Tutorial 3 (Semantic Analysis – Part 1: eLearning)

1. List the differences between the *parse tree* and the *abstract syntax tree* (AST).

2. Study the abstract syntax of PL/3007 in Appendix A and the typeCheck aspect in Appendix B. Explain why abstract AST nodes (classes) Expr, BinaryExpr, CompExpr and ArithCompExpr are defined in the abstract syntax of PL/3007.

3. Understand the abstract syntax of PL/3007 defined in Appendix A and draw the ASTs that will be built by the parser for the following programs.

```
(1)
module Foo {
   import Bar;
   import Baz;
}

(2)
module Foo {
    type integer = “int”;
    public void bar( ) { }
    int baz;
}

(3)
module Foo {
    int bar(int x) {
        int z;
        z = x + 19;
        return z;
   }
}
```

Appendix A: Abstract Syntax for PL/3007

```
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;
ModExpr : BinaryExpr;
abstract UnaryExpr : Expr ::= Operand:Expr;
NegExpr : UnaryExpr;

abstract CompExpr : BinaryExpr;
EqExpr : CompExpr;
NeqExpr : CompExpr;
abstract ArithCompExpr : CompExpr;
LtExpr : ArithCompExpr;
GtExpr : ArithCompExpr;
LeqExpr : ArithCompExpr;
GeqExpr : ArithCompExpr;

abstract Literal : Expr;
StringLiteral : Literal ::= <Value:String>;
IntLiteral : Literal ::= <Value:Integer>;
BooleanLiteral : Literal ::= <Value:Boolean>;
ArrayLiteral : Literal ::= Element:Expr*;

FunctionName ::= <Name:String>;

abstract TypeDescriptor;
IntType : TypeDescriptor;
BooleanType : TypeDescriptor;
VoidType : TypeDescriptor;
ArrayType : TypeDescriptor ::= ElementType:TypeDescriptor;
```

```
JavaType : TypeDescriptor ::= <Name:String>;
```

Appendix B: TypeCheck Aspect for the Abstract Syntax of PL/3007

```
/** Type checking. */
aspect Typecheck {
	// some convenience attributes
	syn boolean TypeDescriptor.isArrayType() = false;
	eq ArrayType.isArrayType() = true;

	syn boolean TypeDescriptor.isInt() = false;
	eq IntType.isInt() = true;

	syn boolean TypeDescriptor.isNumeric() = false;
	eq IntType.isNumeric() = true;

	syn boolean TypeDescriptor.isVoid() = false;
	eq VoidType.isVoid() = true;

	/* Methods for performing type checking. */
	public void Program.typecheck() {
		for(Module module : getModules())
			module.typecheck();
	}

	public void Module.typecheck() {
		for(Declaration decl : getDeclarations())
			decl.typecheck();
	}

	public void Declaration.typecheck() {}

	public void FunctionDeclaration.typecheck() {
		getReturnType().typecheck();
		for(Parameter parm : getParameters())
			parm.typecheck();
		getBody().typecheck();
	}

	public void FieldDeclaration.typecheck() {
		getVarDecl().typecheck();
	}

	public void VarDecl.typecheck() {
		getTypeName().typecheck();
		/* code for checking that the variable is not declared to be
of type 'void' */
	}

	public void Stmt.typecheck() {}

	public void VarDeclStmt.typecheck() {
		getVarDecl().typecheck();
	}

	public void Block.typecheck() {
		for(Stmt stmt : getStmts())
			stmt.typecheck();
```

```
	}

	public void ExprStmt.typecheck() {
		getExpr().typecheck();
	}

	public void IfStmt.typecheck() {
		getExpr().typecheck();
		getThen().typecheck();
		if(hasElse())
			getElse().typecheck();

		/* code for checking the if condition is of type boolean */
	}

	inh FunctionDeclaration Stmt.getFunction();
	eq FunctionDeclaration.getChild().getFunction() = this;

	// check that return statement returns expression of right type
	public void ReturnStmt.typecheck() {
		if(hasExpr()) {
			getExpr().typecheck();
			/* code for checking the return expression is of the
right type, and that
			 *	it is not of type void */
		}
	}

	// check that loop condition is not of type void
	public void WhileStmt.typecheck() {
		getExpr().typecheck();
		getBody().typecheck();

		/* code for checking the loop condition is of type boolean */
	}

	public abstract void Expr.typecheck();

	public void VarName.typecheck() {
	}

	// check that base expression is array, and index expression is
integer
	public void ArrayIndex.typecheck() {
		getBase().typecheck();
		getIndex().typecheck();

		/* code for checking the base expression is of array type, and
that the index
		 *	expression is of type int
		 */
	}

	// typecheck function call
	public void Call.typecheck() {
		FunctionDeclaration callee = getCallTarget();
		/* code for checking the number of arguments is the same as
the number of parameters */

		for(int i=0;i<getNumArgument();++i) {
			getArgument(i).typecheck();
```

```
                  /* code for checking the argument has the right type */
            }
      }

      // check assignment compatibility
      public void Assignment.typecheck() {
            getLHS().typecheck();
            getRHS().typecheck();
            /* code for checking both sides have the same type */
      }

      // check that operands of binary expression are numeric
      public void BinaryExpr.typecheck() {
            getLeft().typecheck();
            getRight().typecheck();
            /* code for checking both operands have numeric type */
      }

      // check types of comparison operands
      public void CompExpr.typecheck() {
            getLeft().typecheck();
            getRight().typecheck();
            /* code for checking that both operands have the same type,
and that this type is not void */
      }

      public void ArithCompExpr.typecheck() {
            super.typecheck();
            /* code for checking that both operands have numeric type */
      }

      public void UnaryExpr.typecheck() {
            getOperand().typecheck();
            /* code for checking that the operand has numeric type */
      }

      public void Literal.typecheck() {}

      public void ArrayLiteral.typecheck() {
            /* code for checking that the array literal has at least one
element;
             *       check that every element has the same type;
             *       check that no element has type void */
            for(int i=0;i<getNumElement();++i) {
                  getElement(i).typecheck();
            }
      }

      public void TypeName.typecheck() {}
      public void ArrayTypeName.typecheck() {
            /* code for checking that the element type is not 'void' */
      }
}
```