CSE 2231 – Software 2: Software Development and Design

Professor: Rob LaTour

Project #7

Program and Statement Kernel Implementations / Implementation of Program and Statement Kernels

---

Date of Submission: March 24th, 2023

The Ohio State University

College of Engineering

Columbus, Ohio

```
import components.map.Map;
import components.map.Map1L;
import components.program.Program;
import components.program.ProgramSecondary;
import components.statement.Statement;
import components.statement.Statement1;
import components.statement.StatementKernel.Kind;
import components.utilities.Tokenizer;

/**
 * {@code Program} represented the obvious way with implementations of primary
 * methods.
 *
 * @convention [$this.name is an IDENTIFIER] and [$this.context is a CONTEXT]
 *             and [$this.body is a BLOCK statement]
 * @correspondence this = ($this.name, $this.context, $this.body)
 *
 * @author Danny Kan (kan.74@osu.edu)
 * @author Jatin Mamtani (mamtani.6@osu.edu)
 *
 */
public class Program2 extends ProgramSecondary {

    /*
     * Private members -----------------------------------------------------
     */

    /**
     * The program name.
     */
    private String name;
```

```
/**
 * The program context.
 */
private Map<String, Statement> context;

/**
 * The program body.
 */
private Statement body;

/**
 * Reports whether all the names of instructions in {@code c} are valid
 * IDENTIFIERs.
 *
 * @param c
 *            the context to check
 * @return true if all instruction names are identifiers; false otherwise
 * @ensures <pre>
 * allIdentifiers =
 *   [all the names of instructions in c are valid IDENTIFIERs]
 * </pre>
 */
private static boolean allIdentifiers(Map<String, Statement> c) {
    for (Map.Pair<String, Statement> pair : c) {
        if (!Tokenizer.isIdentifier(pair.key())) {
            return false;
        }
    }
    return true;
}

/**
```

```
 * Reports whether no instruction name in {@code c} is the name of a
 * primitive instruction.
 *
 * @param c
 *           the context to check
 * @return true if no instruction name is the name of a primitive
 *         instruction; false otherwise
 * @ensures <pre>
 * noPrimitiveInstructions =
 *   [no instruction name in c is the name of a primitive instruction]
 * </pre>
 */
private static boolean noPrimitiveInstructions(Map<String, Statement> c) {
    return !c.hasKey("move") && !c.hasKey("turnleft")
            && !c.hasKey("turnright") && !c.hasKey("infect")
            && !c.hasKey("skip");
}

/**
 * Reports whether all the bodies of instructions in {@code c} are BLOCK
 * statements.
 *
 * @param c
 *           the context to check
 * @return true if all instruction bodies are BLOCK statements; false
 *         otherwise
 * @ensures <pre>
 * allBlocks =
 *   [all the bodies of instructions in c are BLOCK statements]
 * </pre>
 */
private static boolean allBlocks(Map<String, Statement> c) {
    for (Map.Pair<String, Statement> pair : c) {
        if (pair.value().kind() != Kind.BLOCK) {
            return false;
        }
    }
    return true;
}

/**
 * Creator of initial representation.
 */
private void createNewRep() {
    this.name = "Unnamed";
    this.context = new Map1L<String, Statement>();
    this.body = new Statement1();
}

/*
 * Constructors ---------------------------------------------------------
 */

/**
 * No-argument constructor.
 */
public Program2() {
    this.createNewRep();
}

/*
 * Standard methods ----------------------------------------------------
 */
```

```
@Override
public final Program newInstance() {
    try {
        return this.getClass().getConstructor().newInstance();
    } catch (ReflectiveOperationException e) {
        throw new AssertionError(
                "Cannot construct object of type " + this.getClass());
    }
}

@Override
public final void clear() {
    this.createNewRep();
}

@Override
public final void transferFrom(Program source) {
    assert source != null : "Violation of: source is not null";
    assert source != this : "Violation of: source is not this";
    assert source instanceof Program2 : ""
            + "Violation of: source is of dynamic type Program2";
    /*
     * This cast cannot fail since the assert above would have stopped
     * execution in that case: source must be of dynamic type Program2.
     */
    Program2 localSource = (Program2) source;
    this.name = localSource.name;
    this.context = localSource.context;
    this.body = localSource.body;
    localSource.createNewRep();
}
```

```
/*
 * Kernel methods ---------------------------------------------------------
 */

@Override
public final void setName(String n) {
    assert n != null : "Violation of: n is not null";
    assert Tokenizer.isIdentifier(n) : ""
            + "Violation of: n is a valid IDENTIFIER";
    this.name = n;
}

@Override
public final String name() {
    return this.name;
}

@Override
public final Map<String, Statement> newContext() {
    return this.context.newInstance();
}

@Override
public final void swapContext(Map<String, Statement> c) {
    assert c != null : "Violation of: c is not null";
    assert c instanceof Map1L<?, ?> : "Violation of: c is a Map1L<?, ?>";
    assert allIdentifiers(
            c) : "Violation of: names in c are valid IDENTIFIERs";
    assert noPrimitiveInstructions(c) : ""
            + "Violation of: names in c do not match the names"
            + " of primitive instructions in the BL language";
    assert allBlocks(c) : "Violation of: bodies in c"
```

```
            + " are all BLOCK statements";
        Map<String, Statement> context = this.newContext();
        context.transferFrom(this.context);
        this.context.transferFrom(c);
        c.transferFrom(context);
    }

    @Override
    public final Statement newBody() {
        return this.body.newInstance();
    }

    @Override
    public final void swapBody(Statement b) {
        assert b != null : "Violation of: b is not null";
        assert b instanceof Statement1 : "Violation of: b is a Statement1";
        assert b.kind() == Kind.BLOCK : "Violation of: b is a BLOCK statement";
        Statement body = this.newBody();
        body.transferFrom(this.body);
        this.body.transferFrom(b);
        b.transferFrom(body);
    }

}
```