THE OHIO STATE UNIVERSITY

PROJECT 7: IMPLEMENTATION OF PROGRAM AND STATEMENT KERNELS

Daniil Gofman

Ansh Pachauri

SW 2: Dev & Dsgn

Paolo Bucci

Yiyang Chen

Shivam Gupta

November 1, 2023

```java
import components.program.Program;
import components.program.Program1;

/**
 * Customized JUnit test fixture for {@code Program2}.
 */
public class Program2Test extends ProgramTest {

    @Override
    protected final Program constructorTest() {
        return new Program2();
    }

    @Override
    protected final Program constructorRef() {
        return new Program1();
    }

}

```

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

import org.junit.Test;

import components.map.Map;
import components.map.Map.Pair;
import components.program.Program;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.statement.Statement;

/**
 * JUnit test fixture for {@code Program}'s constructor and kernel methods.
 *
 * @author Ansh Pachauri
 * @author Daniil Gofman
 *
 */
public abstract class ProgramTest {

    /**
     * The name of a file containing a BL program.
     */
    private static final String FILE_NAME_1 = "data/program-sample.bl";
    /**
     * The name of a file containing a BL program.
     */
    private static final String FILE_NAME_2 = "data/program-sample2.bl";
    /**
     * The name of a file containing a BL program.
     */
    private static final String FILE_NAME_3 = "data/program-sample3.bl";

    /**
     * Invokes the {@code Program} constructor for the
```

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

    /**
     *
     * Creates and returns a {@code Program}, 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 to create
the program
     * @return the constructed program
     * @ensures createFromFile = [the program as parsed from
the file]
     */
    private Program createFromFileTest(String filename) {
        Program p = this.constructorTest();
        SimpleReader file = new SimpleReader1L(filename);
        p.parse(file);
        file.close();
        return p;
    }

    /**
     *
     * Creates and returns a {@code Program}, of the reference implementation
     * type, from the file with the given name.
     *
     * @param filename
     *            the name of the file to be parsed to create the program
     * @return the constructed program
     * @ensures createFromFile = [the program as parsed from the file]
     */
    private Program createFromFileRef(String filename) {
        Program p = this.constructorRef();
        SimpleReader file = new SimpleReader1L(filename);
        p.parse(file);
        file.close();
        return p;
    }

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

        /*
         * The call
         */
        Program pTest = this.constructorTest();

        /*
```

```
         * Evaluation
         */
        assertEquals(pRef, pTest);
    }

    /**
     * Test name.
     */
    @Test
    public final void testName() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_1);
        Program pRef = this.createFromFileRef(FILE_NAME_1);

        /*
         * The call
         */
        String result = pTest.name();

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals("Test", result);
    }

    /**
     * Test setName.
     */
    @Test
    public final void testSetName() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_1);
        Program pRef = this.createFromFileRef(FILE_NAME_1);
        String newName = "Replacement";
        pRef.setName(newName);

        /*
         * The call
         */
        pTest.setName(newName);

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
    }

    /**
     * Test newContext.
     */
    @Test
    public final void testNewContext() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_1);
        Program pRef = this.createFromFileRef(FILE_NAME_1);
        Map<String, Statement> cRef = pRef.newContext();

        /*
         * The call
         */
        Map<String, Statement> cTest = pTest.newContext();

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(cRef, cTest);
    }

    /**
     * Test swapContext.
```

```java
     */
    @Test
    public final void testSwapContext() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_1);
        Program pRef = this.createFromFileRef(FILE_NAME_1);
        Map<String, Statement> contextRef = pRef.newContext();
        Map<String, Statement> contextTest = pTest.newContext();
        String oneName = "one";
        pRef.swapContext(contextRef);
        Pair<String, Statement> oneRef = contextRef.remove(oneName);
        /* contextRef now has just "two" */
        pRef.swapContext(contextRef);
        /* pRef's context now has just "two" */
        contextRef.add(oneRef.key(), oneRef.value());
        /* contextRef now has just "one" */

        /* Make the reference call, replacing, in pRef, "one" with "two": */
        pRef.swapContext(contextRef);

        pTest.swapContext(contextTest);
        Pair<String, Statement> oneTest = contextTest.remove(oneName);
        /* contextTest now has just "two" */
        pTest.swapContext(contextTest);
        /* pTest's context now has just "two" */
        contextTest.add(oneTest.key(), oneTest.value());
        /* contextTest now has just "one" */

        /*
         * The call
         */
        pTest.swapContext(contextTest);

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(contextRef, contextTest);
    }

    /**
     * Test newBody.
     */
    @Test
    public final void testNewBody() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_1);
        Program pRef = this.createFromFileRef(FILE_NAME_1);
        Statement bRef = pRef.newBody();

        /*
         * The call
         */
        Statement bTest = pTest.newBody();

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(bRef, bTest);
    }

    /**
     * Test swapBody.
     */
    @Test
    public final void testSwapBody() {
        /*
         * Setup
         */
```

```java
        Program pTest = this.createFromFileTest(FILE_NAME_1);
        Program pRef = this.createFromFileRef(FILE_NAME_1);
        Statement bodyRef = pRef.newBody();
        Statement bodyTest = pTest.newBody();
        pRef.swapBody(bodyRef);
        Statement firstRef = bodyRef.removeFromBlock(0);
        /* bodyRef now lacks the first statement */
        pRef.swapBody(bodyRef);
        /* pRef's body now lacks the first statement */
        bodyRef.addToBlock(0, firstRef);
        /* bodyRef now has just the first statement */

        /* Make the reference call, replacing, in pRef,
remaining with first: */
        pRef.swapBody(bodyRef);

        pTest.swapBody(bodyTest);
        Statement firstTest = bodyTest.removeFromBlock(0);
        /* bodyTest now lacks the first statement */
        pTest.swapBody(bodyTest);
        /* pTest's body now lacks the first statement */
        bodyTest.addToBlock(0, firstTest);
        /* bodyTest now has just the first statement */

        /*
         * The call
         */
        pTest.swapBody(bodyTest);

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(bodyRef, bodyTest);
    }

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

