```
1   // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3   package jminusminus;
4
5   import java.util.Map;
6   import java.util.HashMap;
7   import java.util.Set;
8
9   /**
10   * A Context encapsulates the environment in which an AST is analyzed. It
11   * represents a scope; the scope of a variable is captured by its context. It's
12   * the symbol table.
13   *
14   * Because scopes are lexically nested in Java (and so in j--), the environment
15   * can be seen as a stack of contexts, each of which is a mapping from names to
16   * their definitions (IDefns). A Context keeps track of it's (most closely)
17   * surrounding context, its surrounding class context, and its surrounding
18   * compilation unit context, as well as a map of from names to definitions in
19   * the level of scope the Context represents. Contexts are created for the
20   * compilation unit (a CompilationUnitContext), a class (a ClassContext), each
21   * method (a MethodContext), and each block (a LocalContext). If we were to add
22   * the for-statement to j--, we would necessarily create a (local) context.
23   *
24   * From the outside, the structure looks like a tree strung over the AST. But
25   * from any location on the AST, that is from any point along a particular
26   * branch, it looks like a stack of context objects leading back to the root of
27   * the AST, that is, back to the JCompilationUnit object at the root.
28   *
29   * Part of this structure is built during pre-analysis; pre-analysis reaches
30   * only into the type (eg class) declaration for typing the members;
31   * pre-analysis does not reach into the method bodies. The rest of it is built
32   * during analysis.
33   */
34
35  class Context {
36
37      /** The surrounding context (scope). */
38      protected Context surroundingContext;
39
40      /** The surrounding class context. */
41      protected ClassContext classContext;
42
43      /**
44       * The compilation unit context (for the whole source program or file).
45       */
46      protected CompilationUnitContext compilationUnitContext;
47
48      /**
49       * Map of (local variable, formal parameters, type) names to their
50       * definitions.
51       */
52      protected Map<String, IDefn> entries;
53
54      /**
55       * Construct a Context.
56       *
57       * @param surrounding
58       *            the surrounding context (scope).
59       * @param classContext
60       *            the surrounding class context.
61       * @param compilationUnitContext
62       *            the compilation unit context (for the whole source program or
63       *            file).
64       */
65
66      protected Context(Context surrounding, ClassContext classContext,
```

```
67              CompilationUnitContext compilationUnitContext) {
68          this.surroundingContext = surrounding;
69          this.classContext = classContext;
70          this.compilationUnitContext = compilationUnitContext;
71          this.entries = new HashMap<String, IDefn>();
72      }
73
74      /**
75       * Add an entry to the symbol table, binding a name to its definition in the
76       * current context.
77       *
78       * @param name
79       *            the name being declared.
80       * @param definition
81       *            and its definition.
82       */
83
84      public void addEntry(int line, String name, IDefn definition) {
85          if (entries.containsKey(name)) {
86              JAST.compilationUnit.reportSemanticError(line, "redefining name: "
87                      + name);
88          } else {
89              entries.put(name, definition);
90          }
91      }
92
93      /**
94       * Return the definition for a name in the environment. If it's not found in
95       * this context, we look for it in the surrounding context(s).
96       *
97       * @param name
98       *            the name whose definition we're looking for.
99       * @return the definition (or null, if not found).
100      */
101
102     public IDefn lookup(String name) {
103         IDefn iDefn = (IDefn) entries.get(name);
104         return iDefn != null ? iDefn
105                 : surroundingContext != null ? surroundingContext.lookup(name)
106                         : null;
107     }
108
109     /**
110      * Return the definition for a type name in the environment. For now, we
111      * look for types only in the CompilationUnitContext.
112      *
113      * @param name
114      *            the name of the type whose definition we're looking for.
115      * @return the definition (or null, if not found).
116      */
117
118     public Type lookupType(String name) {
119         TypeNameDefn defn = (TypeNameDefn) compilationUnitContext.lookup(name);
120         return defn == null ? null : defn.type();
121     }
122
123     /**
124      * Add the type to the environment.
