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

package jminusminus;

import java.util.ArrayList;

/**
 * The abstract syntax tree (AST) node representing a compilation unit, and so
 * the root of the AST.
 *
 * It keeps track of the name of the source file, its package name, a list of
 * imported types, a list of type (eg class) declarations, and a flag indicating
 * if a semantic error has been detected in analysis or code generation. It also
 * maintains a CompilationUnitContext (built in pre-analysis) for declaring both
 * imported and declared types.
 *
 * The AST is produced by the Parser. Once the AST has been built, three
 * successive methods are invoked:
 *
 * (1) Method preAnalyze() is invoked for making a first pass at type analysis,
 * recursively reaching down to the member headers for declaring types and
 * member interfaces in the environment (contexts). preAnalyze() also creates a
 * partial class file (in memory) for recording member header information, using
 * the partialCodegen() method.
 *
 * (2) Method analyze() is invoked for type-checking field initializations and
 * method bodies, and determining the types of all expressions. A certain amount
 * of tree surgery is also done here. And stack frame offsets are computed for
 * method parameters and local variables.
 *
 * (3) Method codegen() is invoked for generating code for the compilation unit
 * to a class file. For each type declaration, it instantiates a CLEmmiter
 * object (an abstraction of the class file) and then invokes methods on that
 * CLEmitter for generating instructions. At the end of each type declaration, a
 * method is invoked on the CLEmitter which writes the class out to the file
 * system either as a class file or as a .S (SPIM) file. Of course, codegen()
 * makes recursive calls down the tree, to the codegen() methods at each node,
 * for generating the appropriate instructions.
 */

class JCompilationUnit extends JAST {

    /** Name of the source file. */
    private String fileName;

    /** Package name. */
    private TypeName packageName;

    /** List of imports. */
    private ArrayList<TypeName> imports;

    /** List of type declarations. */
    private ArrayList<JAST> typeDeclarations;

    /**
     * List of CLFile objects corresponding to the type declarations in this
     * compilation unit.
     */
    private ArrayList<CLFile> clFiles;

    /** For imports and type declarations. */
    private CompilationUnitContext context;

    /** Whether a semantic error has been found. */
    private boolean isInError;

```

```
67      /**
68       * Construct an AST node for a compilation unit given a file name, class
69       * directory, line number, package name, list of imports, and type
70       * declarations.
71       *
72       * @param fileName
73       *            the name of the source file.
74       * @param line
75       *            line in which the compilation unit occurs in the source file.
76       * @param packageName
77       *            package name.
78       * @param imports
79       *            a list of imports.
80       * @param typeDeclarations
81       *            type declarations.
82       */
83
84      public JCompilationUnit(String fileName, int line, TypeName packageName,
85              ArrayList<TypeName> imports, ArrayList<JAST> typeDeclarations) {
86          super(line);
87          this.fileName = fileName;
88          this.packageName = packageName;
89          this.imports = imports;
90          this.typeDeclarations = typeDeclarations;
91          clFiles = new ArrayList<CLFile>();
92          compilationUnit = this;
93      }
94
95      /**
96       * The package in which this compilation unit is defined.
97       *
98       * @return the package name.
99       */
100
101     public String packageName() {
102         return packageName == null ? "" : packageName.toString();
103     }
104
105     /**
106      * Has a semantic error occurred up to now?
107      *
108      * @return true or false.
109      */
110
111     public boolean errorHasOccurred() {
112         return isInError;
113     }
114
115     /**
116      * Report a semantic error.
117      *
118      * @param line
119      *            line in which the error occurred in the source file.
120      * @param message
121      *            message identifying the error.
122      * @param arguments
123      *            related values.
124      */
125
126     public void reportSemanticError(int line, String message,
127             Object... arguments) {
128         isInError = true;
129         System.err.printf("%s:%d: ", fileName, line);
130         System.err.printf(message, arguments);
131         System.err.println();
132     }
133
134     /**
135      * Construct a context for the compilation unit, initializing it with
```

```
136      * imported types. Then pre-analyze the unit's type declarations, adding
137      * their types to the context.
138      */
139
140     public void preAnalyze() {
141         context = new CompilationUnitContext();
142
143         // Declare the two implicit types java.lang.Object and
144         // java.lang.String
145         context.addType(0, Type.OBJECT);
146         context.addType(0, Type.STRING);
147
148         // Declare any imported types
149         for (TypeName imported : imports) {
150             try {
151                 Class<?> classRep = Class.forName(imported.toString());
152                 context.addType(imported.line(), Type.typeFor(classRep));
153             } catch (Exception e) {
154                 JAST.compilationUnit.reportSemanticError(imported.line(),
155                         "Unable to find %s", imported.toString());
156             }
157         }
158
159         // Declare the locally declared type(s)
160         CLEmitter.initializeByteClassLoader();
161         for (JAST typeDeclaration : typeDeclarations) {
162             ((JTypeDecl) typeDeclaration).declareThisType(context);
163         }
164
165         // Pre-analyze the locally declared types, generate
166         // (partial) class instances, reflecting only the member
167         // interface type information
168         CLEmitter.initializeByteClassLoader();
169         for (JAST typeDeclaration : typeDeclarations) {
170             ((JTypeDecl) typeDeclaration).preAnalyze(context);
171         }
172     }
173
174     /**
175      * Perform semantic analysis on the AST in the specified context.
176      *
177      * @param context
178      *            context in which names are resolved (ignored here).
179      * @return the analyzed (and possibly rewritten) AST subtree.
180      */
181
182     public JAST analyze(Context context) {
183         for (JAST typeDeclaration : typeDeclarations) {
184             typeDeclaration.analyze(this.context);
185         }
186         return this;
187     }
188
189     /**
190      * Generating code for a compilation unit means generating code for each of
191      * the type declarations.
192      *
193      * @param output
194      *            the code emitter (basically an abstraction for producing the
195      *            .class file).
196      */
197
198     public void codegen(CLEmitter output) {
199         for (JAST typeDeclaration : typeDeclarations) {
200             typeDeclaration.codegen(output);
201             output.write();
202             clFiles.add(output.clFile());
203         }
204     }
```

```
205
206     /**
207      * Return the list of CLFile objects corresponding to the type declarations
208      * in this compilation unit.
209      *
210      * @return list of CLFile objects.
211      */
212
213     public ArrayList<CLFile> clFiles() {
214         return clFiles;
215     }
216
217     /**
218      * @inheritDoc
219      */
220
221     public void writeToStdOut(PrettyPrinter p) {
222         p.println("<?xml version=\"1.0\" encoding=\"utf-8\"?>");
223         p.printf("<JCompilationUnit line=\"%d\">\n", line());
224         p.indentRight();
225         p.printf("<Source fileName=\"%s\"/>\n", fileName);
226         if (context != null) {
227             context.writeToStdOut(p);
228         }
229         if (packageName != null) {
230             p.printf("<Package name=\"%s\"/>\n", packageName());
231         }
232         if (imports != null) {
233             p.println("<Imports>");
234             p.indentRight();
235             for (TypeName imported : imports) {
236                 p.printf("<Import name=\"%s\"/>\n", imported.toString());
237             }
238             p.indentLeft();
239             p.println("</Imports>");
240         }
241         if (typeDeclarations != null) {
242             p.println("<TypeDeclarations>");
243             p.indentRight();
244             for (JAST typeDeclaration : typeDeclarations) {
245                 typeDeclaration.writeToStdOut(p);
246             }
247             p.indentLeft();
248             p.println("</TypeDeclarations>");
249         }
250         p.indentLeft();
251         p.println("</JCompilationUnit>");
252     }
253
254 }
255
```