```

```java
    /**
     * Test name.
     */
    @Test
    public final void testNameFileName2() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_2);
        Program pRef = this.createFromFileRef(FILE_NAME_2);

        /*
         * The call
         */
        String result = pTest.name();

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals("BugTerminator", result);
    }

    /**
     * Test setName.
     */
    @Test
    public final void testSetNameFileName2() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_2);
        Program pRef = this.createFromFileRef(FILE_NAME_2);
        String newName = "Replacement";
        pRef.setName(newName);

        /*
         * The call
         */
        pTest.setName(newName);

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
    }

    /**
     * Test newContext.
     */
    @Test
    public final void testNewContextFileName2() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_2);
        Program pRef = this.createFromFileRef(FILE_NAME_2);
        Map<String, Statement> cRef = pRef.newContext();

        /*
         * The call
         */
        Map<String, Statement> cTest = pTest.newContext();

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(cRef, cTest);
    }

    /**
     * Test swapContext.
     */
    @Test
    public final void testSwapContextFileName2() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_2);
        Program pRef = this.createFromFileRef(FILE_NAME_2);
        Map<String, Statement> contextRef = pRef.newContext();
        Map<String, Statement> contextTest =
pTest.newContext();
        String oneName = "kill";
        pRef.swapContext(contextRef);
        Pair<String, Statement> oneRef =
contextRef.remove(oneName);
        pRef.swapContext(contextRef);
        contextRef.add(oneRef.key(), oneRef.value());
        pRef.swapContext(contextRef);

        pTest.swapContext(contextTest);
        Pair<String, Statement> oneTest =
contextTest.remove(oneName);
        pTest.swapContext(contextTest);
        contextTest.add(oneTest.key(), oneTest.value());

        /*
         * The call
         */
        pTest.swapContext(contextTest);

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(contextRef, contextTest);
    }

    /**
     * Test newBody.
     */
    @Test
    public final void testNewBodyFileName2() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_2);
        Program pRef = this.createFromFileRef(FILE_NAME_2);
        Statement bRef = pRef.newBody();

        /*
         * The call
         */
        Statement bTest = pTest.newBody();

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(bRef, bTest);
    }

    /**
     * Test swapBody.
     */
    @Test
    public final void testSwapBodyFileName2() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_2);
        Program pRef = this.createFromFileRef(FILE_NAME_2);
        Statement bodyRef = pRef.newBody();
        Statement bodyTest = pTest.newBody();
        pRef.swapBody(bodyRef);
        Statement firstRef = bodyRef.removeFromBlock(0);
        /* bodyRef now lacks the first statement */
        pRef.swapBody(bodyRef);
        /* pRef's body now lacks the first statement */
        bodyRef.addToBlock(0, firstRef);
        /* bodyRef now has just the first statement */

        /* Make the reference call, replacing, in pRef,
   remaining with first: */
```

```
        pRef.swapBody(bodyRef);

        pTest.swapBody(bodyTest);
        Statement firstTest = bodyTest.removeFromBlock(0);
        /* bodyTest now lacks the first statement */
        pTest.swapBody(bodyTest);
        /* pTest's body now lacks the first statement */
        bodyTest.addToBlock(0, firstTest);
        /* bodyTest now has just the first statement */

        /*
         * The call
         */
        pTest.swapBody(bodyTest);

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(bodyRef, bodyTest);
    }

    /**
     * Test name.
     */
    @Test
