Some Notes on JastAddJ

February 17, 2009

1 The Java 1.4 Frontend

The Java 1.4 Frontend implements parsing, type checking, and error checking for the Java 1.4 language.

1.1 Abstract Grammar

The abstract grammar of Java 1.4 is specified in the file `java.ast`. Here is a short summary of the most important AST node classes.

- `Program`

  ```
  Program ::= CompilationUnit*
  ```

  This node type represents an entire Java program. Its only child is a list of compilation units.

- `CompilationUnit`

  ```
  CompilationUnit ::= <PackageDecl:java.lang.String>
                      ImportDecl* TypeDecl*;
  ```

  This node represents a compilation unit, i.e. either a source file or a class file. `PackageDecl` is the name of the package this compilation unit belongs to, `ImportDecl*` and `TypeDecl*` are its import declarations and type declarations, respectively.

- `Access`

  ```
  abstract Access : Expr;
  ```

  An `Access` is an expression that refers to a declared entity, such as a package, a type, a variable, or a method. Roughly speaking, accesses are qualified names, including array accesses and method calls.

- `AbstractDot`, `Dot`

```
AbstractDot : Access ::= Left:Expr Right:Access;
Dot : AbstractDot;
```

A `Dot` is a qualified access, such as **`this`**`.x` or `"aluis".length()`, with `Left` and `Right` being the qualifier and the qualifiee, respectively. Note that `Left` can be any expression, not necessarily an access (for example, it is a string literal in the second example above), whereas `Right` has to be an access.

The node type `AbstractDot` represents both `Dot`s and array access expressions: for instance, `x[i++]` is represented as an `AbstractDot` whose `Left` child is the access `x`, and whose `Right` child is an `ArrayAccess`, in turn containing the expression `i++`.

Usually, it is more convenient to program against `AbstractDot` instead of `Dot`, since the two cases can often be treated identically.

- `VarAccess`

```
VarAccess : Access ::= <ID:String>
```

A `VarAccess` is an occurrence of a variable, whose name is given by the terminal child `ID`.

- `MethodAccess`

```
MethodAccess : Access ::= <ID:String> Arg:Expr*;
```

A `MethodAccess` is a call to a method `ID` with the list of arguments `Arg`.

- `ConstructorAccess`, `SuperConstructorAccess`

```
ConstructorAccess : Access ::= <ID:String> Arg:Expr*;
SuperConstructorAccess : ConstructorAccess;
```

A `ConstructorAccess` is an explicit constructor invocation of the form **`this`**`(x, y)`. The `ID` is always `"this"`, and `Arg` contains the list of arguments.

A `SuperConstructorAccess` is an explicit super constructor invocation such as **`super`**`(z)`. Here, `ID` is always `"super"`, with `Arg` again giving the list of arguments.

It is important not to confuse these nodes with `ClassInstanceExpr`: the latter represents object construction through **`new`**, whereas these two nodes are only for explicit constructor calls from within other constructors.

- `TypeAccess`

```
TypeAccess : Access ::= <Package:String> <ID:String>;
```

A `TypeAccess` is the equivalent of a `VarAccess` for types. The name of the type being referenced is given by `ID`. As an optimisation, if a type is accessed with a qualifier indicating its package, that package is given by `Package` (which otherwise is `""`).

For example, in the declaration `String s`, the type access `String` is represented as a `TypeAccess` with `Package` being the empty string and `ID` being `"String"`. If, on the other hand, we have a declaration `java.lang.String s`, then `Package` will be `"java.lang"`, and `ID` as before.

- `PrimitiveTypeAccess`

```
PrimitiveTypeAccess : TypeAccess ::= /<Package:String>/
                              /<ID:String>/ <Name:String>;
```

A `PrimitiveTypeAccess` is represents a keyword like **`int`** or **`char`** that accesses a primitive type. The name of the primitive type is in `Name`, `Package` is the special string `"@primitive"`, and `ID` is the same as `Name`.

