Sample Type-Checking Routines for CSX-Lite

```
// abstract superclass; only
subclasses are actually created
abstract class ASTNode {
// Total number of type errors found
static int typeErrors =  0;


static void typeMustBe(int testType,
 int requiredType,String errorMsg) {
 if ((testType != Types.Error) &&
(testType != requiredType)) {
  System.out.println(errorMsg);
  typeErrors++;
  }
}
```

```
static void typesMustBeEqual(
int type1,int type2,String errorMsg)
{
 if ((type1 != Types.Error) &&
     (type2 != Types.Error) &&
     (type1 != type2)) {

System.out.println(errorMsg);
typeErrors++;
}}

String error() {
 return “Error (line “ + linenum +
   “): “;}

public static SymbolTable st =
  new SymbolTable();

void checkTypes(){};
// This will normally need to be
// redefined in a subclass
```

```
// This node is used to root only
// CSX lite programs
class csxLiteNode extends ASTNode {

  void checkTypes(){
    fields.checkTypes();
    progStmts.checkTypes();
  }
  boolean isTypeCorrect() {
        checkTypes();
        return (typeErrors == 0);
  };

  private stmtsNode progStmts;
  private fieldDeclsNode fields;
};
```

```
// Root of all ASTs for CSX
class classNode extends ASTNode {
  // You need to refine this one
  boolean isTypeCorrect() {
    return true;};

  private identNodeclassName;
  private memberDeclsNodemembers;
};


class fieldDeclsNode extends ASTNode
{
  void checkTypes() {
        thisField.checkTypes();
        moreFields.checkTypes();
  };

  private declNodethisField;
  private fieldDeclsNode moreFields;
};
```

```
class nullFieldDeclsNode extends
fieldDeclsNode {
   void checkTypes(){};
};


class varDeclNode extends declNode {
   void checkTypes() {
     SymbolInfo     id;
     id = (SymbolInfo)
     st.localLookup(varName.idname);
     if (id != null) {
       System.out.println(error() +
        id.name()+
        " is already declared.");
        typeErrors++;
        varName.type =
           new Types(Types.Error);
     } else {
      id =
      new SymbolInfo(varName.idname,
         new Kinds(Kinds.Var),
         varType.type);
      varName.type = varType.type;
      try {
        st.insert(id);
```

```
      } catch (DuplicateException d)
           { /* can't happen */ }
        catch (EmptySTException e)
           { /* can't happen */ }
      varName.idinfo=id;
  }
};
  privateidentNodevarName;
  privatetypeNode varType;
  privateexprNode initValue;
};


abstract class typeNode extends
ASTNode {
// abstract superclass; only
// subclasses are actually created
 Types   type;
 // Used for typechecking
 // -- the type of this typeNode
};
```

```
class intTypeNode extends typeNode {
  intTypeNode(int line, int col){
     super(line,col, new
     Types(Types.Integer));
  }
  void checkTypes() {
   //       No type checking needed
  }
};


class stmtsNode extends ASTNode {
  void checkTypes() {
    thisStmt.checkTypes();
    moreStmts.checkTypes();
  };

  private stmtNodethisStmt;
  private stmtsNode moreStmts;
};
```

```
class nullStmtsNode extends
stmtsNode {
   void checkTypes(){};
};

class asgNode extends stmtNode {
void checkTypes() {
  target.checkTypes();
  source.checkTypes();
  //In CSX-lite all IDs are vars!
  assert(target.kind.val ==
          Kinds.Var);
  typesMustBeEqual(source.type.val,
                   target.type.val,
     error() +
     “Both the left and right” +
“ hand sides of an assignment must “
    + “have the same type.”);
  }

  private nameNodetarget;
  private exprNode source;
};
```

```
class ifThenNode extends stmtNode {
  void checkTypes() {
    condition.checkTypes();
    typeMustBe(condition.type.val,
                Types.Boolean,
      error() +
      “The control expression of an”
      + “ if must be a bool.”);
     thenPart.checkTypes();
     // No else parts in CSX Lite
  };
  private exprNode condition;
  private stmtNode thenPart;
  private stmtNode elsePart;
};
```

```
class printNode extends stmtNode {
  void checkTypes() {
  outputValue.checkTypes();
  typeMustBe(outputValue.type.val,
             Types.Integer,
     error() +
 “Only int values may be printed.”);
   };
  private exprNode outputValue;
  private printNode morePrints;
};

// abstract superclass;
// only subclasses are actually
//created
abstract class exprNode extends
ASTNode {
  protected Types   type;
  // Used for typechecking:
  //  the type of this node
  protected Kinds   kind;
  // Used for typechecking:
  // the kind of this node
};
```

```
class binaryOpNode extends exprNode
{
  void checkTypes() {
   //Only two bin ops in CSX-lite
   assert(operatorCode== sym.PLUS
    ||operatorCode==sym.MINUS);
   leftOperand.checkTypes();
   rightOperand.checkTypes();
   type = new Types(Types.Integer);

   typeMustBe(leftOperand.type.val,
              Types.Integer,
      error() +
      "Left operand of" +
      toString(operatorCode)
      +  "must be an int.");

  typeMustBe(rightOperand.type.val,
             Types.Integer,
   error() +  "Right operand of" +
      toString(operatorCode)
   +  "must be an int.");
  };
  private exprNode leftOperand;
  private exprNode rightOperand;
  private int operatorCode;
};
```

```
class identNode extends exprNode {
  void checkTypes() {
   SymbolInfo    id;
   //In CSX-lite all IDs are vars!
   assert(kind.val == Kinds.Var);

   id = (SymbolInfo)
          st.localLookup(idname);
   if (id == null) {
     System.out.println(error() +
      idname +" is not declared.");
     typeErrors++;
     type = new Types(Types.Error);
    } else {
      type = id.type;
      idinfo = id;
      // Save ptr to sym table entry
    }
  }

  publicString idname;
  public  SymbolInfo  idinfo;
  // sym table entry for this ident
  private boolean nullFlag;
};
```

```
class intLitNode extends exprNode {
  void checkTypes() {
  // All intLits are automatically
  // type-correct
  }
  private int intval;
};

class nameNode extends exprNode {
  void checkTypes() {
    varName.checkTypes();
    // Subscripts not in CSX Lite
    type=varName.type;
  };

  private identNode varName;
  private exprNode subscriptVal;
};
}
```