    public final void testNameFileName3() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_3);
        Program pRef = this.createFromFileRef(FILE_NAME_3);

        /*
         * The call
         */
        String result = pTest.name();

        /*
```

```
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals("WeAreBorg", result);
    }

    /**
     * Test setName.
     */
    @Test
    public final void testSetNameFileName3() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_3);
        Program pRef = this.createFromFileRef(FILE_NAME_3);
        String newName = "Replacement";
        pRef.setName(newName);

        /*
         * The call
         */
        pTest.setName(newName);

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
    }

    /**
     * Test newContext.
     */
    @Test
    public final void testNewContextFileName3() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_3);
        Program pRef = this.createFromFileRef(FILE_NAME_3);
        Map<String, Statement> cRef = pRef.newContext();

        /*
         * The call
         */
        Map<String, Statement> cTest = pTest.newContext();

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(cRef, cTest);
    }

    /**
     * Test swapContext.
     */
    @Test
    public final void testSwapContextFileName3() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_3);
        Program pRef = this.createFromFileRef(FILE_NAME_3);
        Map<String, Statement> contextRef = pRef.newContext();
        Map<String, Statement> contextTest =
pTest.newContext();
        String oneName = "FindSpecies8472";
        pRef.swapContext(contextRef);
        Pair<String, Statement> oneRef =
contextRef.remove(oneName);
        pRef.swapContext(contextRef);
        contextRef.add(oneRef.key(), oneRef.value());

        pRef.swapContext(contextRef);

        pTest.swapContext(contextTest);
        Pair<String, Statement> oneTest =
contextTest.remove(oneName);
        pTest.swapContext(contextTest);
        contextTest.add(oneTest.key(), oneTest.value());

        /*
         * The call
         */
        pTest.swapContext(contextTest);

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(contextRef, contextTest);
    }

    /**
     * Test newBody.
     */
    @Test
    public final void testNewBodyFileName3() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_3);
        Program pRef = this.createFromFileRef(FILE_NAME_3);
        Statement bRef = pRef.newBody();

        /*
         * The call
         */
        Statement bTest = pTest.newBody();

        /*
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(bRef, bTest);
    }

    /**
     * Test swapBody.
     */
    @Test
    public final void testSwapBodyFileName3() {
        /*
         * Setup
         */
        Program pTest = this.createFromFileTest(FILE_NAME_3);
        Program pRef = this.createFromFileRef(FILE_NAME_3);
        Statement bodyRef = pRef.newBody();
        Statement bodyTest = pTest.newBody();
        pRef.swapBody(bodyRef);
        Statement firstRef = bodyRef.removeFromBlock(0);
        /* bodyRef now lacks the first statement */
        pRef.swapBody(bodyRef);
        /* pRef's body now lacks the first statement */
        bodyRef.addToBlock(0, firstRef);
        /* bodyRef now has just the first statement */

        /* Make the reference call, replacing, in pRef,
remaining with first: */
        pRef.swapBody(bodyRef);

        pTest.swapBody(bodyTest);
        Statement firstTest = bodyTest.removeFromBlock(0);
        /* bodyTest now lacks the first statement */
        pTest.swapBody(bodyTest);
        /* pTest's body now lacks the first statement */
        bodyTest.addToBlock(0, firstTest);
        /* bodyTest now has just the first statement */

        /*
         * The call
         */
        pTest.swapBody(bodyTest);

        /*
```

```
         * Evaluation
         */
        assertEquals(pRef, pTest);
        assertEquals(bodyRef, bodyTest);
    }

}

