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.sequence.Sequence;
import components.statement.Statement;
import components.statement.StatementSecondary;
import components.tree.Tree;
import components.tree.Tree1;
import components.utilities.Tokenizer;

/**
 * {@code Statement} represented as a {@code Tree<StatementLabel>} with
 * implementations of primary methods.
 *
 * @convention [$this.rep is a valid representation of a Statement]
 * @correspondence this = $this.rep
 *
 * @author Danny Kan (kan.74@osu.edu)
 * @author Jatin Mamtani (mamtani.6@osu.edu)
 *
 */
public class Statement2 extends StatementSecondary {

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

    /**
     * Label class for the tree representation.
     */
    private static final class StatementLabel {

        /**
         * Statement kind.
         */
```

```
private Kind kind;

/**
 * IF/IF_ELSE/WHILE statement condition.
 */
private Condition condition;

/**
 * CALL instruction name.
 */
private String instruction;

/**
 * Constructor for BLOCK.
 *
 * @param k
 *            the kind of statement
 *
 * @requires k = BLOCK
 * @ensures this = (BLOCK, ?, ?)
 */
private StatementLabel(Kind k) {
    assert k == Kind.BLOCK : "Violation of: k = BLOCK";
    this.kind = k;
}

/**
 * Constructor for IF, IF_ELSE, WHILE.
 *
 * @param k
 *            the kind of statement
 * @param c
```

```
     *            the statement condition
     *
     * @requires k = IF or k = IF_ELSE or k = WHILE
     * @ensures this = (k, c, ?)
     */
    private StatementLabel(Kind k, Condition c) {
        assert k == Kind.IF || k == Kind.IF_ELSE || k == Kind.WHILE : ""
                + "Violation of: k = IF or k = IF_ELSE or k = WHILE";
        this.kind = k;
        this.condition = c;
    }

    /**
     * Constructor for CALL.
     *
     * @param k
     *            the kind of statement
     * @param i
     *            the instruction name
     *
     * @requires k = CALL and [i is an IDENTIFIER]
     * @ensures this = (CALL, ?, i)
     */
    private StatementLabel(Kind k, String i) {
        assert k == Kind.CALL : "Violation of: k = CALL";
        assert i != null : "Violation of: i is not null";
        assert Tokenizer
                .isIdentifier(i) : "Violation of: i is an IDENTIFIER";
        this.kind = k;
        this.instruction = i;
    }
```

```
    @Override
    public String toString() {
        String condition = "?", instruction = "?";
        if ((this.kind == Kind.IF) || (this.kind == Kind.IF_ELSE)
                || (this.kind == Kind.WHILE)) {
            condition = this.condition.toString();
        } else if (this.kind == Kind.CALL) {
            instruction = this.instruction;
        }
        return "(" + this.kind + "," + condition + "," + instruction + ")";
    }

}

/**
 * The tree representation field.
 */
private Tree<StatementLabel> rep;

/**
 * Creator of initial representation.
 */
private void createNewRep() {
    this.rep = new Tree1<StatementLabel>();
    StatementLabel label = new StatementLabel(Kind.BLOCK);
    this.rep.assemble(label, this.rep.newSequenceOfTree());
}

