

## JBooleanBinaryExpression.java

```
1  // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import static jminusminus.CLConstants.*;
6
7  /**
8   * Most binary expressions that return booleans can be recognized by their
9   * syntax. We take advantage of this to define a common codegen(), which relies
10  * on the short-circuiting code generation for control and puts either a 1 or a
11  * 0 onto the stack.
12  */
13
14  abstract class JBooleanBinaryExpression extends JBinaryExpression {
15
16      /**
17       * Construct an AST node for a boolean binary expression.
18       *
19       * @param line
20       *     line in which the boolean binary expression occurs in the
21       *     source file.
22       * @param operator
23       *     the boolean binary operator.
24       * @param lhs
25       *     lhs operand.
26       * @param rhs
27       *     rhs operand.
28      */
29
30      protected JBooleanBinaryExpression(int line, String operator,
31          JExpression lhs, JExpression rhs) {
32          super(line, operator, lhs, rhs);
33      }
34
35      /**
36       * Generate code for the case where we actually have a boolean value (true
37       * or false) computed onto the stack, eg for assignment to a boolean
38       * variable.
39       *
40       * @param output
41       *     the code emitter (basically an abstraction for producing the
42       *     .class file).
43      */
44
45      public void codegen(CLEmitter output) {
46          String elseLabel = output.createLabel();
47          String endIfLabel = output.createLabel();
48          this.codegen(output, elseLabel, false);
49          output.addNoArgInstruction(ICONST_1); // true
50          output.addBranchInstruction(GOTO, endIfLabel);
51          output.addLabel(elseLabel);
52          output.addNoArgInstruction(ICONST_0); // false
53          output.addLabel(endIfLabel);
54      }
55  }
56
57  /**
58   * The AST node for an equality (==) expression. Implements short-circuiting
59   * branching.
60   */
61
62  class JEqualOp extends JBooleanBinaryExpression {
63
64      /**
65       * Construct an AST node for an equality expression.
66       *
67       * @param line
68       *     line in which the equality expression occurs in the
69       *     source file.
70       * @param lhs
71       *     lhs operand.
72       * @param rhs
73       *     rhs operand.
74      */
75
76      protected JEqualOp(int line, String operator, JExpression lhs, JExpression rhs) {
77          super(line, operator, lhs, rhs);
78      }
79
80      public void codegen(CLEmitter output) {
81          String elseLabel = output.createLabel();
82          String endIfLabel = output.createLabel();
83          this.codegen(output, elseLabel, false);
84          output.addNoArgInstruction(ICONST_1); // true
85          output.addBranchInstruction(GOTO, endIfLabel);
86          output.addLabel(elseLabel);
87          output.addNoArgInstruction(ICONST_0); // false
88          output.addLabel(endIfLabel);
89      }
90  }
```

```

67      *
68      * @param line
69      *         line number in which the equality expression occurs in the
70      *         source file.
71      * @param lhs
72      *         lhs operand.
73      * @param rhs
74      *         rhs operand.
75      */
76
77      public JEqualOp(int line, JExpression lhs, JExpression rhs) {
78          super(line, "==", lhs, rhs);
79      }
80
81      /**
82       * Analyzing an equality expression means analyzing its operands and
83       * checking that the types match.
84       *
85       * @param context
86       *         context in which names are resolved.
87       * @return the analyzed (and possibly rewritten) AST subtree.
88       */
89
90      public JExpression analyze(Context context) {
91          lhs = (JExpression) lhs.analyze(context);
92          rhs = (JExpression) rhs.analyze(context);
93          lhs.type().mustMatchExpected(line(), rhs.type());
94          type = Type.BOOLEAN;
95          return this;
96      }
97
98      /**
99       * Branching code generation for == operation.
100      *
101      * @param output
102      *         the code emitter (basically an abstraction for producing the
103      *         .class file).
104      * @param targetLabel
105      *         target for generated branch instruction
106      * @param onTrue
107      *         should we branch on true?
108      */
109
110      public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
111          lhs.codegen(output);
112          rhs.codegen(output);
113          if (lhs.type().isReference()) {
114              output.addBranchInstruction(onTrue ? IF_ACMPEQ : IF_ACMUNE,
115                  targetLabel);
116          } else {
117              output.addBranchInstruction(onTrue ? IF_ICMPEQ : IF_ICMUNE,
118                  targetLabel);
119          }
120      }
121
122  }
123
124  /**
125   * The AST node for a logical AND (&) expression. Implements short-circuiting
126   * branching.
127   */
128
129  class JLogicalAndOp extends JBooleanBinaryExpression {
130
131      /**
132       * Construct an AST node for a logical AND expression given its line number,
133       * and lhs and rhs operands.
134       *
135       * @param line

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

136 *           line in which the logical AND expression occurs in the source
137 *           file.
138 * @param lhs
139 *           lhs operand.
140 * @param rhs
141 *           rhs operand.
142 */
143
144 public JLogicalAndOp(int line, JExpression lhs, JExpression rhs) {
145     super(line, "&&", lhs, rhs);
146 }
147
148 /**
149  * Analyzing a logical AND expression involves analyzing its operands and
150  * insuring they are boolean; the result type is of course boolean.
151  *
152  * @param context
153  *       context in which names are resolved.
154  * @return the analyzed (and possibly rewritten) AST subtree.
155  */
156
157 public JExpression analyze(Context context) {
158     lhs = (JExpression) lhs.analyze(context);
159     rhs = (JExpression) rhs.analyze(context);
160     lhs.type().mustMatchExpected(line(), Type.BOOLEAN);
161     rhs.type().mustMatchExpected(line(), Type.BOOLEAN);
162     type = Type.BOOLEAN;
163     return this;
164 }
165
166 /**
167  * The semantics of j-- require that we implement short-circuiting branching
168  * in implementing the logical AND.
169  *
170  * @param output
171  *       the code emitter (basically an abstraction for producing the
172  *       .class file).
173  * @param targetLabel
174  *       target for generated branch instruction
175  * @param onTrue
176  *       should we branch on true?
177  */
178
179 public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
180     if (onTrue) {
181         String falseLabel = output.createLabel();
182         lhs.codegen(output, falseLabel, false);
183         rhs.codegen(output, targetLabel, true);
184         output.addLabel(falseLabel);
185     } else {
186         lhs.codegen(output, targetLabel, false);
187         rhs.codegen(output, targetLabel, false);
188     }
189 }
190
191 }
192

```

**Assignment Project Exam Help**

<https://powcoder.com>

**Add WeChat powcoder**