```

```java
import components.statement.Statement;
import components.statement.Statement1;

/**
 * Customized JUnit test fixture for {@code Statement2}.
 */
public class Statement2Test extends StatementTest {

    @Override
    protected final Statement constructorTest() {
        return new Statement2();
    }

    @Override
    protected final Statement constructorRef() {
        return new Statement1();
    }

}

```

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

import org.junit.Test;

import components.queue.Queue;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.statement.Statement;
import components.statement.StatementKernel.Condition;
import components.statement.StatementKernel.Kind;
import components.utilities.Tokenizer;

/**
 * JUnit test fixture for {@code Statement}'s constructor and
kernel methods.
 *
 * @author Ansh Pachauri
 * @author Daniil Gofman
 *
 */
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";

    /**
     * The name of a file containing a sequence of BL
statements.
     */
    private static final String FILE_NAME_2 = "data/statement-
sample2.bl";

    /**
     * The name of a file containing a sequence of BL
statements.
```

```java
     */
    private static final String FILE_NAME_3 = "data/statement-sample3.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 testKindWhile1() {
        /*
         * 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();

        /*
```

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

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

    /**
     * Test kind of a WHILE statement.
     */
    @Test
    public final void testKindWhile2() {
        /*
         * Setup
         */
        final int whilePos = 3;
        Statement sourceTest =
this.createFromFileTest(FILE_NAME_2);
        Statement sourceRef =
this.createFromFileRef(FILE_NAME_2);
        Statement sTest =
sourceTest.removeFromBlock(whilePos);
        Statement sRef = sourceRef.removeFromBlock(whilePos);
        Kind kRef = sRef.kind();

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

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

    /**
     * Test kind of a WHILE statement.
     */
    @Test
    public final void testKindWhile3() {
        /*
         * Setup
         */
        final int whilePos = 3;
        Statement sourceTest =
this.createFromFileTest(FILE_NAME_3);
        Statement sourceRef =
this.createFromFileRef(FILE_NAME_3);
        Statement sTest =
sourceTest.removeFromBlock(whilePos);
        Statement sRef = sourceRef.removeFromBlock(whilePos);
        Kind kRef = sRef.kind();

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

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

    /**
     * Test addToBlock at an interior position.
     */
    @Test
    public final void testAddToBlockInterior1() {
        /*
         * 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 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 addToBlock at an interior position.
     */
    @Test
    public final void testAddToBlockInterior3() {
        /*
         * Setup
         */
        Statement sTest =
this.createFromFileTest(FILE_NAME_3);
        Statement sRef = this.createFromFileRef(FILE_NAME_3);
        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 testRemoveFromBlockFrontLeavingNonEmpty1() {
        /*
         * 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 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 removeFromBlock at the front leaving a non-empty block behind.
     */
    @Test
    public final void testRemoveFromBlockFrontLeavingNonEmpty3() {
        /*
         * Setup
         */
        Statement sTest = this.createFromFileTest(FILE_NAME_3);
        Statement sRef = this.createFromFileRef(FILE_NAME_3);
        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 testLengthOfBlockNonEmpty1() {
        /*
         * 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 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();
```

```java

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

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

    /**
     * Test lengthOfBlock, greater than zero.
     */
    @Test
    public final void testLengthOfBlockNonEmpty3() {
        /*
         * Setup
         */
        Statement sTest =
this.createFromFileTest(FILE_NAME_3);
        Statement sRef = this.createFromFileRef(FILE_NAME_3);
        int lengthRef = sRef.lengthOfBlock();

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

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

    /**
```

```
     * Test assembleIf.
     */
    @Test
    public final void testAssembleIf1() {
        /*
         * 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 assembleIf.
     */
    @Test
    public final void testAssembleIf2() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_2);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_2);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(8);
        Statement sRef = blockRef.removeFromBlock(8);
        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 assembleIf.
     */
    @Test
