```
1  // Copyright 2011 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import static jminusminus.CLConstants.*;
6
7  /**
8   * The AST for an array indexing operation. It has an expression
9   * denoting an array object and an expression denoting an integer
10  * index.
11  */
12
13 class JArrayExpression
14     extends JExpression implements JLhs {
15
16     /** The array. */
17     private JExpression theArray;
18
19     /** The array index expression. */
20     private JExpression indexExpr;
21
22     /**
23      * Construct an AST node for an array indexing operation.
24      *
25      * @param line
26      *                  line in which the operation occurs in the
27      *                  source file.
28      * @param theArray
29      *                  the array we're indexing.
30      * @param indexExpr
31      *                  the index expression.
32      */
33
34     public JArrayExpression(int line, JExpression theArray,
35         JExpression indexExpr) {
36         super(line);
37         this.theArray = theArray;
38         this.indexExpr = indexExpr;
39     }
40
41     /**
42      * Perform semantic analysis on an array indexing expression
43      * such as A[i].
44      *
45      * @param context
46      *                  context in which names are resolved.
47      * @return the analyzed (and possibly rewritten) AST subtree.
48      */
49
50     public JExpression analyze(Context context) {
51         theArray = (JExpression) theArray.analyze(context);
52         indexExpr = (JExpression) indexExpr.analyze(context);
53         if (!(theArray.type().isArray())) {
54             JAST.compilationUnit.reportSemanticError(line(),
55                 "attempt to index a non-array object");
56             this.type = Type.ANY;
57         } else {
58             this.type = theArray.type().componentType();
59         }
60         indexExpr.type().mustMatchExpected(line(), Type.INT);
61         return this;
62     }
63
64     /**
65      * Analyzing the array expression as an Lvalue is like
66      * analyzing it for its Rvalue.
```

```
67          *
68          * @param context
69          *                context in which names are resolved.
70          */
71
72      public JExpression analyzeLhs(Context context) {
73          analyze(context);
74          return this;
75      }
76
77      /**
78       * Perform code generation from the JArrayExpression using
79       * the specified code emitter. Generate the code necessary
80       * for loading the Rvalue.
81       *
82       * @param output
83       *                the code emitter (basically an abstraction
84       *                for producing the .class file).
85       */
86
87      public void codegen(CLEmitter output) {
88          theArray.codegen(output);
89          indexExpr.codegen(output);
90          if (type == Type.INT) {
91              output.addNoArgInstruction(IALOAD);
92      } else if (type == Type.BOOLEAN) {
93              output.addNoArgInstruction(BALOAD);
94      } else if (type == Type.CHAR) {
95              output.addNoArgInstruction(CALOAD);
96          } else if (type.isPrimitive()) {
97              output.addNoArgInstruction(AALOAD);
98          }
99      }
100
101     /**
102      * Generate the code required for setting up an Lvalue, eg
103      * for use in an assignment. Here, this requires loading the
104      * array and the index.
105      *
106      * @param output
107      *                the code emitter (basically an abstraction
108      *                for producing the .class file).
109      */
110
111     public void codegenLoadLhsLvalue(CLEmitter output) {
112         // Load the lvalue onto the stack: the array and the
113         // index.
114         theArray.codegen(output);
115         indexExpr.codegen(output);
116     }
117
118     /**
119      * Generate the code required for loading an Rvalue for this
120      * variable, eg for use in a +=. Here, this requires
121      * duplicating the array and the index on the stack and doing
122      * an array load.
123      *
124      * @param output
125      *                the code emitter (basically an abstraction
126      *                for producing the .class file).
127      */
128
129     public void codegenLoadLhsRvalue(CLEmitter output) {
130         // Load rvalue onto stack, by duplicating the lvalue,
131         // and fetching it's content
132         if (type == Type.STRING) {
133             output.addNoArgInstruction(DUP2_X1);
134         } else {
135             output.addNoArgInstruction(DUP2);
```

```
        }
    if (type == Type.INT) {
        output.addNoArgInstruction(IALOAD);
    } else if (type == Type.BOOLEAN) {
        output.addNoArgInstruction(BALOAD);
    } else if (type == Type.CHAR) {
        output.addNoArgInstruction(CALOAD);
        } else if (!type.isPrimitive()) {
            output.addNoArgInstruction(AALOAD);
        }
    }

    /**
     * Generate the code required for duplicating the Rvalue that
     * is on the stack becuase it is to be used in a surrounding
     * expression, as in a[i] = x = <expr> or x = y--. Here this
     * means copying it down two locations (beneath the array and
     * index).
     *
     * @param output
     *                the code emitter (basically an abstraction
     *                for producing the .class file).
     */

    public void codegenDuplicateRvalue(CLEmitter output) {
        // It's copied down below the array and index
        output.addNoArgInstruction(DUP_X2);
    }

    /**
     * Generate the code required for doing the actual
     * assignment. Here, this requires an array store.
     *
     * @param output
     *                the code emitter (basically an abstraction
     *                for producing the .class file).
     */

    public void codegenStore(CLEmitter output) {
    if (type == Type.INT) {
        output.addNoArgInstruction(IASTORE);
    } else if (type == Type.BOOLEAN) {
        output.addNoArgInstruction(BASTORE);
    } else if (type == Type.CHAR) {
        output.addNoArgInstruction(CASTORE);
        } else if (!type.isPrimitive()) {
            output.addNoArgInstruction(AASTORE);
        }

    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.println("<JArrayExpression>");
        p.indentRight();
        if (theArray != null) {
            p.println("<TheArray>");
            p.indentRight();
            theArray.writeToStdOut(p);
            p.indentLeft();
            p.println("</TheArray>");
        }
        if (indexExpr != null) {
            p.println("<IndexExpression>");
            p.indentRight();
            indexExpr.writeToStdOut(p);
```

```
205             p.indentLeft();
206             p.println("</IndexExpression>");
207         }
208         p.indentLeft();
209         p.println("</JArrayExpression>");
210     }
211 }
212
```