```
1  // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import static jminusminus.CLConstants.*;
6
7  /**
8   * The AST node for a comparison expression. This class captures common aspects
9   * of comparison operations.
10  */
11
12 abstract class JComparison extends JBooleanBinaryExpression {
13
14     /**
15      * Create an AST node for a comparison expression.
16      *
17      * @param line
18      *             line in which the expression occurs in the source file.
19      * @param operator
20      *             the comparison operator.
21      * @param lhs
22      *             the lhs operand.
23      * @param rhs
24      *             the rhs operand.
25      */
26
27     protected JComparison(int line, String operator, JExpression lhs,
28             JExpression rhs) {
29         super(line, operator, lhs, rhs);
30     }
31
32     /**
33      * The analysis of a comparison operation consists of analyzing its two
34      * operands, and making sure they both have the same numeric type.
35      *
36      * @param context
37      *             context in which names are resolved.
38      * @return the analyzed (and possibly rewritten) AST subtree.
39      */
40
41     public JExpression analyze(Context context) {
42         lhs = (JExpression) lhs.analyze(context);
43         rhs = (JExpression) rhs.analyze(context);
44         lhs.type().mustMatchExpected(line(), Type.INT);
45         rhs.type().mustMatchExpected(line(), lhs.type());
46         type = Type.BOOLEAN;
47         return this;
48     }
49
50 }
51
52 /**
53  * The AST node for a greater-than (>) expression. Implements short-circuiting
54  * branching.
55  */
56
57 class JGreaterThanOp extends JComparison {
58
59     /**
60      * Construct an AST node for a greater-than expression given its line
61      * number, and the lhs and rhs operands.
62      *
63      * @param line
64      *             line in which the greater-than expression occurs in the source
65      *             file.
66      * @param lhs
```

```
67          *            lhs operand.
68          * @param rhs
69          *            rhs operand.
70          */
71
72         public JGreaterThanOp(int line, JExpression lhs, JExpression rhs) {
73             super(line, ">", lhs, rhs);
74         }
75
76         /**
77          * Branching code generation for > operation.
78          *
79          * @param output
80          *            the code emitter (basically an abstraction for producing the
81          *            .class file).
82          * @param targetLabel
83          *            target for generated branch instruction.
84          * @param onTrue
85          *            should we branch on true?
86          */
87
88         public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
89             lhs.codegen(output);
90             rhs.codegen(output);
91             output
92                     .addBranchInstruction(onTrue ? IF_ICMPGT : IF_ICMPLE,
93                             targetLabel);
94         }
95
96     }
97
98     /**
99      * The AST node for a less-than-or-equal-to (<=) expression. Implements
100     * short-circuiting branching.
101     */
102
103    class JLessEqualOp extends JComparison {
104
105        /**
106         * Construct an AST node for a less-than-or-equal-to expression given its
107         * line number, and the lhs and rhs operands.
108         *
109         * @param line
110         *            line in which the less-than-or-equal-to expression occurs in
111         *            the source file.
112         * @param lhs
113         *            lhs operand.
114         * @param rhs
115         *            rhs operand.
116         */
117
118        public JLessEqualOp(int line, JExpression lhs, JExpression rhs) {
119            super(line, "<=", lhs, rhs);
120        }
121
122        /**
123         * Branching code generation for <= operation.
124         *
125         * @param output
126         *            the code emitter (basically an abstraction for producing the
127         *            .class file).
128         * @param targetLabel
129         *            target for generated branch instruction.
130         * @param onTrue
131         *            should we branch on true?
132         */
133
134        public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
135            lhs.codegen(output);
```

```
136         rhs.codegen(output);
137         output
138                 .addBranchInstruction(onTrue ? IF_ICMPLE : IF_ICMPGT,
139                         targetLabel);
140     }
141
142 }
143
```