```
 1 import java.util.Map;
 2 import java.util.HashMap;
 3 import java.lang.reflect.Modifier;
 4
 5 /** Implementation of name lookup. */
 6 aspect names {
 7     // qualified names for modules and import statements
 8     syn lazy String Import.getQualifiedName() = getPackage().isEmpty()
    ? getName() : getPackage() + "." + getName();
 9     syn lazy String Module.getQualifiedName() = getPackage().isEmpty()
    ? getName() : getPackage() + "." + getName();
10
11     // variable lookup
12     syn lazy VarDecl VarName.decl() = lookupVar(getName());
13
14     // block structured lookup
15     inh lazy VarDecl VarName.lookupVar(String name);
16     inh lazy VarDecl Block.lookupVar(String name);
17     inh lazy VarDecl FunctionDeclaration.lookupVar(String name);
18
19     // handle local declarations
20     eq Block.getStmt(int j).lookupVar(String name) {
21         for(int i=0;i<j;++i)
22             if(getStmt(i) instanceof VarDeclStmt) {
23                 VarDecl vd = ((VarDeclStmt)getStmt(i)).getVarDecl();
24                 if(vd.getName().equals(name))
25                     return vd;
26             }
27         return this.lookupVar(name);
28     }
29
30     // handle parameter declarations
31     eq FunctionDeclaration.getChild().lookupVar(String name) {
32         for(Parameter parm : getParameters())
33             if(parm.getName().equals(name))
34                 return parm;
35         return this.lookupVar(name);
36     }
37
38     // handle field declarations
39     eq Module.getChild().lookupVar(String name) {
40         // first try local lookup
41         FieldDeclaration decl = lookupLocalField(name);
42         if(decl != null)
43             return decl.getVarDecl();
44
45         // otherwise try lookup imported fields
46         for(Import imp : getImports()) {
47             decl = imp.resolve().lookupLocalField(name);
48             // import only if it is public
```

```
49              if(decl != null && decl.isPublic())
50                  return decl.getVarDecl();
51          }
52          return null;
53      }
54
55      // look up local field inside module
56      syn lazy FieldDeclaration Module.lookupLocalField(String name) {
57          for(Declaration decl : getDeclarations()) {
58              if(decl instanceof FieldDeclaration) {
59                  VarDecl vd = ((FieldDeclaration)decl).getVarDecl();
60                  if(vd.getName().equals(name))
61                      return (FieldDeclaration)decl;
62              }
63          }
64          return null;
65      }
66
67      // module lookup
68      syn lazy Module Program.resolveModule(String qualifiedName) {
69          for(Module module : getModules())
70              if(module.getQualifiedName().equals(qualifiedName))
71                  return module;
72          return null;
73      }
74
75      syn lazy Module Import.resolve() = getProgram().resolveModule(
    getQualifiedName());
76
77      // type lookup
78      inh lazy TypeDeclaration UserTypeName.lookupType(String name);
79
80      // lookup types in module
81      eq Module.getChild().lookupType(String name) {
82          TypeDeclaration tp = lookupLocalType(name);
83          if(tp != null)
84              return tp;
85
86          for(Import imp : getImports()) {
87              tp = imp.resolve().lookupLocalType(name);
88              if(tp != null && tp.isPublic())
89                  return tp;
90          }
91          return null;
92      }
93
94      // lookup local types
95      syn lazy TypeDeclaration Module.lookupLocalType(String name) {
96          for(Declaration decl : getDeclarations())
97              if(decl instanceof TypeDeclaration && ((TypeDeclaration)
```

```
 97 decl).getName().equals(name))
 98                 return (TypeDeclaration)decl;
 99         return null;
100     }
101 
102     // function lookup
103     syn FunctionDeclaration FunctionName.decl() = lookupFunction(
    getName());
104 
105     inh lazy FunctionDeclaration FunctionName.lookupFunction(String
    name);
106     eq Module.getChild().lookupFunction(String name) {
107         FunctionDeclaration fn = lookupLocalFunction(name);
108         if(fn != null)
109             return fn;
110 
111         for(Import imp : getImports()) {
112             fn = imp.resolve().lookupLocalFunction(name);
113             if(fn != null && fn.isPublic())
114                 return fn;
115         }
116         return null;
117     }
118 
119     syn lazy FunctionDeclaration Module.lookupLocalFunction(String
    name) {
120         for(Declaration decl : getDeclarations())
121             if(decl instanceof FunctionDeclaration && ((
    FunctionDeclaration)decl).getName().equals(name))
122                 return (FunctionDeclaration)decl;
123         return null;
124     }
125 
126     syn lazy FunctionDeclaration Call.getCallTarget() = getCallee().
