```
 1 package backend;
 2
 3 import java.util.ArrayList;
 4
 5 import soot.Local;
 6 import soot.Scene;
 7 import soot.SootClass;
 8 import soot.SootMethodRef;
 9 import soot.Type;
10 import soot.Unit;
11 import soot.Value;
12 import soot.jimple.Constant;
13 import soot.jimple.IntConstant;
14 import soot.jimple.Jimple;
15 import soot.jimple.NopStmt;
16 import soot.jimple.StaticInvokeExpr;
17 import soot.jimple.StringConstant;
18 import soot.util.Chain;
19 import ast.AddExpr;
20 import ast.ArrayIndex;
21 import ast.ArrayLiteral;
22 import ast.Assignment;
23 import ast.BinaryExpr;
24 import ast.BooleanLiteral;
25 import ast.Call;
26 import ast.CompExpr;
27 import ast.DivExpr;
28 import ast.EqExpr;
29 import ast.Expr;
30 import ast.FunctionDeclaration;
31 import ast.GeqExpr;
32 import ast.GtExpr;
33 import ast.IntLiteral;
34 import ast.LeqExpr;
35 import ast.LtExpr;
36 import ast.ModExpr;
37 import ast.Module;
38 import ast.MulExpr;
39 import ast.NegExpr;
40 import ast.NeqExpr;
41 import ast.Parameter;
42 import ast.StringLiteral;
43 import ast.SubExpr;
44 import ast.VarDecl;
45 import ast.VarName;
46 import ast.Visitor;
47
48 /**
49  * This class is in charge of creating Jimple code for a given expression (and its
    nested
50  * expressions, if applicable).
51  */
52 public class ExprCodeGenerator extends Visitor<Value> {
53     /** The {@link FunctionCodeGenerator} that instantiated this object. */
54     private final FunctionCodeGenerator fcg;
```

```
 55
 56     /** We cache the statement list of the enclosing function for convenience. */
 57     private final Chain<Unit> units;
 58
 59     private ExprCodeGenerator(FunctionCodeGenerator fcg) {
 60         this.fcg = fcg;
 61         this.units = fcg.getBody().getUnits();
 62     }
 63
 64     /**
 65      * Ensures that the given value can be used as an operand; that is, if the
 66      * value is not a {@link Local} or a {@link Constant}, this method allocates
 67      * a new temporary variable and stores the value into that temporary.
 68      */
 69     private Value wrap(Value v) {
 70         if(v == null || v instanceof Local || v instanceof Constant) {
 71             return v;
 72         } else {
 73             Local temp = fcg.mkTemp(v.getType());
 74             units.add(Jimple.v().newAssignStmt(temp, v));
 75             return temp;
 76         }
 77     }
 78
 79     /**
 80      * Convenience method to generate code for an expression and wrap it.
 81      */
 82     public static Value generate(Expr expr, FunctionCodeGenerator fcg) {
 83         ExprCodeGenerator gen = new ExprCodeGenerator(fcg);
 84         return gen.wrap(expr.accept(gen));
 85     }
 86
 87     /** Generate code for an assignment. */
 88     @Override
 89     public Value visitAssignment(Assignment nd) {
 90         // note that the left hand side should _not_ be wrapped!
 91         Value lhs = nd.getLHS().accept(this),
 92               rhs = wrap(nd.getRHS().accept(this));
 93         units.add(Jimple.v().newAssignStmt(lhs, rhs));
 94         return rhs;
 95     }
 96
 97     /** Generate code for an integer literal. */
 98     @Override
 99     public Value visitIntLiteral(IntLiteral nd) {
100         /* TODO: return something meaningful here */
101         return IntConstant.v(nd.getValue());
102     }
103
104     /** Generate code for a string literal. */
105     @Override
106     public Value visitStringLiteral(StringLiteral nd) {
107         /* TODO: return something meaningful here */
108         return StringConstant.v(nd.getValue());
109     }
```

```
110
111     /** Generate code for a Boolean literal. */
112     @Override
113     public Value visitBooleanLiteral(BooleanLiteral nd) {
114         /* TODO: return something meaningful here (hint: translate 'true' to
    integer
115          *       constant 1, 'false' to integer constant 0) */
116         if (nd.getValue()) {
117             return IntConstant.v(1);
118         }
119         return IntConstant.v(0);
120     }
121
122     /** Generate code for an array literal. */
123     @Override
124     public Value visitArrayLiteral(ArrayLiteral nd) {
125         Type elttp = SootTypeUtil.getSootType(nd.getElement(0).type());
126         // create a new array with the appropriate number of elements
127         Value array = wrap(Jimple.v().newNewArrayExpr(elttp, IntConstant.v(nd.
    getNumElement())));
128         for(int i=0;i<nd.getNumElement();++i) {
129             // generate code to store the individual expressions into the
    elements of the array
130             Value elt = wrap(nd.getElement(i).accept(this));
131             units.add(Jimple.v().newAssignStmt(Jimple.v().newArrayRef(array,
    IntConstant.v(i)), elt));
132         }
133         return array;
134     }
135
136     /** Generate code for an array index expression. */
137     @Override
138     public Value visitArrayIndex(ArrayIndex nd) {
139         /* TODO: generate code for array index */
140         Value index = wrap(nd.getIndex().accept(this));
141         Value base = wrap(nd.getBase().accept(this));
142         return Jimple.v().newArrayRef(base, index);
143     }
144
145     /** Generate code for a variable name. */
146     @Override
147     public Value visitVarName(VarName nd) {
148         VarDecl decl = nd.decl();
149         // determine whether this name refers to a local or to a field
150         if(decl.isLocal()) {
151             return fcg.getSootLocal(decl);