- `PrimitiveTypeAccess`

```
ArrayTypeAccess : TypeAccess ::= /<Package:String>/
                                 /<ID:String>/ Access;
ArrayTypeWithSizeAccess : ArrayTypeAccess ::= Expr;
```

An `ArrayTypeAccess` represents the array type whose component type is represented by its child `Access`. For example, the type `String[][]` is represented by an `ArrayTypeAccess` whose child `Access` is another `ArrayTypeAccess`, whose child is a `TypeAccess` with `ID` `"String"`.

Both `Package` and `ID` of an `ArrayTypeAccess` are copied from its `Access`.

An `ArrayTypeWithSizeAccess` has additional information about the size of the array type to represent type accesses like `String[42]`.

- `ThisAccess`, `SuperAccess`

```
ThisAccess : Access ::= <ID:String>;
SuperAccess : Access ::= <ID:String>;
```

These node types represent, respectively, the accesses **`this`** and **`super`**. Their `ID`s are `"this"` and `"super"`.

- `PackageAccess`

```
PackageAccess : Access ::= <Package:String>;
```

A `PackageAccess` represents a package name. Package names are not hierarchical, i.e. an access to `java.lang` is simply stored as a `PackageAccess` with `Package` being `"java.lang"`.

`PackageAccess` nodes occur very rarely. Mostly, package names only occur as qualifiers of type accesses, and are then merged into their `TypeAccess` node as explained above.

- `ArrayAccess`

```
ArrayAccess : Access ::= Expr;
```

An `ArrayAccess` is an expression that indexes an array. As explained above, an expression like `x[i++]` is represented by an `AbstractDot` node, whose `Left` child represents the array (here `x`); its `Right` child is an `ArrayAccess`, with the expression `i++` represented by its child `Expr`.

- `ClassInstanceExpr`

```
ClassInstanceExpr : Access ::= Access Arg:Expr*
                               [TypeDecl];
```

A `ClassInstanceExpression` is an object instantiation. The simplest case would be an expression like **`new`** `ArrayList()`: It corresponds to a `ClassInstanceExpression` whose `Access` child holds a `TypeAccess` with `ID "ArrayList"`. The list `Arg` is empty, and the `TypeDecl` child is an empty optional node.

A slightly more complicated example would be **`new`** `ArrayList(23)`. This `ClassInstanceExpression`'s `Access` child is the same as before, but now `Arg` is the one-element list containing the expression `23`. As before, `TypeDecl` is empty.

Finally, in an anonymous class like **`new`** `MouseInputAdapter() { ... }`, the declaration of the anonymous class becomes the value of the `TypeDecl` field of the `ClassInstanceExpression`.

- `ClassAccess`

```
ClassAccess : Access ::= ;
```

A `ClassAccess` is used to represent class literals. For example, `String.`**`class`** corresponds to an `AbstractDot` whose `Left` child is the access to `String`, and whose `Right` child is a `ClassAccess`.

Note that despite their similar names `TypeAccess` and `ClassAccess` are very different beasts.

- `TypeDecl`

```
abstract TypeDecl ::= Modifiers <ID:String> BodyDecl*;
abstract ReferenceType : TypeDecl;
Modifiers ::= Modifier*;
Modifier ::= <ID:String>;
```

`TypeDecl` is an abstract super class representing all sorts of type declarations. `ReferenceType` more specifically represents class and interface declarations.

All type declarations have `Modifiers` and a name, given by `ID`, as well as some declarations (e.g., of member methods), given by `BodyDecl*`.

Modifiers are simply given by their string representations in `ID`.