125      *
126      * @param line
127      *            line number of type declaration.
128      * @param type
129      *            the type we are declaring.
130      */
131
132     public void addType(int line, Type type) {
133         IDefn iDefn = new TypeNameDefn(type);
134         compilationUnitContext.addEntry(line, type.simpleName(), iDefn);
135         if (!type.toString().equals(type.simpleName())) {
```

```
            compilationUnitContext.addEntry(line, type.toString(), iDefn);
        }
    }

    /**
     * The type that defines this context (used principally for checking
     * acessibility).
     *
     * @return the type that defines this context.
     */

    public Type definingType() {
        return ((JTypeDecl) classContext.definition()).thisType();
    }

    /**
     * Return the surrounding context (scope) in the stack of contexts.
     *
     * @return the surrounding context.
     */

    public Context surroundingContext() {
        return surroundingContext;
    }

    /**
     * Return the surrounding class context.
     *
     * @return the surrounding class context.
     */

    public ClassContext classContext() {
        return classContext;
    }

    /**
     * Return the surrounding compilation unit context. This is where imported
     * types and other types defined in the compilation unit are declared.
     *
     * @return the compilation unit context.
     */

    public CompilationUnitContext compilationUnitContext() {
        return compilationUnitContext;
    }

    /**
     * Return the closest surrounding method context. Return null if we're not
     * within a method.
     *
     * @return the method context.
     */

    public MethodContext methodContext() {
        Context context = this;
        while (context != null && !(context instanceof MethodContext)) {
            context = context.surroundingContext();
        }
        return (MethodContext) context;
    }

    /**
     * The names declared in this context.
     *
     * @return the set of declared names.
     */

    public Set<String> names() {
        return entries.keySet();
```

```
205     }
206
207     /**
208      * Write the contents of this context to STDOUT.
209      *
210      * @param p
211      *             for pretty printing with indentation.
212      */
213
214     public void writeToStdOut(PrettyPrinter p) {
215         // Nothing to write here
216     }
217
218 }
219
220 /**
221  * The compilation unit context is always the outermost context, and is where
222  * imported types and locally defined types (classes) are declared.
223  */
224
225 class CompilationUnitContext extends Context {
226
227     /**
228      * Construct a new compilation unit context. There are no surrounding
229      * contexts.
230      */
231
232     public CompilationUnitContext() {
233         super(null, null, null);
234         compilationUnitContext = this;
235     }
236
237     /**
238      * @inheritDoc
239      */
240
241     public void writeToStdOut(PrettyPrinter p) {
242         p.println("<CompilationUnitContext>");
243         p.indentRight();
244         p.println("<Entries>");
245         if (entries != null) {
246             p.indentRight();
247             for (String key : names()) {
248                 p.println("<Entry>" + key + "</Entry>");
249             }
250             p.indentLeft();
251         }
252         p.println("</Entries>");
253         p.indentLeft();
254         p.println("</CompilationUnitContext>");
255     }
256
257 }
258
259 /**
260  * Represents the context (scope, environment, symbol table) for a type, eg a
261  * class, in j--. It also keeps track of its surrounding context(s), and the
262  * type whose context it represents.
263  */
264
265 class ClassContext extends Context {
266
267     /** AST node of the type that this class represents. */
268     private JAST definition;
269
270     /**
271      * Construct a class context.
272      *
273      * @param definition
```

```
274      *            the AST node of the type that this class represents.
275      * @param surrounding
276      *            the surrounding context(s).
277      */
278
279     public ClassContext(JAST definition, Context surrounding) {
280         super(surrounding, null, surrounding.compilationUnitContext());
281         classContext = this;
282         this.definition = definition;
283     }
284
285     /**
286      * Return the AST node of the type defined by this class.
287      *
288      * @return the AST of the type defined by this class.
