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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a binary expression. A binary expression has an operator and
 * two operands: a lhs and a rhs.
 */

abstract class JBinaryExpression extends JExpression {

    /** The binary operator. */
    protected String operator;

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

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

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

    protected JBinaryExpression(int line, String operator, JExpression lhs,
            JExpression rhs) {
        super(line);
        this.operator = operator;
        this.lhs = lhs;
        this.rhs = rhs;
    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.printf("<JBinaryExpression line=\"%d\" type=\"%s\" "
                + "operator=\"%s\">\n", line(), ((type == null) ? "" : type
                .toString()), Util.escapeSpecialXMLChars(operator));
        p.indentRight();
        p.printf("<Lhs>\n");
        p.indentRight();
        lhs.writeToStdOut(p);
        p.indentLeft();
        p.printf("</Lhs>\n");
        p.printf("<Rhs>\n");
        p.indentRight();
        rhs.writeToStdOut(p);
        p.indentLeft();
        p.printf("</Rhs>\n");
        p.indentLeft();
        p.printf("</JBinaryExpression>\n");
    }
```

```
67
68  }
69
70  /**
71   * The AST node for a plus (+) expression. In j--, as in Java, + is overloaded
72   * to denote addition for numbers and concatenation for Strings.
73   */
74
75  class JPlusOp extends JBinaryExpression {
76
77      /**
78       * Construct an AST node for an addition expression given its line number,
79       * and the lhs and rhs operands.
80       *
81       * @param line
82       *             line in which the addition expression occurs in the source
83       *             file.
84       * @param lhs
85       *             the lhs operand.
86       * @param rhs
87       *             the rhs operand.
88       */
89
90      public JPlusOp(int line, JExpression lhs, JExpression rhs) {
91          super(line, "+", lhs, rhs);
92      }
93
94      /**
95       * Analysis involves first analyzing the operands. If this is a string
96       * concatenation, we rewrite the subtree to make that explicit (and analyze
97       * that). Otherwise we check the types of the addition operands and compute
98       * the result type.
99       *
100      * @param context
101      *             context in which names are resolved.
102      * @return the analyzed (and possibly rewritten) AST subtree.
103      */
104
105     public JExpression analyze(Context context) {
106         lhs = (JExpression) lhs.analyze(context);
107         rhs = (JExpression) rhs.analyze(context);
108         if (lhs.type() == Type.STRING || rhs.type() == Type.STRING) {
109             return (new JStringConcatenationOp(line, lhs, rhs))
110                     .analyze(context);
111         } else if (lhs.type() == Type.INT && rhs.type() == Type.INT) {
112             type = Type.INT;
113         } else {
114             type = Type.ANY;
115             JAST.compilationUnit.reportSemanticError(line(),
116                     "Invalid operand types for +");
117         }
118         return this;
119     }
120
121     /**
122      * Any string concatenation has been rewritten as a JStringConcatenationOp
123      * (in analyze()), so code generation here involves simply generating code
124      * for loading the operands onto the stack and then generating the
125      * appropriate add instruction.
126      *
127      * @param output
128      *             the code emitter (basically an abstraction for producing the
129      *             .class file).
130      */
131
132     public void codegen(CLEmitter output) {
133         if (type == Type.INT) {
134             lhs.codegen(output);
135             rhs.codegen(output);
```

```
136             output.addNoArgInstruction(IADD);
137         }
138     }
139 
140 }
141 
142 /**
143  * The AST node for a subtraction (-) expression.
144  */
145 
146 class JSubtractOp extends JBinaryExpression {
147 
148     /**
149      * Construct an AST node for a subtraction expression given its line number,
150      * and lhs and rhs operands.
151      *
152      * @param line
153      *             line in which the subtraction expression occurs in the source
154      *             file.
155      * @param lhs
156      *             the lhs operand.
157      * @param rhs
158      *             the rhs operand.
159      */
160 
161     public JSubtractOp(int line, JExpression lhs, JExpression rhs) {
162         super(line, "-", lhs, rhs);
163     }
164 
165     /**
166      * Analyzing the - operation involves analyzing its operands, checking
167      * types, and determining the result type.
168      *
169      * @param context
170      *             context in which names are resolved.
171      * @return the analyzed (and possibly rewritten) AST subtree.
172      */
173 
174     public JExpression analyze(Context context) {
175         lhs = (JExpression) lhs.analyze(context);
176         rhs = (JExpression) rhs.analyze(context);
177         lhs.type().mustMatchExpected(line(), Type.INT);
178         rhs.type().mustMatchExpected(line(), Type.INT);
179         type = Type.INT;
180         return this;
181     }
182 
183     /**
184      * Generating code for the - operation involves generating code for the two
185      * operands, and then the subtraction instruction.
186      *
187      * @param output
188      *             the code emitter (basically an abstraction for producing the
189      *             .class file).
190      */
191 
192     public void codegen(CLEmitter output) {
193         lhs.codegen(output);
194         rhs.codegen(output);
195         output.addNoArgInstruction(ISUB);
196     }
197 
198 }
199 
200 /**
201  * The AST node for a multiplication (*) expression.
202  */
203 
204 class JMultiplyOp extends JBinaryExpression {
```

```
205
206     /**
207      * Construct an AST for a multiplication expression given its line number,
208      * and the lhs and rhs operands.
209      *
210      * @param line
211      *            line in which the multiplication expression occurs in the
212      *            source file.
213      * @param lhs
214      *            the lhs operand.
215      * @param rhs
216      *            the rhs operand.
217      */
218
219     public JMultiplyOp(int line, JExpression lhs, JExpression rhs) {
220         super(line, "*", lhs, rhs);
221     }
222
223     /**
224      * Analyzing the * operation involves analyzing its operands, checking
225      * types, and determining the result type.
226      *
227      * @param context
228      *            context in which names are resolved.
229      * @return the analyzed (and possibly rewritten) AST subtree.
230      */
231
232     public JExpression analyze(Context context) {
233         lhs = (JExpression) lhs.analyze(context);
234         rhs = (JExpression) rhs.analyze(context);
235         lhs.type().mustMatchExpected(line(), Type.INT);
236         rhs.type().mustMatchExpected(line(), Type.INT);
237         type = Type.INT;
238         return this;
239     }
240
241     /**
242      * Generating code for the * operation involves generating code for the two
243      * operands, and then the multiplication instruction.
244      *
245      * @param output
246      *            the code emitter (basically an abstraction for producing the
247      *            .class file).
248      */
249
250     public void codegen(CLEmitter output) {
251         lhs.codegen(output);
252         rhs.codegen(output);
253         output.addNoArgInstruction(IMUL);
254     }
255
256 }
257
```