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

```
/**
 * No-argument constructor.
 */
public Statement2() {
    this.createNewRep();
}

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

@Override
public final Statement2 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(Statement source) {
    assert source != null : "Violation of: source is not null";
    assert source != this : "Violation of: source is not this";
    assert source instanceof Statement2 : ""
            + "Violation of: source is of dynamic type Statement2";
```

```
    /*
     * This cast cannot fail since the assert above would have stopped
     * execution in that case: source must be of dynamic type Statement2.
     */
    Statement2 localSource = (Statement2) source;
    this.rep = localSource.rep;
    localSource.createNewRep();
}

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

@Override
public final Kind kind() {
    return this.rep.root().kind;
}

@Override
public final void addToBlock(int pos, Statement s) {
    assert s != null : "Violation of: s is not null";
    assert s != this : "Violation of: s is not this";
    assert s instanceof Statement2 : "Violation of: s is a Statement2";
    assert this.kind() == Kind.BLOCK : ""
            + "Violation of: [this is a BLOCK statement]";
    assert 0 <= pos : "Violation of: 0 <= pos";
    assert pos <= this.lengthOfBlock() : ""
            + "Violation of: pos <= [length of this BLOCK]";
    assert s.kind() != Kind.BLOCK : "Violation of: [s is not a BLOCK statement]";
    Statement2 localS = (Statement2) s;
    Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
    StatementLabel label = this.rep.disassemble(children);
```

```
    children.add(pos, localS.rep);
    s.clear();
    this.rep.assemble(label, children);
}

@Override
public final Statement removeFromBlock(int pos) {
    assert 0 <= pos : "Violation of: 0 <= pos";
    assert pos < this.lengthOfBlock() : ""
            + "Violation of: pos < [length of this BLOCK]";
    assert this.kind() == Kind.BLOCK : ""
            + "Violation of: [this is a BLOCK statement]";
    Statement2 s = this.newInstance();
    Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
    StatementLabel label = this.rep.disassemble(children);
    s.rep = children.remove(pos);
    this.rep.assemble(label, children);
    return s;
}

@Override
public final int lengthOfBlock() {
    assert this.kind() == Kind.BLOCK : ""
            + "Violation of: [this is a BLOCK statement]";
    Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
    StatementLabel label = this.rep.disassemble(children);
    int length = children.length();
    this.rep.assemble(label, children);
    return length;
}

@Override
```

```
public final void assembleIf(Condition c, Statement s) {
    assert c != null : "Violation of: c is not null";
    assert s != null : "Violation of: s is not null";
    assert s != this : "Violation of: s is not this";
    assert s instanceof Statement2 : "Violation of: s is a Statement2";
    assert s.kind() == Kind.BLOCK : ""
            + "Violation of: [s is a BLOCK statement]";
    Statement2 localS = (Statement2) s;
    StatementLabel label = new StatementLabel(Kind.IF, c);
    Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
    children.add(0, localS.rep);
    this.rep.assemble(label, children);
    localS.createNewRep(); // clears s
}

@Override
public final Condition disassembleIf(Statement s) {
    assert s != null : "Violation of: s is not null";
    assert s != this : "Violation of: s is not this";
    assert s instanceof Statement2 : "Violation of: s is a Statement2";
    assert this.kind() == Kind.IF : ""
            + "Violation of: [this is an IF statement]";
    Statement2 localS = (Statement2) s;
    Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
    StatementLabel label = this.rep.disassemble(children);
    localS.rep = children.remove(0);
    this.createNewRep(); // clears this
    return label.condition;
}

@Override
public final void assembleIfElse(Condition c, Statement s1, Statement s2) {
```

```
    assert c != null : "Violation of: c is not null";
    assert s1 != null : "Violation of: s1 is not null";
    assert s2 != null : "Violation of: s2 is not null";
    assert s1 != this : "Violation of: s1 is not this";
    assert s2 != this : "Violation of: s2 is not this";
    assert s1 != s2 : "Violation of: s1 is not s2";
    assert s1 instanceof Statement2 : "Violation of: s1 is a Statement2";
    assert s2 instanceof Statement2 : "Violation of: s2 is a Statement2";
    assert s1
            .kind() == Kind.BLOCK : "Violation of: [s1 is a BLOCK statement]";
    assert s2
            .kind() == Kind.BLOCK : "Violation of: [s2 is a BLOCK statement]";
    Statement2 localS1 = (Statement2) s1;
    Statement2 localS2 = (Statement2) s2;
    StatementLabel label = new StatementLabel(Kind.IF_ELSE, c);
    Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
    children.add(0, localS1.rep);
    children.add(1, localS2.rep);
    this.rep.assemble(label, children);
    localS1.createNewRep(); // clears s1
    localS2.createNewRep(); // clears s2
}

@Override
public final Condition disassembleIfElse(Statement s1, Statement s2) {
    assert s1 != null : "Violation of: s1 is not null";
    assert s2 != null : "Violation of: s1 is not null";
    assert s1 != this : "Violation of: s1 is not this";
    assert s2 != this : "Violation of: s2 is not this";
    assert s1 != s2 : "Violation of: s1 is not s2";
    assert s1 instanceof Statement2 : "Violation of: s1 is a Statement2";
    assert s2 instanceof Statement2 : "Violation of: s2 is a Statement2";
```

```
    assert this.kind() == Kind.IF_ELSE : ""
            + "Violation of: [this is an IF_ELSE statement]";
    Statement2 localS1 = (Statement2) s1;
    Statement2 localS2 = (Statement2) s2;
    Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
    StatementLabel label = this.rep.disassemble(children);
    localS1.rep = children.remove(0);
    localS2.rep = children.remove(0);
    this.createNewRep(); // clears this
    return label.condition;
}

@Override
public final void assembleWhile(Condition c, Statement s) {
    assert c != null : "Violation of: c is not null";
    assert s != null : "Violation of: s is not null";
    assert s != this : "Violation of: s is not this";
    assert s instanceof Statement2 : "Violation of: s is a Statement2";
    assert s.kind() == Kind.BLOCK : "Violation of: [s is a BLOCK statement]";
    Statement2 localS = (Statement2) s;
    StatementLabel label = new StatementLabel(Kind.WHILE, c);
    Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
    children.add(0, localS.rep);
    this.rep.assemble(label, children);
    localS.createNewRep(); // clears s
}

@Override
public final Condition disassembleWhile(Statement s) {
    assert s != null : "Violation of: s is not null";
    assert s != this : "Violation of: s is not this";
    assert s instanceof Statement2 : "Violation of: s is a Statement2";
```

```
        assert this.kind() == Kind.WHILE : ""
                + "Violation of: [this is a WHILE statement]";
        Statement2 localS = (Statement2) s;
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
        StatementLabel label = this.rep.disassemble(children);
        localS.rep = children.remove(0);
        this.createNewRep(); // clears this
        return label.condition;
    }

    @Override
    public final void assembleCall(String inst) {
        assert inst != null : "Violation of: inst is not null";
        assert Tokenizer.isIdentifier(inst) : ""
                + "Violation of: inst is a valid IDENTIFIER";
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
        StatementLabel label = new StatementLabel(Kind.CALL, inst);
        this.rep.assemble(label, children);
    }

    @Override
    public final String disassembleCall() {
        assert this.kind() == Kind.CALL : ""
                + "Violation of: [this is a CALL statement]";
        Sequence<Tree<StatementLabel>> children = this.rep.newSequenceOfTree();
        StatementLabel label = this.rep.disassemble(children);
        this.createNewRep(); // clears this
        return label.instruction;
    }

}
```