- `PrimitiveType`, `NumericType`, `BooleanType`, `IntegralType`, `ByteType`, `ShortType`, `IntType`, `LongType`, `CharType`, `FloatingPointType`, `FloatType`, `DoubleType`, `NullType`, `VoidType`

```
PrimitiveType : TypeDecl ::= Modifiers <ID:String>
              [SuperClassAccess:Access] BodyDecl*;
abstract NumericType : PrimitiveType;
BooleanType : PrimitiveType;
abstract IntegralType : NumericType;
ByteType : IntegralType;
ShortType : IntegralType;
IntType : IntegralType;
LongType : IntegralType;
CharType : IntegralType;
FloatingPointType : NumericType;
FloatType : FloatingPointType;
DoubleType : FloatingPointType;
NullType : TypeDecl;
VoidType : TypeDecl;
```

  Node types to represent the builtin types of Java.

- `EmptyType`, `VoidType`

```
EmptyType : PrimitiveType;
UnknownType : ClassDecl;
```

  Convenience node types.

- `ClassDecl`, `InterfaceDecl`, `ArrayDecl`

```
ClassDecl : ReferenceType ::= Modifiers <ID:String>
       [SuperClassAccess:Access] Implements:Access*
                                         BodyDecl*;
InterfaceDecl : ReferenceType ::= Modifiers <ID:String>
                     SuperInterfaceId:Access* BodyDecl*;
ArrayDecl : ClassDecl;
```

  A `ClassDecl` represents a declaration of a class, and `InterfaceDecl` a declaration of an interface. Their children correspond straightforwardly to the syntactic elements of the declarations.

  An `ArrayDecl` does not correspond to a source-level declaration; it is created on-demand for every `ArrayTypeAccess` occurring in the program.

- `BodyDecl`, `InstanceInitializer`, `StaticInitalizer`, `ConstructorDecl`, `MemberDecl`

```
abstract BodyDecl;
InstanceInitializer : BodyDecl ::= Block;
StaticInitializer : BodyDecl ::= Block;
```

```
ConstructorDecl : BodyDecl ::= Modifiers <ID:String>
   Parameter:ParameterDeclaration* Exception:Access*
                 [ConstructorInvocation:Stmt] Block;
abstract MemberDecl : BodyDecl;
```

A BodyDecl is any declaration that occurs within a type declaration. Such a body declaration can be an initializer, either a InstanceInitializer or a StaticInitializer, or a constructor declaration (ConstructorDecl), or a member declaration (MemberDecl), i.e. a declaration of a method or field.

Initializers only consist of a single Block of statements. Constructor declarations have modifiers, parameters, declared exceptions, and a block. Their ID is always the name of the surrounding class. The first statement of a constructor may be an invocation of another constructor of the same class or a constructor of the super class. Such a statement, although syntactically part of the Block making up the body of the constructor, is in many ways special (for example, it can only occur in this precise syntactic position), so it is treated specially in the syntax tree, and appears as an optional child ConstructorInvocation of the constructor declaration.

- FieldDeclaration

```
FieldDeclaration : MemberDecl ::= Modifiers
              TypeAccess:Access <ID:String> [Init:Expr];
MethodDecl : MemberDecl ::= Modifiers TypeAccess:Access
              <ID:String> Parameter:ParameterDeclaration*
              Exception:Access* [Block];
ParameterDeclaration ::= Modifiers TypeAccess:Access
              <ID:String>;
abstract MemberTypeDecl : MemberDecl;
MemberClassDecl : MemberTypeDecl ::= ClassDecl;
MemberInterfaceDecl : MemberTypeDecl ::= InterfaceDecl;
```

Different kinds of member declarations for fields, methods, and member types. The children correspond directly to the syntactic elements of the corresponding declarations.

Note that parameters can be declared **final**, hence a ParameterDeclaration has Modifiers.

- Expr

```
abstract Expr;
```

The abstract super type of all expressions.

