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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for an assignment statement. This is an abtract class into which
 * we factor behavior common to all assignment operations.
 */

abstract class JAssignment extends JBinaryExpression {

    /**
     * Construct an AST node for an assignment operation.
     *
     * @param line
     *            line in which the assignment operation occurs in the source
     *            file.
     * @param operator
     *            the actual assignment operator.
     * @param lhs
     *            the lhs operand.
     * @param rhs
     *            the rhs operand.
     */

    public JAssignment(int line, String operator, JExpression lhs,
            JExpression rhs) {
        super(line, operator, lhs, rhs);
    }

}

/**
 * The AST node for an assignment (=) expression. The = operator has two
 * operands: a lhs and a rhs.
 */

class JAssignOp extends JAssignment {

    /**
     * Construct the AST node for an assignment (=) expression given the lhs and
     * rhs operands.
     *
     * @param line
     *            line in which the assignment expression occurs in the source
     *            file.
     * @param lhs
     *            lhs operand.
     * @param rhs
     *            rhs operand.
     */

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

    /**
     * Analyze the lhs and rhs, checking that types match, and set the result
     * type.
     *
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */
```

```
67
68      public JExpression analyze(Context context) {
69          if (!(lhs instanceof JLhs)) {
70              JAST.compilationUnit.reportSemanticError(line(),
71                      "Illegal lhs for assignment");
72          } else {
73              lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
74          }
75          rhs = (JExpression) rhs.analyze(context);
76          rhs.type().mustMatchExpected(line(), lhs.type());
77          type = rhs.type();
78          if (lhs instanceof JVariable) {
79              IDefn defn = ((JVariable) lhs).iDefn();
80              if (defn != null) {
81                  // Local variable; consider it to be initialized now.
82                  ((LocalVariableDefn) defn).initialize();
83              }
84          }
85          return this;
86      }
87
88      /**
89       * Code generation for an assignment involves, generating code for loading
90       * any necessary Lvalue onto the stack, for loading the Rvalue, for (unless
91       * a statement) copying the Rvalue to its proper place on the stack, and for
92       * doing the store.
93       *
94       * @param output
95       *             the code emitter (basically an abstraction for producing the
96       *             class file)
97       */
98
99      public void codegen(CLEmitter output) {
100         ((JLhs) lhs).codegenLoadLhsLvalue(output);
101         rhs.codegen(output);
102         if (!isStatementExpression) {
103             // Generate code to leave the Rvalue atop stack
104             ((JLhs) lhs).codegenDuplicateRvalue(output);
105         }
106         ((JLhs) lhs).codegenStore(output);
107     }
108
109 }
110
111 /**
112  * The AST node for a += expression. A += expression has two operands: a lhs and
113  * a rhs.
114  */
115
116 class JPlusAssignOp extends JAssignment {
117
118     /**
119      * Construct the AST node for a += expression given its lhs and rhs
120      * operands.
121      *
122      * @param line
123      *             line in which the assignment expression occurs in the source
124      *             file.
125      * @param lhs
126      *             the lhs operand.
127      * @param rhs
128      *             the rhs operand.
129      */
130
131     public JPlusAssignOp(int line, JExpression lhs, JExpression rhs) {
132         super(line, "+=", lhs, rhs);
133     }
134
135     /**
```

```
136      * Analyze the lhs and rhs, rewrite rhs as lhs + rhs (string concatenation)
137      * if lhs is a String, and set the result type.
138      *
139      * @param context
140      *            context in which names are resolved.
141      * @return the analyzed (and possibly rewritten) AST subtree.
142      */
143
144     public JExpression analyze(Context context) {
145         if (!(lhs instanceof JLhs)) {
146             JAST.compilationUnit.reportSemanticError(line(),
147                     "Illegal lhs for assignment");
148         return this;
149         } else {
150             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
151         }
152         rhs = (JExpression) rhs.analyze(context);
153         if (lhs.type().equals(Type.INT)) {
154             rhs.type().mustMatchExpected(line(), Type.INT);
155             type = Type.INT;
156         } else if (lhs.type().equals(Type.STRING)) {
157             rhs = (new JStringConcatenationOp(line, lhs, rhs)).analyze(context);
158             type = Type.STRING;
159         } else {
160             JAST.compilationUnit.reportSemanticError(line(),
161                     "Invalid lhs type for +=: " + lhs.type());
162         }
163         return this;
164     }
165
166     /**
167      * Code generation for += involves, generating code for loading any
168      * necessary l-value onto the stack, for (unless a string concatenation)
169      * loading the r-value, for (unless a statement) copying the r-value to its
170      * proper place on the stack, and for doing the store.
171      *
172      * @param output
173      *            the code emitter (basically an abstraction for producing the
174      *            .class file)
175      */
176
177     public void codegen(CLEmitter output) {
178         ((JLhs) lhs).codegenLoadLhsLvalue(output);
179         if (lhs.type().equals(Type.STRING)) {
180             rhs.codegen(output);
181         } else {
182             ((JLhs) lhs).codegenLoadLhsRvalue(output);
183             rhs.codegen(output);
184             output.addNoArgInstruction(IADD);
185         }
186         if (!isStatementExpression) {
187             // Generate code to leave the r-value atop stack
188             ((JLhs) lhs).codegenDuplicateRvalue(output);
189         }
190         ((JLhs) lhs).codegenStore(output);
191     }
192
193 }
194
```