JVariable.java

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for an identifier used as a primary expression.
 */

class JVariable extends JExpression implements JLhs {

    /** The variable's name. */
    private String name;

    /** The variable's definition. */
    private IDefn iDefn;

    /** Was analyzeLhs() done? */
    private boolean analyzeLhs;

    /**
     * Construct the AST node for a variable given its line number and name.
     *
     * @param line
     *            line in which the variable occurs in the source file.
     * @param name
     *            the name.
     */

    public JVariable(int line, String name) {
        super(line);
        this.name = name;
    }

    /**
     * Return the identifier name.
     *
     * @return the identifier name.
     */

    public String name() {
        return name;
    }

    /**
     * Return the identifier's definition.
     *
     * @return the identifier's definition.
     */

    public IDefn iDefn() {
        return iDefn;
    }

    /**
     * Analyzing identifiers involves resolving them in the context. Identifiers
     * denoting fileds (with implicit targets) are rewritten as explicit field
     * selection operations.
     *
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JExpression analyze(Context context) {
        iDefn = context.lookup(name);
        if (iDefn == null) {
            // Not a local, but is it a field?
            Type definingType = context.definingType();
            Field field = definingType.fieldFor(name);
            if (field == null) {
                type = Type.ANY;
                JAST.compilationUnit.reportSemanticError(line,
                        "Cannot find name: " + name);
            } else {
                // Rewrite a variable denoting a field as an
                // explicit field selection
                type = field.type();
                JExpression newTree = new JFieldSelection(line(), field
                        .isStatic()
                        || (context.methodContext() != null && context
                                .methodContext().isStatic()) ? new JVariable(
                        line(), definingType.toString()) : new JThis(line),
                        name);
                return (JExpression) newTree.analyze(context);
            }
        } else {
            if (!analyzeLhs && iDefn instanceof LocalVariableDefn
                    && !((LocalVariableDefn) iDefn).isInitialized()) {
                JAST.compilationUnit.reportSemanticError(line, "Variable "
                        + name + " might not have been initialized");
            }
            type = iDefn.type();
        }
        return this;
    }
```

```

    /**
     * Analyze the identifier as used on the lhs of an assignment.
     * 
     * @param context
     *             context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */
```

```

    public JExpression analyzeLhs(Context context) {
        analyzeLhs = true;
        JExpression newTree = analyze(context);
        if (newTree instanceof JVariable) {
            // Could (now) be a JFieldSelection, but if it's
            // (still) a JVariable
            if (iDefn != null && !(iDefn instanceof LocalVariableDefn)) {
                JAST.compilationUnit.reportSemanticError(line(), name
                        + " is a bad lhs to a  =");
            }
        }
        return newTree;
    }

    /**
     * Generate code to load value of variable on stack.
     * 
     * @param output
     *             the code emitter (basically an abstraction for producing the
     *             .class file).
     */

    public void codegen(CLEmitter output) {
        if (iDefn instanceof LocalVariableDefn) {
            int offset = ((LocalVariableDefn) iDefn).offset();
            if (type.isReference()) {
                switch (offset) {
                case 0:
                    output.addNoArgInstruction(ALOAD_0);
```

```java
                    break;
                case 1:
                    output.addNoArgInstruction(ALOAD_1);
                    break;
                case 2:
                    output.addNoArgInstruction(ALOAD_2);
                    break;
                case 3:
                    output.addNoArgInstruction(ALOAD_3);
                    break;
                default:
                    output.addOneArgInstruction(ALOAD, offset);
                    break;
                }
            } else {
                // Primitive types
                if (type == Type.INT || type == Type.BOOLEAN
                        || type == Type.CHAR) {
                    switch (offset) {
                    case 0:
                        output.addNoArgInstruction(ILOAD_0);
                        break;
                    case 1:
                        output.addNoArgInstruction(ILOAD_1);
                        break;
                    case 2:
                        output.addNoArgInstruction(ILOAD_2);
                        break;
                    case 3:
                        output.addNoArgInstruction(ILOAD_3);
                        break;
                    default:
                        output.addOneArgInstruction(ILOAD, offset);
                        break;
                    }
                }
            }
        }
    }

    /**
     * The semantics of j-- require that we implement short-circuiting branching
     * in implementing the identifier expression.
     * 
     * @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) {
        if (iDefn instanceof LocalVariableDefn) {
            // Push the value
            codegen(output);

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

    /**
```

```java
     * Generate the code required for setting up an Lvalue, eg for use in an
     * assignment. Here, this requires nothing; all information is in the the
     * store instruction.
     * 
     * @param output
     *            the emitter (an abstraction of the class file.
     */

    public void codegenLoadLhsLvalue(CLEmitter output) {
        // Nothing goes here.
    }

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

    public void codegenLoadLhsRvalue(CLEmitter output) {
        codegen(output);
    }

    /**
     * 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 simply duplicating the value on
     * the stack.
     * 
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegenDuplicateRvalue(CLEmitter output) {
        if (iDefn instanceof LocalVariableDefn) {
            // It's copied atop the stack.
            output.addNoArgInstruction(DUP);
        }
    }

    /**
     * Generate the code required for doing the actual assignment. Here, this
     * requires storing what's on the stack at the appropriate offset.
     * 
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegenStore(CLEmitter output) {
        if (iDefn instanceof LocalVariableDefn) {
            int offset = ((LocalVariableDefn) iDefn).offset();
            if (type.isReference()) {
                switch (offset) {
                case 0:
                    output.addNoArgInstruction(ASTORE_0);
                    break;
                case 1:
                    output.addNoArgInstruction(ASTORE_1);
                    break;
                case 2:
                    output.addNoArgInstruction(ASTORE_2);
                    break;
                case 3:
                    output.addNoArgInstruction(ASTORE_3);
                    break;
                default:
                    output.addOneArgInstruction(ASTORE, offset);
                    break;
                }
            } else {
                // Primitive types
                if (type == Type.INT || type == Type.BOOLEAN
                        || type == Type.CHAR) {
                    switch (offset) {
                    case 0:
                        output.addNoArgInstruction(ISTORE_0);
                        break;
                    case 1:
                        output.addNoArgInstruction(ISTORE_1);
                        break;
                    case 2:
                        output.addNoArgInstruction(ISTORE_2);
                        break;
                    case 3:
                        output.addNoArgInstruction(ISTORE_3);
                        break;
                    default:
                        output.addOneArgInstruction(ISTORE, offset);
                        break;
                    }
                }
            }
        }
    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.println("<JVariable name=\"" + name + "\" />");
    }

}
```