- assignment expressions

```
abstract AssignExpr : Expr ::= Dest:Expr Source:Expr;
AssignSimpleExpr : AssignExpr ;
abstract AssignMultiplicativeExpr : AssignExpr;
```

```
AssignMulExpr : AssignMultiplicativeExpr ;
AssignDivExpr : AssignMultiplicativeExpr ;
AssignModExpr : AssignMultiplicativeExpr ;
abstract AssignAdditiveExpr : AssignExpr;
AssignPlusExpr : AssignAdditiveExpr ;
AssignMinusExpr : AssignAdditiveExpr ;
abstract AssignShiftExpr : AssignExpr;
AssignLShiftExpr : AssignShiftExpr ;
AssignRShiftExpr : AssignShiftExpr ;
AssignURShiftExpr : AssignShiftExpr ;
abstract AssignBitwiseExpr : AssignExpr;
AssignAndExpr : AssignBitwiseExpr ;
AssignXorExpr : AssignBitwiseExpr ;
AssignOrExpr : AssignBitwiseExpr ;
```

Node types for the manifold assignment operators of Java. Every assignment has a left hand side `Dest` and a right hand side `Source`, both of which are expressions.

- Primary Expressions

```
abstract PrimaryExpr : Expr;

abstract Literal : PrimaryExpr ::= <LITERAL:String>;
IntegerLiteral : Literal ;
LongLiteral : Literal ;
FloatingPointLiteral : Literal ;
DoubleLiteral : Literal ;
BooleanLiteral : Literal ;
CharacterLiteral : Literal ;
StringLiteral : Literal ;
NullLiteral : Literal ;

ParExpr : PrimaryExpr ::= Expr;
```

Primary expressions are the simplest kind of expressions. Literals have a child `LITERAL` that contains a string representation of their constant value. For the `NullLiteral`, this is always `"null"`.

`ParExpr` nodes represent parenthesised expressions. They are not handled specially in any way, in particular they are not automatically inserted when creating ASTs.

- `ArrayCreationExpression`, `ArrayInit`

```
ArrayCreationExpr : PrimaryExpr ::= TypeAccess:Access
                                        [ArrayInit];
ArrayInit : Expr ::= Init:Expr*;
```

An `ArrayCreationExpr` represents an expression that allocates an array, such as **`new int`**`[] {23, 42}`. By contrast, an `ArrayInit` can only occur in the initialising expression of a variable declaration, and represents an array literal like `{23, 42}`.

- `CastExpr`

```
CastExpr : Expr ::= TypeAccess:Access Expr;
```

A `CastExpr` node represents a cast, with the child expression giving the expression to be casted, and `TypeAcess` the type it is casted to.

- `InstanceOfExpression`

```
InstanceOfExpr : Expr ::= Expr TypeAccess:Access;
```

An `InstanceOfExpression` represents a run-time type check using **`instanceof`**, with `Expr` the expression being checked, and `TypeAccess` the type it is checked against.

- Unary Expressions

```
abstract Unary : Expr ::= Operand:Expr;
PreIncExpr : Unary ;
PreDecExpr : Unary ;
MinusExpr : Unary ;
PlusExpr : Unary ;
BitNotExpr : Unary ;
LogNotExpr : Unary ;
abstract PostfixExpr : Unary;
PostIncExpr : PostfixExpr ;
PostDecExpr : PostfixExpr ;
```

Node types to represent unary expressions.

- Binary Expressions

```
abstract Binary : Expr ::= LeftOperand:Expr
                           RightOperand:Expr;

abstract ArithmeticExpr : Binary;
abstract MultiplicativeExpr : ArithmeticExpr;
MulExpr : MultiplicativeExpr ;
DivExpr : MultiplicativeExpr ;
ModExpr : MultiplicativeExpr ;
abstract AdditiveExpr : ArithmeticExpr;
AddExpr : AdditiveExpr ;
SubExpr : AdditiveExpr ;

abstract ShiftExpr : Binary;
LShiftExpr : ShiftExpr ;
RShiftExpr : ShiftExpr ;
URShiftExpr : ShiftExpr ;
```

```
abstract BitwiseExpr : Binary;
AndBitwiseExpr : BitwiseExpr ;
OrBitwiseExpr : BitwiseExpr ;
XorBitwiseExpr : BitwiseExpr ;

abstract LogicalExpr : Binary;
AndLogicalExpr : LogicalExpr ;
OrLogicalExpr : LogicalExpr ;

abstract RelationalExpr : Binary;
LTExpr : RelationalExpr ;
GTExpr : RelationalExpr ;
LEExpr : RelationalExpr ;
GEExpr : RelationalExpr ;

abstract EqualityExpr : RelationalExpr;
EQExpr : EqualityExpr ;
NEExpr : EqualityExpr ;
```

Node types to represent binary expressions. In particular, `AndLogicalExpr` is `&&` and `AndBitwiseExpr` is `&`.