    decl();
127 
128     syn boolean Declaration.isPublic() = getAccessibility().getPublic
    ();
129 }
```

```
 1 /** Type inference. */
 2 aspect types {
 3     /** Singleton instances of types int, boolean and void. */
 4     public static final IntType TypeDescriptor.INT = new IntType();
 5     public static final BooleanType TypeDescriptor.BOOLEAN = new
   BooleanType();
 6     public static final VoidType TypeDescriptor.VOID = new VoidType();
 7 
 8     /** Lazily create array type for an existing type. */
 9     syn lazy ArrayType TypeDescriptor.arrayType() = new ArrayType(this
   );
10 
11     /** Type descriptors for Java types. */
12     private static final HashMap<String, TypeDescriptor>
   TypeDescriptor.javaTypeDescriptors = new HashMap<String,
   TypeDescriptor>();
13     public static final TypeDescriptor TypeDescriptor.forJavaType(
   String name) {
14         TypeDescriptor desc = javaTypeDescriptors.get(name);
15         if(desc == null)
16             javaTypeDescriptors.put(name, desc = new JavaType(name));
17         return desc;
18     }
19 
20     /** Determine type descriptor for type name. */
21     syn lazy TypeDescriptor TypeName.getDescriptor();
22     eq IntTypeName.getDescriptor() = TypeDescriptor.INT;
23     eq BooleanTypeName.getDescriptor() = TypeDescriptor.BOOLEAN;
24     eq VoidTypeName.getDescriptor() = TypeDescriptor.VOID;
25     eq ArrayTypeName.getDescriptor() {
26         TypeDescriptor desc = getElementType().getDescriptor();
27         return desc == null ? null : desc.arrayType();
28     }
29     eq UserTypeName.getDescriptor() {
30         TypeDeclaration decl = lookupType(getName());
31         return decl == null ? null : TypeDescriptor.forJavaType(decl.
   getJavaType());
32     }
33     eq JavaTypeName.getDescriptor() = TypeDescriptor.forJavaType(
   getName());
34 
35     /** Type inference for parameters. */
36     syn TypeDescriptor Parameter.type() = getTypeName().getDescriptor(
   );
37 
38     /** Type inference for expressions. */
39     syn lazy TypeDescriptor Expr.type();
40 
41     eq VarName.type() = decl().getTypeName().getDescriptor();
42     eq ArrayIndex.type() = ((ArrayType)getBase().type()).
```

```
42 getElementType();
43 
44     eq Call.type() = getCallTarget().getReturnType().getDescriptor();
45     eq Assignment.type() = getLHS().type();
46     eq BinaryExpr.type() = TypeDescriptor.INT;
47     eq CompExpr.type() = TypeDescriptor.BOOLEAN;
48     eq NegExpr.type() = TypeDescriptor.INT;
49     eq StringLiteral.type() = TypeDescriptor.forJavaType("java.lang.
