JBooleanBinaryExpression.java

```java
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * Most binary expressions that return booleans can be recognized by their
 * syntax. We take advantage of this to define a common codegen(), which relies
 * on the short-circuiting code generation for control and puts either a 1 or a
 * 0 onto the stack.
 */

abstract class JBooleanBinaryExpression extends JBinaryExpression {

    /**
     * Construct an AST node for a boolean binary expression.
     * 
     * @param line
     *            line in which the boolean binary expression occurs in the
     *            source file.
     * @param operator
     *            the boolean binary operator.
     * @param lhs
     *            lhs operand.
     * @param rhs
     *            rhs operand.
     */

    protected JBooleanBinaryExpression(int line, String operator,
            JExpression lhs, JExpression rhs) {
        super(line, operator, lhs, rhs);
    }

    /**
     * Generate code for the case where we actually want a boolean value (true
     * or false) computed onto the stack, eg for assignment to a boolean
     * variable.
     * 
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output) {
        String elseLabel = output.createLabel();
        String endIfLabel = output.createLabel();
        this.codegen(output, elseLabel, false);
        output.addNoArgInstruction(ICONST_1); // true
        output.addBranchInstruction(GOTO, endIfLabel);
        output.addLabel(elseLabel);
        output.addNoArgInstruction(ICONST_0); // false
        output.addLabel(endIfLabel);
    }

}

/**
 * The AST node for an equality (==) expression. Implements short-circuiting
 * branching.
 */

class JEqualOp extends JBooleanBinaryExpression {

    /**
     * Construct an AST node for an equality expression.
     *
     * @param line
     *            line number in which the equality expression occurs in the
     *            source file.
     * @param lhs
     *            lhs operand.
     * @param rhs
     *            rhs operand.
     */

    public JEqualOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "==", lhs, rhs);
    }

    /**
     * Analyzing an equality expression means analyzing its operands and
     * checking that the types match.
     *
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JExpression analyze(Context context) {
        lhs = (JExpression) lhs.analyze(context);
        rhs = (JExpression) rhs.analyze(context);
        lhs.type().mustMatchExpected(line(), rhs.type());
        type = Type.BOOLEAN;
        return this;
    }

    /**
     * Branching code generation for == operation.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     * @param targetLabel
     *            target for generated branch instruction.
     * @param onTrue
     *            should we branch on true?
     */

    public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
        lhs.codegen(output);
        rhs.codegen(output);
        if (lhs.type().isReference()) {
            output.addBranchInstruction(onTrue ? IF_ACMPEQ : IF_ACMPNE,
                    targetLabel);
        } else {
            output.addBranchInstruction(onTrue ? IF_ICMPEQ : IF_ICMPNE,
                    targetLabel);
        }
    }

}

/**
 * The AST node for a logical AND (&&) expression. Implements short-circuiting
 * branching.
 */

class JLogicalAndOp extends JBooleanBinaryExpression {

    /**
     * Construct an AST node for a logical AND expression given its line number,
     * and lhs and rhs operands.
     *
     * @param line
     *             line in which the logical AND expression occurs in the source
     *             file.
     * @param lhs
     *             lhs operand.
     * @param rhs
     *             rhs operand.
     */

    public JLogicalAndOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "&&", lhs, rhs);
    }

    /**
     * Analyzing a logical AND expression involves analyzing its operands and
     * insuring they are boolean; the result type is of course boolean.
     *
     * @param context
     *             context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JExpression analyze(Context context) {
        lhs = (JExpression) lhs.analyze(context);
        rhs = (JExpression) rhs.analyze(context);
        lhs.type().mustMatchExpected(line(), Type.BOOLEAN);
        rhs.type().mustMatchExpected(line(), Type.BOOLEAN);
        type = Type.BOOLEAN;
        return this;
    }

    /**
     * The semantics of j-- require that we implement short-circuiting branching
     * in implementing the logical AND.
     *
     * @param output
     *             the code emitter (basically an abstraction for producing the
     *             .class file).
     * @param targetLabel
     *             target for generated branch instruction.
     * @param onTrue
     *             should we branch on true?
     */

    public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
        if (onTrue) {
            String falseLabel = output.createLabel();
            lhs.codegen(output, falseLabel, false);
            rhs.codegen(output, targetLabel, true);
            output.addLabel(falseLabel);
        } else {
            lhs.codegen(output, targetLabel, false);
            rhs.codegen(output, targetLabel, false);
        }
    }

}
```