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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a field selection operation. It has a target object, a field
 * name, and the Field it defines.
 */

class JFieldSelection extends JExpression implements JLhs {

    /** The target expression. */
    protected JExpression target;

    /** The ambiguous part that is reclassified in analyze(). */
    private AmbiguousName ambiguousPart;

    /** The field name. */
    private String fieldName;

    /** The Field representing this field. */
    private Field field;

    /**
     * Construct an AST node for a field selection without an ambiguous part.
     *
     * @param line
     *            the line number of the selection.
     * @param target
     *            the target of the selection.
     * @param fieldName
     *            the field name.
     */

    public JFieldSelection(int line, JExpression target, String fieldName) {
        this(line, null, target, fieldName);
    }

    /**
     * Construct an AST node for a field selection having an ambiguous part.
     *
     * @param line
     *            line in which the field selection occurs in the source file.
     * @param ambiguousPart
     *            the ambiguous part.
     * @param target
     *            the target of the selection.
     * @param fieldName
     *            the field name.
     */

    public JFieldSelection(int line, AmbiguousName ambiguousPart,
            JExpression target, String fieldName) {
        super(line);
        this.ambiguousPart = ambiguousPart;
        this.target = target;
        this.fieldName = fieldName;
    }

    /**
     * Analyzing a field selection expression involves, (1) reclassifying any
     * ambiguous part, (2) analyzing the target, (3) treating "length" field of
     * arrays specially, or computing the Field object, (4) checking the access
     * rules, and (5) computing the resultant type.
```

```
67       *
68       * @param context
69       *            context in which names are resolved.
70       * @return the analyzed (and possibly rewritten) AST subtree.
71       */
72
73      public JExpression analyze(Context context) {
74          // Reclassify the ambiguous part.
75          if (ambiguousPart != null) {
76              JExpression expr = ambiguousPart.reclassify(context);
77              if (expr != null) {
78                  if (target == null)
79                      target = expr;
80                  else {
81                      // Can't even happen syntactically
82                      JAST.compilationUnit.reportSemanticError(line(),
83                              "Badly formed suffix");
84                  }
85              }
86          }
87          target = (JExpression) target.analyze(context);
88          Type targetType = target.type();
89
90          // We use a workaround for the "length" field of arrays.
91          if ((targetType.isArray()) && fieldName.equals("length")) {
92              type = Type.INT;
93          } else {
94              // Other than that, targetType has to be a
95              // ReferenceType
96              if (targetType.isPrimitive()) {
97                  JAST.compilationUnit.reportSemanticError(line(),
98                          "Target of a field selection must "
99                                  + "be a defined type");
100                 type = Type.ANY;
101                 return this;
102             }
103             field = targetType.fieldFor(fieldName);
104             if (field == null) {
105                 JAST.compilationUnit.reportSemanticError(line(),
106                         "Cannot find a field: " + fieldName);
107                 type = Type.ANY;
108             } else {
109                 context.definingType().checkAccess(line, (Member) field);
110                 type = field.type();
111
112                 // Non-static field cannot be referenced from a static context.
113                 if (!field.isStatic()) {
114                     if (target instanceof JVariable
115                             && ((JVariable) target).iDefn() instanceof
TypeNameDefn) {
116                         JAST.compilationUnit
117                                 .reportSemanticError(
118                                         line(),
119                                         "Non-static field "
120                                                 + fieldName
121                                                 + " cannot be referenced from a
static context");
122                     }
123                 }
124             }
125         }
126         return this;
127     }
128
129     /**
130      * Analyze the field selection expression for use on the lhs of an
131      * assignment. Although the final keyword is not in j--, we do make use of
132      * the Java api and so must repect its constraints.
133      *
```

```
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JExpression analyzeLhs(Context context) {
        JExpression result = analyze(context);
        if (field.isFinal()) {
            JAST.compilationUnit.reportSemanticError(line, "The field "
                    + fieldName + " in type " + target.type.toString()
                    + " is declared final.");
        }
        return result;
    }

    /**
     * Generate the code necessary to load the Rvalue for this field selection.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output) {
        target.codegen(output);

        // We use a workaround for the "length" field of arrays
        if ((target.type().isArray()) && fieldName.equals("length")) {
            output.addNoArgInstruction(ARRAYLENGTH);
        } else {
            int mnemonic = field.isStatic() ? GETSTATIC : GETFIELD;
            output.addMemberAccessInstruction(mnemonic,
                    target.type().jvmName(), fieldName, type.toDescriptor());
        }
    }

    /**
     * The semantics of && require that we implement short-circuiting branching
     * in implementing field selections.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     * @param targetLabel
     *            the label to which we should branch.
     * @param onTrue
     *            do we branch on true?
     */

    public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
        // Push the value
        codegen(output);

        if (onTrue) {
            // Branch on true
            output.addBranchInstruction(IFNE, targetLabel);
        } else {
            // Branch on false
            output.addBranchInstruction(IFEQ, targetLabel);
        }
    }

    /**
     * Generate the code required for setting up an Lvalue, eg, for use in an
     * assignment.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
```

```
     */

    public void codegenLoadLhsLvalue(CLEmitter output) {
        // Nothing to do for static fields.
        if (!field.isStatic()) {
            // Just load the target
            target.codegen(output);
        }
    }

    /**
     * Generate the code required for loading an Rvalue for this variable, eg
     * for use in a +=. Here, this requires either a getstatic or getfield.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegenLoadLhsRvalue(CLEmitter output) {
        String descriptor = field.type().toDescriptor();
        if (field.isStatic()) {
            output.addMemberAccessInstruction(GETSTATIC, target.type()
                    .jvmName(), fieldName, descriptor);
        } else {
            output.addNoArgInstruction(type == Type.STRING ? DUP_X1 : DUP);
            output.addMemberAccessInstruction(GETFIELD,
                    target.type().jvmName(), fieldName, descriptor);
        }
    }

    /**
     * Generate the code required for duplicating the Rvalue that is on the
     * stack because it is to be used in a surrounding expression, as in a[i] =
     * x = <expr>, or x = y = 1, where this means copying it down
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file)
     */

    public void codegenDuplicateRvalue(CLEmitter output) {
        if (field.isStatic()) {
            output.addNoArgInstruction(DUP);
        } else {
            output.addNoArgInstruction(DUP_X1);
        }
    }

    /**
     * Generate the code required for doing the actual assignment.
     *
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegenStore(CLEmitter output) {
        String descriptor = field.type().toDescriptor();
        if (field.isStatic()) {
            output.addMemberAccessInstruction(PUTSTATIC, target.type()
                    .jvmName(), fieldName, descriptor);
        } else {
            output.addMemberAccessInstruction(PUTFIELD,
                    target.type().jvmName(), fieldName, descriptor);
        }
    }

    /**
```

```
272      * @inheritDoc
273      */
274
275     public void writeToStdOut(PrettyPrinter p) {
276         p.printf("<JFieldSelection line=\"%d\" name=\"%s\"/>\n", line(),
277                 fieldName);
278         p.indentRight();
279         if (target != null) {
280             p.println("<Target>");
281             p.indentRight();
282             target.writeToStdOut(p);
283             p.indentLeft();
284             p.println("</Target>");
285         }
286         p.indentLeft();
287         p.println("</JFieldSelection>");
288     }
289
290 }
291
```