   String");
50     eq IntLiteral.type() = TypeDescriptor.INT;
51     eq BooleanLiteral.type() = TypeDescriptor.BOOLEAN;
52     eq ArrayLiteral.type() = getElement(0).type().arrayType();
53 }
```

```
 1 import java.util.ArrayList;
 2
 3 /** Utility attributes for error reporting. */
 4 aspect Errors {
 5     /** Utility class for representing compiler errors. */
 6     public class CompilerError {
 7         private final String msg;
 8         private final int line, column;
 9
10         public CompilerError(String msg, int line, int column) {
11             this.msg = msg;
12             this.line = line;
13             this.column = column;
14         }
15
16         @Override
17         public String toString() {
18             return "Line " + line + ", column " + column + ": " + msg;
19         }
20     }
21
22     /** Compiler errors are stored in a list at the program root. */
23     private ArrayList<CompilerError> Program.errors = new ArrayList<
   CompilerError>();
24
25     /** Enter a new error into the list. */
26     protected void Program.error(String msg, int line, int column) {
27         errors.add(new CompilerError(msg, line, column));
28     }
29
30     /** Report an error from some node within the AST. */
31     public void ASTNode.error(String msg) {
32         getProgram().error(msg, getLine(getStart()), getColumn(
   getStart()));
33     }
34
35     /** Provide access to the list of compiler errors. */
36     public Iterable<CompilerError> Program.getErrors() {
37         return errors;
38     }
39
40     /** Check whether any errors have been reported. */
41     public boolean Program.hasErrors() {
42         return !errors.isEmpty();
43     }
44 }
```

```
Program ::= Module*;

Module ::= <Package:String> <Name:String> Import* Declaration*;

Import ::= <Package:String> <Name:String>;

abstract Declaration ::= Accessibility;

Accessibility ::= <Public:Boolean>;

FunctionDeclaration : Declaration ::= ReturnType:TypeName <Name:String
> Parameter* Body:Block;
FieldDeclaration : Declaration ::= VarDecl;
TypeDeclaration : Declaration ::= <Name:String> <JavaType:String>;

VarDecl ::= TypeName <Name:String>;
LocalVarDecl : VarDecl;
Parameter : VarDecl;

abstract TypeName;
IntTypeName : TypeName;
BooleanTypeName : TypeName;
VoidTypeName : TypeName;
ArrayTypeName : TypeName ::= ElementType:TypeName;
UserTypeName : TypeName ::= <Name:String>;
JavaTypeName : TypeName ::= <Name:String>;

abstract Stmt;

ExprStmt : Stmt ::= Expr;
VarDeclStmt : Stmt ::= VarDecl;
Block : Stmt ::= Stmt*;
IfStmt : Stmt ::= Expr Then:Stmt [Else:Stmt];
WhileStmt : Stmt ::= Expr Body:Stmt;
ReturnStmt : Stmt ::= [Expr];
BreakStmt : Stmt;

abstract Expr;

abstract LHSExpr : Expr;
VarName : LHSExpr ::= <Name:String>;
ArrayIndex : LHSExpr ::= Base:Expr Index:Expr;

Call : Expr ::= Callee:FunctionName Argument:Expr*;
Assignment : Expr ::= LHS:LHSExpr RHS:Expr;
abstract BinaryExpr : Expr ::= Left:Expr Right:Expr;
AddExpr : BinaryExpr;
SubExpr : BinaryExpr;
MulExpr : BinaryExpr;
DivExpr : BinaryExpr;
```

```
50 ModExpr : BinaryExpr;
51 abstract UnaryExpr : Expr ::= Operand:Expr;
52 NegExpr : UnaryExpr;
53
54 abstract CompExpr : BinaryExpr;
55 EqExpr : CompExpr;
56 NeqExpr : CompExpr;
57 abstract ArithCompExpr : CompExpr;
58 LtExpr : ArithCompExpr;
59 GtExpr : ArithCompExpr;
60 LeqExpr : ArithCompExpr;
61 GeqExpr : ArithCompExpr;
62
63 abstract Literal : Expr;
64 StringLiteral : Literal ::= <Value:String>;
65 IntLiteral : Literal ::= <Value:Integer>;
66 BooleanLiteral : Literal ::= <Value:Boolean>;
67 ArrayLiteral : Literal ::= Element:Expr*;
68
69 FunctionName ::= <Name:String>;
70
71 abstract TypeDescriptor;
72 IntType : TypeDescriptor;
73 BooleanType : TypeDescriptor;
74 VoidType : TypeDescriptor;
75 ArrayType : TypeDescriptor ::= ElementType:TypeDescriptor;
76 JavaType : TypeDescriptor ::= <Name:String>;
```

```
 1 /** Utility attributes for navigating the AST. */
 2 aspect ASTUtil {
 3     // find the innermost enclosing while statement; return null if
   there isn't one
 4     inh WhileStmt Stmt.getEnclosingLoop();