289      */
290
291     public JAST definition() {
292         return definition;
293     }
294
295 }
296
297 /**
298  * A local context is a context (scope) in which local variables (including
299  * formal parameters) can be declared. Local variables are allocated at fixed
300  * offsets from the base of the current method's stack frame; this is done
301  * during anaysis. The definitions for local variables record these offsets. The
302  * offsets are used in code generation.
303  */
304
305 class LocalContext extends Context {
306
307     /** Next offset for a local variable. */
308     protected int offset;
309
310     /**
311      * Construct a local context. A local context is constructed for each block.
312      *
313      * @param surrounding
314      *            the surrounding context.
315      */
316
317     public LocalContext(Context surrounding) {
318         super(surrounding, surrounding.classContext(), surrounding
319                 .compilationUnitContext());
320         offset = (surrounding instanceof LocalContext) ? ((LocalContext)
surrounding)
321                 .offset()
322                 : 0;
323     }
324
325     /**
326      * The "next" offset. A simple getter. Not to be used for allocating new
327      * offsets (nextOffset() is used for that).
328      *
329      * @return the next available offset.
330      */
331
332     public int offset() {
333         return offset;
334     }
335
336     /**
337      * Allocate a new offset (eg for a parameter or local variable).
338      *
339      * @return the next allocated offset.
340      */
341
```

```
342     public int nextOffset() {
343         return offset++;
344     }
345
346     /**
347      * @inheritDoc
348      */
349
350     public void writeToStdOut(PrettyPrinter p) {
351         p.println("<LocalContext>");
352         p.indentRight();
353         p.println("<Entries>");
354         if (entries != null) {
355             p.indentRight();
356             for (String key : names()) {
357                 IDefn defn = entries.get(key);
358                 if (defn instanceof LocalVariableDefn) {
359                     p.printf("<Entry name=\"%s\" " + "offset=\"%d\"/>\n", key,
360                             ((LocalVariableDefn) defn).offset());
361                 }
362             }
363             p.indentLeft();
364         }
365         p.println("</Entries>");
366         p.indentLeft();
367         p.println("</LocalContext>");
368     }
369
370 }
371
372 /**
373  * A method context is where formal parameters are declared. Also, it's where we
374  * start computing the offsets for local variables (formal parameters included),
375  * which are allocated in the current stack frame (for a method invocation).
376  */
377
378 class MethodContext extends LocalContext {
379
380     /** Is this method static? */
381     private boolean isStatic;
382
383     /** Return type of this method. */
384     private Type methodReturnType;
385
386     /** Does (non-void) method have at least one return? */
387     private boolean hasReturnStatement = false;
388
389     /**
390      * Construct a method context.
391      *
392      * @param surrounding
393      *             the surrounding (class) context.
394      * @param isStatic
395      *             is this method static?
396      * @param methodReturnType
397      *             return type of this method.
398      */
399
400     public MethodContext(Context surrounding, boolean isStatic,
401             Type methodReturnType) {
402         super(surrounding);
403         this.isStatic = isStatic;
404         this.methodReturnType = methodReturnType;
405         offset = 0;
406     }
407
408     /**
409      * Is this method static?
410      *
```

```
411      * @return true or false.
412      */
413
414     public boolean isStatic() {
415         return isStatic;
416     }
417
418     /**
419      * Record fact that (non-void) method has at least one return.
420      */
421
422     public void confirmMethodHasReturn() {
423         hasReturnStatement = true;
424     }
425
426     /**
427      * Does this (non-void) method have at least one return?
428      *
429      * @return true or false.
430      */
431
432     public boolean methodHasReturn() {
433         return hasReturnStatement;
434     }
435
436     /**
437      * Return the return type of this method.
438      *
439      * @return return type of this method.
440      */
441
442     public Type methodReturnType() {
443         return methodReturnType;
444     }
445
446     /**
447      * @inheritDoc
448      */
449
450     public void writeToStdOut(PrettyPrinter p) {
451         p.println("<MethodContext>");
452         p.indentRight();
453         super.writeToStdOut(p);
454         p.indentLeft();
455         p.println("</MethodContext>");
456     }
457
458 }
459
```