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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a unary expression. A unary expression has a single operand.
 */

abstract class JUnaryExpression extends JExpression {

    /** The operator. */
    private String operator;

    /** The operand. */
    protected JExpression arg;

    /**
     * Construct an AST node for a unary expression given its line number, the
     * unary operator, and the operand.
     *
     * @param line
     *            line in which the unary expression occurs in the source file.
     * @param operator
     *            the unary operator.
     * @param arg
     *            the operand.
     */

    protected JUnaryExpression(int line, String operator, JExpression arg) {
        super(line);
        this.operator = operator;
        this.arg = arg;
    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.printf("<JUnaryExpression line=\"%d\" type=\"%s\" "
                + "operator=\"%s\">\n", line(), ((type == null) ? "" : type
                .toString()), Util.escapeSpecialXMLChars(operator));
        p.indentRight();
        p.printf("<Operand>\n");
        p.indentRight();
        arg.writeToStdOut(p);
        p.indentLeft();
        p.printf("</Operand>\n");
        p.indentLeft();
        p.printf("</JUnaryExpression>\n");
    }

}

/**
 * The AST node for a unary negation (-) expression.
 */

class JNegateOp extends JUnaryExpression {

    /**
     * Construct an AST node for a negation expression given its line number,
     * and the operand.
     *
```

```
67       * @param line
68       *             line in which the negation expression occurs in the source
69       *             file.
70       * @param arg
71       *             the operand.
72       */
73
74      public JNegateOp(int line, JExpression arg) {
75          super(line, "-", arg);
76      }
77
78      /**
79       * Analyzing the negation operation involves analyzing its operand, checking
80       * its type and determining the result type.
81       *
82       * @param context
83       *             context in which names are resolved.
84       * @return the analyzed (and possibly rewritten) AST subtree.
85       */
86
87      public JExpression analyze(Context context) {
88          arg = arg.analyze(context);
89          arg.type().mustMatchExpected(line(), Type.INT);
90          type = Type.INT;
91          return this;
92      }
93
94      /**
95       * Generating code for the negation operation involves generating code for
96       * the operand, and then the negation instruction.
97       *
98       * @param output
99       *             the code emitter (basically an abstraction for producing the
100      *             class file)
101      */
102
103     public void codegen(CLEmitter output) {
104         arg.codegen(output);
105         output.addNoArgInstruction(INEG);
106     }
107
108 }
109
110 /**
111  * The AST node for a logical NOT (!) expression.
112  */
113
114 class JLogicalNotOp extends JUnaryExpression {
115
116     /**
117      * Construct an AST for a logical NOT expression given its line number, and
118      * the operand.
119      *
120      * @param line
121      *             line in which the logical NOT expression occurs in the source
122      *             file.
123      * @param arg
124      *             the operand.
125      */
126
127     public JLogicalNotOp(int line, JExpression arg) {
128         super(line, "!", arg);
129     }
130
131     /**
132      * Analyzing a logical NOT operation means analyzing its operand, insuring
133      * it's a boolean, and setting the result to boolean.
134      *
135      * @param context
```

```
     *            context in which names are resolved.
     */

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

    /**
     * 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 code generation necessary for branching simply flips the condition on
     * which we branch.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
        arg.codegen(output, targetLabel, !onTrue);
    }

}

/**
 * The AST node for an expr--.
 */

class JPostDecrementOp extends JUnaryExpression {

    /**
     * Construct an AST node for an expr-- expression given its line number, and
     * the operand.
     *
     * @param line
     *            line in which the expression occurs in the source file.
     * @param arg
     *            the operand.
     */

    public JPostDecrementOp(int line, JExpression arg) {
        super(line, "post--", arg);
    }

    /**
     * Analyze the operand as a lhs (since there is a side effect), check types
     * and determine the type of the result.
```