 5     eq FunctionDeclaration.getChild().getEnclosingLoop() = null;
 6     eq WhileStmt.getBody().getEnclosingLoop() = this;
 7 
 8     // find the AST's root node
 9     protected Program ASTNode.getProgram() {
10         if(getParent() == null)
11             return null;
12         return getParent().getProgram();
13     }
14     protected Program Program.getProgram() {
15         return this;
16     }
17 
18     // common interface for scope-creating AST nodes
19     interface Scope {}
20     Module implements Scope;
21     Block implements Scope;
22     FunctionDeclaration implements Scope;
23 
24     // get the scope of a variable declaration
25     inh Scope VarDecl.getScope();
26     eq Scope.getChild().getScope() = this;
27 
28     // get the module a function is declared in
29     inh Module FunctionDeclaration.getModule();
30     eq Module.getChild().getModule() = this;
31 }
```

```
 1 /** The runtime module. */
 2 aspect Runtime {
 3     public static TypeName TypeName.forClass(Class<?> klass) {
 4         String name = klass.getCanonicalName();
 5         if("void".equals(name))
 6             return new VoidTypeName();
 7         else if("int".equals(name))
 8             return new IntTypeName();
 9         else if("boolean".equals(name))
10             return new BooleanTypeName();
11         else
12             return new JavaTypeName(name);
13     }
14 }
```

```
 1 import java.util.HashSet;
 2 import java.util.Set;
 3
 4 /** Attributes for checking whether the program has any name or scope
   errors.
 5  *  Most of the definitions are just traversal methods that recurse
   over the children. */
 6 aspect Namecheck {
 7     public void Program.namecheck() {
 8         // check for name clashes on modules
 9         Set<String> module_names = new HashSet<String>();
10         for(Module module : getModules()) {
11             if(!module_names.add(module.getQualifiedName()))
12                 error("Multiple modules with name " + module.
   getQualifiedName());
13             module.namecheck();
14         }
15     }
16
17     public void Module.namecheck() {
18         // check for duplicate imports
19         Set<String> imports = new HashSet<String>();
20         for(Import imp : getImports()) {
21             imp.namecheck();
22             if(!imports.add(imp.getQualifiedName()))
23                 error("Multiple imports of module " + imp.
   getQualifiedName());
24         }
25
26         for(Declaration decl : getDeclarations())
27             decl.namecheck();
28     }
29
30     inh Module Import.getModule();
31     public void Import.namecheck() {
32         if(resolve() == null)
33             error("Module cannot be resolved: " + getQualifiedName());
34         if(getQualifiedName().equals(getModule().getQualifiedName()))
35             error("Modules cannot import themselves.");
36     }
37
38     public abstract void Declaration.namecheck();
39
40     inh FunctionDeclaration FunctionDeclaration.lookupFunction(String
   name);
41     public void FunctionDeclaration.namecheck() {
42         if(lookupFunction(getName()) != this)
43             error("Multiple declarations for function " + getName());
44
45         getReturnType().namecheck();
```

```
46          for(Parameter parm : getParameters())
47              parm.namecheck();
48          getBody().namecheck();
49      }
50
51      public void FieldDeclaration.namecheck() {
52          getVarDecl().namecheck();
53      }
54
55      inh TypeDeclaration TypeDeclaration.lookupType(String name);
56      public void TypeDeclaration.namecheck() {
57          if(lookupType(getName()) != this)
58              error("Multiple declarations for type " + getName());
59      }
60
61      inh VarDecl VarDecl.lookupVar(String name);
62      public void VarDecl.namecheck() {
63          getTypeName().namecheck();
64
65          // check that there aren't two variables with the same name in
    the same scope
66          VarDecl decl = lookupVar(getName());
67          if(decl != null && decl != this && decl.getScope() == this.