152         } else {
153             SootClass declaringClass = fcg.getModuleCodeGenerator().
    getProgramCodeGenerator().getSootClass(decl.getModule());
154             Type fieldType = SootTypeUtil.getSootType(decl.getTypeName().
    getDescriptor());
155             return Jimple.v().newStaticFieldRef(Scene.v().makeFieldRef(
    declaringClass, decl.getName(), fieldType, true));
156         }
157     }
```

```
158
159     /** Generate code for a binary expression. */
160     @Override
161     public Value visitBinaryExpr(BinaryExpr nd) {
162         /* TODO: generate code for binary expression here; you can either use a
    visitor
163          *       to determine the type of binary expression you are dealing with
    , or
164          *       generate code in the more specialised visitor methods
    visitAddExpr,
165          *       visitSubExpr, etc., instead
166          */
167
168         final Value lhs = wrap(nd.getLeft().accept(this));
169         final Value rhs = wrap(nd.getRight().accept(this));
170         Value res = nd.accept(new Visitor<Value>() {
171             @Override
172             public Value visitAddExpr(AddExpr nd) {
173                 return Jimple.v().newAddExpr(lhs, rhs);
174             }
175             @Override
176             public Value visitSubExpr(SubExpr nd) {
177                 return Jimple.v().newSubExpr(lhs, rhs);
178             }
179             @Override
180             public Value visitMulExpr(MulExpr nd) {
181                 return Jimple.v().newMulExpr(lhs, rhs);
182             }
183             @Override
184             public Value visitDivExpr(DivExpr nd) {
185                 return Jimple.v().newDivExpr(lhs, rhs);
186             }
187             @Override
188             public Value visitModExpr(ModExpr nd) {
189                 return Jimple.v().newRemExpr(lhs, rhs);
190             }
191             @Override
192             public Value visitNegExpr(NegExpr nd) {
193                 return Jimple.v().newNegExpr(lhs);
194             }
195         });
196         return res;
197     }
198
199
200     /** Generate code for a comparison expression. */
201     @Override
202     public Value visitCompExpr(CompExpr nd) {
203         final Value left = wrap(nd.getLeft().accept(this)),
204                     right = wrap(nd.getRight().accept(this));
205         Value res = nd.accept(new Visitor<Value>() {
206             @Override
207             public Value visitEqExpr(EqExpr nd) {
208                 return Jimple.v().newEqExpr(left, right);
209             }
```

```
210             @Override
211             public Value visitNeqExpr(NeqExpr nd) {
212                 return Jimple.v().newNeExpr(left, right);
213             }
214             @Override
215             public Value visitLtExpr(LtExpr nd) {
216                 return Jimple.v().newLtExpr(left, right);
217             }
218             @Override
219             public Value visitGtExpr(GtExpr nd) {
220                 return Jimple.v().newGtExpr(left, right);
221             }
222             @Override
223             public Value visitLeqExpr(LeqExpr nd) {
224                 return Jimple.v().newLeExpr(left, right);
225             }
226             @Override
227             public Value visitGeqExpr(GeqExpr nd) {
228                 return Jimple.v().newGeExpr(left, right);
229             }
230         });
231         // compute a result of 0 or 1 depending on the truth value of the
    expression
232         Local resvar = fcg.mkTemp(SootTypeUtil.getSootType(nd.type()));
233         units.add(Jimple.v().newAssignStmt(resvar, IntConstant.v(1)));
234         NopStmt join = Jimple.v().newNopStmt();
235         units.add(Jimple.v().newIfStmt(res, join));
236         units.add(Jimple.v().newAssignStmt(resvar, IntConstant.v(0)));
237         units.add(join);
238         return resvar;
239     }
240
241     /** Generate code for a negation expression. */
242     @Override
243     public Value visitNegExpr(NegExpr nd) {
244         /* TODO: generate code for negation expression */
245         return Jimple.v().newNegExpr(wrap(nd.getOperand().accept(this)));
246     }
247
248     /** Generate code for a function call. */
249     @Override
250     public Value visitCall(Call nd) {
251         String calleeName = nd.getCallee().getName();
252         FunctionDeclaration calleeDecl = nd.getCallTarget();
253         Module calleeModule = calleeDecl.getModule();
254         ArrayList<Type> parmTypes = new ArrayList<Type>(calleeDecl.
    getNumParameter());
255         for(Parameter parm : calleeDecl.getParameters())
256             parmTypes.add(SootTypeUtil.getSootType(parm.type()));
257         Type rettp = SootTypeUtil.getSootType(calleeDecl.getReturnType().
    getDescriptor());
258
259         // compute reference to callee
260         SootClass calleeSootClass = fcg.getModuleCodeGenerator().
    getProgramCodeGenerator().getSootClass(calleeModule);
```

```
261         SootMethodRef callee = Scene.v().makeMethodRef(calleeSootClass,
    calleeName, parmTypes, rettp, true);
262
263         // prepare arguments
264         Value[] args = new Value[nd.getNumArgument()];
265         for(int i=0;i<args.length;++i)
266             args[i] = wrap(nd.getArgument(i).accept(this));
267
268         // assemble invoke expression
269         StaticInvokeExpr invk = Jimple.v().newStaticInvokeExpr(callee, args);
270
271         // decide what to do with the result
272         if(rettp == soot.VoidType.v()) {
273             units.add(Jimple.v().newInvokeStmt(invk));
274             return null;
275         } else {
276             Local res = fcg.mkTemp(rettp);
277             units.add(Jimple.v().newAssignStmt(res, invk));
278             return res;
279         }
280     }
281 }
```