```
205      *
206      * @param context
207      *            context in which names are resolved.
208      * @return the analyzed (and possibly rewritten) AST subtree.
209      */
210
211     public JExpression analyze(Context context) {
212         if (!(arg instanceof JLhs)) {
213             JAST.compilationUnit.reportSemanticError(line,
214                     "Operand to expr-- must have an LValue.");
215             type = Type.ANY;
216         } else {
217             arg = (JExpression) arg.analyze(context);
218             arg.type().mustMatchExpected(line(), Type.INT);
219             type = Type.INT;
220         }
221         return this;
222     }
223
224     /**
225      * In generating code for a post-decrement operation, we treat simple
226      * variable (JVariable) operands specially since the JVM has an increment
227      * instruction. Otherwise, we rely on the JLhs code generation support for
228      * generating the proper code. Notice that we distinguish between
229      * expressions that are statement expressions and those that are not; we
230      * insure the proper value (before the decrement) is left atop the stack in
231      * the latter case.
232      *
233      * @param output
234      *            the code emitter (basically an abstraction for producing the
235      *            .class file).
236      */
237
238     public void codegen(CLEmitter output) {
239         if (arg instanceof JVariable) {
240             // A local variable; otherwise analyze() would
241             // have replaced it with an explicit field selection.
242             int offset = ((LocalVariableDefn) ((JVariable) arg).iDefn())
243                     .offset();
244             if (!isStatementExpression) {
245                 // Loading its original rvalue
246                 arg.codegen(output);
247             }
248             output.addIINCInstruction(offset, -1);
249         } else {
250             ((JLhs) arg).codegenLoadLhsLvalue(output);
251             ((JLhs) arg).codegenLoadLhsRvalue(output);
252             if (!isStatementExpression) {
253                 // Loading its original rvalue
254                 ((JLhs) arg).codegenDuplicateRvalue(output);
255             }
256             output.addNoArgInstruction(ICONST_1);
257             output.addNoArgInstruction(ISUB);
258             ((JLhs) arg).codegenStore(output);
259         }
260     }
261
262 }
263
264 /**
265  * The AST node for a ++expr expression.
266  */
267
268 class JPreIncrementOp extends JUnaryExpression {
269
270     /**
271      * Construct an AST node for a ++expr given its line number, and the
272      * operand.
273      *
```

```
     * @param line
     *            line in which the expression occurs in the source file.
     * @param arg
     *            the operand.
     */

    public JPreIncrementOp(int line, JExpression arg) {
        super(line, "++pre", arg);
    }

    /**
     * Analyze the operand as a lhs (since there is a side effect), check types
     * and determine the type of the result.
     *
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JExpression analyze(Context context) {
        if (!(arg instanceof JLhs)) {
            JAST.compilationUnit.reportSemanticError(line,
                    "Operand to ++expr must have an LValue.");
            type = Type.ANY;
        } else {
            arg = (JExpression) arg.analyze(context);
            arg.type().mustMatchExpected(line(), Type.INT);
            type = Type.INT;
        }
        return this;
    }

    /**
     * In generating code for a pre-increment operation, we treat simple
     * variable (JVariable) operands specially, since the JVM has an increment
     * instruction. Otherwise, we rely on the JLhs code generation support for
     * generating the proper code. Notice that we distinguish between
     * expressions that are statement expressions and those that are not; we
     * insure the proper value (after the increment) is left atop the stack in
     * the latter case.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output) {
        if (arg instanceof JVariable) {
            // A local variable; otherwise analyze() would
            // have replaced it with an explicit field selection.
            int offset = ((LocalVariableDefn) ((JVariable) arg).iDefn())
                    .offset();
            output.addIINCInstruction(offset, 1);
            if (!isStatementExpression) {
                // Loading its original rvalue
                arg.codegen(output);
            }
        } else {
            ((JLhs) arg).codegenLoadLhsLvalue(output);
            ((JLhs) arg).codegenLoadLhsRvalue(output);
            output.addNoArgInstruction(ICONST_1);
            output.addNoArgInstruction(IADD);
            if (!isStatementExpression) {
                // Loading its original rvalue
                ((JLhs) arg).codegenDuplicateRvalue(output);
            }
            ((JLhs) arg).codegenStore(output);
        }
    }
```

```
343
344 }
345
```