   getScope())
68              error("Multiple declarations for " + getName() + " in same
    scope");
69      }
70
71      public void TypeName.namecheck() {}
72      public void UserTypeName.namecheck() {
73          if(lookupType(getName()) == null)
74              error("Type " + getName() + " could not be resolved.");
75      }
76
77      public abstract void Stmt.namecheck();
78
79      public void Block.namecheck() {
80          for(Stmt stmt : getStmts())
81              stmt.namecheck();
82      }
83
84      public void BreakStmt.namecheck() {
85          if(getEnclosingLoop() == null)
86              error("Break statement outside loop.");
87      }
88
89      public void ExprStmt.namecheck() {
90          getExpr().namecheck();
91      }
92
```

```
 93     public void IfStmt.namecheck() {
 94         getExpr().namecheck();
 95         getThen().namecheck();
 96         if(hasElse())
 97             getElse().namecheck();
 98     }
 99 
100     public void ReturnStmt.namecheck() {
101         if(hasExpr())
102             getExpr().namecheck();
103     }
104 
105     public void VarDeclStmt.namecheck() {
106         getVarDecl().namecheck();
107     }
108 
109     public void WhileStmt.namecheck() {
110         getExpr().namecheck();
111         getBody().namecheck();
112     }
113 
114     public abstract void Expr.namecheck();
115 
116     public void VarName.namecheck() {
117         if(decl() == null)
118             error("Variable " + getName() + " cannot be resolved.");
119     }
120 
121     public void ArrayIndex.namecheck() {
122         getBase().namecheck();
123         getIndex().namecheck();
124     }
125 
126     public void Call.namecheck() {
127         getCallee().namecheck();
128         for(Expr argument : getArguments())
129             argument.namecheck();
130     }
131 
132     public void Assignment.namecheck() {
133         getLHS().namecheck();
134         getRHS().namecheck();
135     }
136 
137     public void BinaryExpr.namecheck() {
138         getLeft().namecheck();
139         getRight().namecheck();
140     }
141 
142     public void UnaryExpr.namecheck() {
```

```
143         getOperand().namecheck();
144     }
145 
146     public void Literal.namecheck() {}
147 
148     public void ArrayLiteral.namecheck() {
149         for(Expr element : getElements())
150             element.namecheck();
151     }
152 
153     public void FunctionName.namecheck() {
154         if(decl() == null)
155             error("Function " + getName() + " cannot be resolved.");
156     }
157 }
```

```
 1 /** Type checking. */
 2 aspect Typecheck {
 3     // some convenience attributes
 4     syn boolean TypeDescriptor.isArrayType() = false;
 5     eq ArrayType.isArrayType() = true;
 6 
 7     syn boolean TypeDescriptor.isBoolean() = false;
 8     eq BooleanType.isBoolean() = true;
 9 
10     syn boolean TypeDescriptor.isNumeric() = false;
11     eq IntType.isNumeric() = true;
12 
13     syn boolean TypeDescriptor.isVoid() = false;
14     eq VoidType.isVoid() = true;
15 
16     public void Program.typecheck() {
17         for(Module module : getModules())
18             module.typecheck();
19     }
20 
21     public void Module.typecheck() {
22         for(Declaration decl : getDeclarations())
23             decl.typecheck();
24     }
25 
26     public void Declaration.typecheck() {}
27 
28     public void FunctionDeclaration.typecheck() {
29         getReturnType().typecheck();
30         for(Parameter parm : getParameters())
31             parm.typecheck();
32         getBody().typecheck();
33     }
34 
35     public void FieldDeclaration.typecheck() {
36         getVarDecl().typecheck();
37     }
38 
39     // check that no variable is declared to be of type 'void'
40     public void VarDecl.typecheck() {
41         getTypeName().typecheck();
42         if(getTypeName().getDescriptor().isVoid())
43             error("Cannot declare variable of type void.");
44     }
45 
46     public void Stmt.typecheck() {}
47 
48     public void VarDeclStmt.typecheck() {
49         getVarDecl().typecheck();
50     }
```

```
51
52      public void Block.typecheck() {
53          for(Stmt stmt : getStmts())
54              stmt.typecheck();
55      }
56
57      public void ExprStmt.typecheck() {
58          getExpr().typecheck();
59      }
60
61      public void IfStmt.typecheck() {
62          getExpr().typecheck();
63          getThen().typecheck();
64          if(hasElse())
65              getElse().typecheck();
66
67          // check that if condition is 'boolean'
68          if(!getExpr().type().isBoolean())
69              error("If condition is not of type boolean.");
70      }
71
72      inh FunctionDeclaration Stmt.getFunction();
73      eq FunctionDeclaration.getChild().getFunction() = this;
74
75      // check that return statement returns expression of right type
76      public void ReturnStmt.typecheck() {
77          if(!hasExpr()) {
78              if (!getFunction().getReturnType().getDescriptor().isVoid
    ())
79                  error("Return expression missing.");
80          } else  {
81              getExpr().typecheck();
82              if(getExpr().type() != getFunction().getReturnType().
