
     */
    @Test
    public final void testDisassembleWhile2() {
        /*
         * Setup
         */
        final int whilePos = 3;
        Statement blockTest =
this.createFromFileTest(FILE_NAME_2);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_2);
        Statement sTest =
blockTest.removeFromBlock(whilePos);
        Statement sRef = blockRef.removeFromBlock(whilePos);
        Statement nestedTest = sTest.newInstance();
        Statement nestedRef = sRef.newInstance();
        Condition cRef = sRef.disassembleWhile(nestedRef);

        /*
         * The call
         */
        Condition cTest = sTest.disassembleWhile(nestedTest);

        /*
         * Evaluation
         */
        assertEquals(nestedRef, nestedTest);
        assertEquals(sRef, sTest);
        assertEquals(cRef, cTest);
    }

    /**
     * Test assembleCall.
     */
    @Test
    public final void testAssembleCall2() {
        /*
         * Setup
         */
        Statement sRef = this.constructorRef().newInstance();
        Statement sTest =
this.constructorTest().newInstance();

        String name = "look-for-something";
        sRef.assembleCall(name);

        /*
         * The call
         */
        sTest.assembleCall(name);

        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
    }

    /**
     * Test disassembleCall.
     */
    @Test
    public final void testDisassembleCall2() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_2);
```

```java
        Statement blockRef = this.createFromFileRef(FILE_NAME_2);
        Statement sTest = blockTest.removeFromBlock(0);
        Statement sRef = blockRef.removeFromBlock(0);
        String nRef = sRef.disassembleCall();

        /*
         * The call
         */
        String nTest = sTest.disassembleCall();

        /*
         * Evaluation
         */
        assertEquals(sRef, sTest);
        assertEquals(nRef, nTest);
    }

    /**
     * Test assembleIfElse.
     */
    @Test
    public final void testAssembleIfElse3() {
        /*
         * Setup
         */
        final int ifElsePos = 2;
        Statement blockTest = this.createFromFileTest(FILE_NAME_2);
        Statement blockRef = this.createFromFileRef(FILE_NAME_2);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(ifElsePos);
        Statement sRef = blockRef.removeFromBlock(ifElsePos);
        Statement thenBlockTest = sourceTest.newInstance();
        Statement elseBlockTest = sourceTest.newInstance();
        Condition cTest = sourceTest.disassembleIfElse(thenBlockTest,
                elseBlockTest);
        Statement sTest = blockTest.newInstance();

        /*
         * The call
         */
        sTest.assembleIfElse(cTest, thenBlockTest, elseBlockTest);

        /*
         * Evaluation
         */
        assertEquals(emptyBlock, thenBlockTest);
        assertEquals(emptyBlock, elseBlockTest);
        assertEquals(sRef, sTest);
    }

}

```