    public final void testAssembleIf3() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_3);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_3);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(4);
        Statement sRef = blockRef.removeFromBlock(4);
        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 testDisassembleIf1() {
        /*
         * 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);

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

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

    /**
     * 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(0);
        Statement sRef = blockRef.removeFromBlock(0);
        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 disassembleIf.
     */
    @Test
    public final void testDisassembleIf3() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_3);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_3);
        Statement sTest = blockTest.removeFromBlock(4);
        Statement sRef = blockRef.removeFromBlock(4);
        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 testAssembleIfElse1() {
        /*
         * Setup
         */
        final int ifElsePos = 2;
        Statement blockTest =
```

```java
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 assembleIfElse.
     */
    @Test
    public final void testAssembleIfElse2() {
        /*
         * Setup
         */
        final int ifElsePos = 6;
        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 assembleIfElse.
     */
    @Test
    public final void testAssembleIfElse3() {
        /*
         * Setup
         */
        final int ifElsePos = 0;
        Statement blockTest =
this.createFromFileTest(FILE_NAME_3);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_3);
        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 testDisassembleIfElse1() {
        /*
         * 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 disassembleIfElse.
     */
    @Test
    public final void testDisassembleIfElse2() {
        /*
         * Setup
         */
        final int ifElsePos = 6;
        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 disassembleIfElse.
     */
    @Test
    public final void testDisassembleIfElse3() {
        /*
         * Setup
         */
        final int ifElsePos = 0;
        Statement blockTest =
this.createFromFileTest(FILE_NAME_3);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_3);
        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 testAssembleWhile1() {
        /*
         * 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();
```

```java
        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 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(8);
        Statement sourceRef = blockRef.removeFromBlock(8);
        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 assembleWhile.
     */
    @Test
    public final void testAssembleWhile3() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_3);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_3);
        Statement emptyBlock = blockRef.newInstance();
        Statement sourceTest = blockTest.removeFromBlock(4);
        Statement sourceRef = blockRef.removeFromBlock(4);
        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);
```

```java
        Statement sTest = sourceTest.newInstance();

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

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

    /**
     * Test disassembleWhile.
     */
    @Test
    public final void testDisassembleWhile1() {
        /*
         * 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 disassembleWhile.
     */
    @Test
    public final void testDisassembleWhile2() {
        /*
         * Setup
         */
        final int whilePos = 5;
        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);
    }

```

```java
    /**
     * Test disassembleWhile.
     */
    @Test
    public final void testDisassembleWhile3() {
        /*
         * Setup
         */
        final int whilePos = 2;
        Statement blockTest = this.createFromFileTest(FILE_NAME_3);
        Statement blockRef = this.createFromFileRef(FILE_NAME_3);
        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 testDisassembleCall1() {
        /*
         * 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();
```

```java

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

    /**
     * Test disassembleCall.
     */
    @Test
    public final void testDisassembleCall2() {
        /*
         * 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);
        String nRef = sRef.disassembleCall();

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

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

    /**
     * Test disassembleCall.
     */
    @Test
    public final void testDisassembleCall3() {
        /*
         * Setup
         */
        Statement blockTest =
this.createFromFileTest(FILE_NAME_3);
        Statement blockRef =
this.createFromFileRef(FILE_NAME_3);
        Statement sTest = blockTest.removeFromBlock(3);
        Statement sRef = blockRef.removeFromBlock(3);
        String nRef = sRef.disassembleCall();

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

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

}

```