    getDescriptor())
83                  error("Return statement returns expression of wrong
    type.");
84          }
85      }
86
87      // check that loop condition is not of type void
88      public void WhileStmt.typecheck() {
89          getExpr().typecheck();
90          getBody().typecheck();
91
92          if(!getExpr().type().isBoolean())
93
94          error("Loop condition is not of type boolean.");
95      }
96
97      public abstract void Expr.typecheck();
```

```
 98
 99     public void VarName.typecheck() {
100     }
101
102     // check that base expression is array, and index expression is
    integer
103     public void ArrayIndex.typecheck() {
104         getBase().typecheck();
105         getIndex().typecheck();
106
107         if(!getBase().type().isArrayType())
108             error("Base expression of array index should be of array
    type.");
109         if(!getIndex().type().isNumeric())
110             error("Index expression of array index should be of
    integer type.");
111     }
112
113     // typecheck function call
114     public void Call.typecheck() {
115         FunctionDeclaration callee = getCallTarget();
116         for(int i=0;i<getNumArgument(); i++)
117             getArgument(i).typecheck();
118         if (getNumArgument() != callee.getNumParameter()) {
119             error("Number of arguments (" + getNumArgument() + ") and
     number of " +
120                     "parameters (" + callee.getNumParameter() + ") do
     not match.");
121         } else {
122             for(int i=0;i<getNumArgument(); i++)
123                 if(getArgument(i).type() != callee.getParameter(i).
    type())
124                     error("The " + i + "th argument has the wrong
    type.");
125         }
126     }
127
128     // check assignment compatibility
129     public void Assignment.typecheck() {
130         getLHS().typecheck();
131         getRHS().typecheck();
132         if(getLHS().type() != getRHS().type())
133             error("The two sides of the assignment have different
    types.");
134     }
135
136     // check that operands of binary expression are numeric
137     public void BinaryExpr.typecheck() {
138         getLeft().typecheck();
139         getRight().typecheck();
```

```
140         if(!getLeft().type().isNumeric() || !getRight().type().
    isNumeric())
141             error("Both operands of a binary arithmetic operator must
     have numeric type.");
142     }
143 
144     // check types of comparison operands
145     public void CompExpr.typecheck() {
146         getLeft().typecheck();
147         getRight().typecheck();
148         if(getLeft().type() != getRight().type())
149             error("Both operands of a comparison operator must be of
    the same type.");
150         if(getLeft().type().isVoid())
151             error("Operands in comparison may not be of type void.");
152     }
153 
154     public void ArithCompExpr.typecheck() {
155         getLeft().typecheck();
156         getRight().typecheck();
157         if(!getLeft().type().isNumeric())
158             error("Operands of arithmetic comparison must be numeric
    .");
159         if(!getRight().type().isNumeric())
160             error("Operands of arithmetic comparison must be numeric
    .");
161 
162     }
163 
164     public void UnaryExpr.typecheck() {
165         getOperand().typecheck();
166         if(!getOperand().type().isNumeric())
167             error("The operand of a unary arithmetic operator must
    have numeric type.");
168     }
169 
170     public void Literal.typecheck() {}
171 
172     public void ArrayLiteral.typecheck() {
173         if(getNumElement() == 0) {
174             error("Array literals must contain at least one element."
    );
175         } else {
176             if(getElement(0).type().isVoid())
177                 error("Array literal elements may not be of type void
    .");
178             for(int i=0;i<getNumElement();++i) {
179                 getElement(i).typecheck();
180                 if(i > 0 && getElement(i).type() != getElement(0).
    type())  /* one dimensional array assumed  */
```

```
181                     error("Every element in an array literal must
    have the same type.");
182             }
183         }
184     }
185 
186     public void TypeName.typecheck() {}
187     public void ArrayTypeName.typecheck() {
188         if(getElementType().getDescriptor().isVoid())
189             error("Component type of array cannot be void.");
190     }
191 }
```