- `ConditionalExpression`

```
ConditionalExpr : Expr ::= Condition:Expr TrueExpr:Expr
                                          FalseExpr:Expr;
```

The ternary conditional expression.

- `Stmt`

```
abstract Stmt;
```

An abstract node type to represent statements.

- `VariableDeclaration`

```
VariableDeclaration : Stmt ::= Modifiers TypeAccess:Access
                                    <ID:String> [Init:Expr];
```

Similar to a `FieldDeclaration`, this node type represents a declaration of a local variable with an optional initialisation expression.

- `Block`, `EmptyStmt`, `ExprStmt`

```
Block : Stmt ::= Stmt*;
EmptyStmt : Stmt;
ExprStmt : Stmt ::= Expr;
```

Three particularly simple kinds of statements: A `Block` represents a block of statements in curly braces, an `EmptyStmt` is the do-nothing statement "`;`", and an expression statement wraps an expression.

- Branch Target Statements

```
abstract BranchTargetStmt : Stmt;
LabeledStmt : BranchTargetStmt ::= <Label:String> Stmt;
SwitchStmt : BranchTargetStmt ::= Expr Block;
WhileStmt : BranchTargetStmt ::= Condition:Expr Stmt;
DoStmt : BranchTargetStmt ::= Stmt Condition:Expr;
ForStmt : BranchTargetStmt ::= InitStmt:Stmt*
            [Condition:Expr] UpdateStmt:Stmt* Stmt;
```

A branch target statement is a statement that can be reached by a **`break`** or **`continue`**. In particular, all the loops are branch targets, but also switch statements and labeled statements.

Note that switch statements are represented amorphously as the `Expr` to be tested, and a block of statements, some of which may be case labels.

- `Case`, `ConstCase`, `DefaultCase`

```
abstract Case : Stmt;
ConstCase : Case ::= Value:Expr;
DefaultCase : Case;
```

These node types represent case labels in a switch statements. The AST does not structurally guarantee that these labels only occur inside a `SwitchStmt`, this must be checked by the compiler frontend.

- `IfStmt`

```
IfStmt : Stmt ::= Condition:Expr Then:Stmt [Else:Stmt];
```

An **`if`** statement with optional **`else`** branch.

- Control Transfer Statements

```
BreakStmt : Stmt ::= <Label:String>;
ContinueStmt : Stmt ::= <Label:String>;
ReturnStmt : Stmt ::= [Result:Expr];
```

Node types to represent the "disguised gotos" **`break`** and **`continue`**, and the **`return`** statement.

For the former two, the `Label` will be the empty string if there is no explicit label.

- Synchronization

```
SynchronizedStmt : Stmt ::= Expr Block;
```

A **synchronized** statement, which synchronizes execution of `Block` on the value of `Expr`.

- Exception Handling

```
ThrowStmt : Stmt ::= Expr;
TryStmt : Stmt ::= Block CatchClause* [Finally:Block];
CatchClause ::= Parameter:ParameterDeclaration Block;
```

These node types represent the Java exception handling constructs.

- Assertions

```
AssertStmt : Stmt ::= first:Expr [Expr];
```

An `AssertStmt` represents an assertion statement.

- Local and anonymous classes

```
AnonymousDecl : ClassDecl ::= Modifiers <ID:String>
       /[SuperClassAccess:Access]/ /Implements:Access*/
                                             BodyDecl*;
LocalClassDeclStmt : Stmt ::= ClassDecl;
```

An `AnonymousDecl` represents a declaration of an anonymous class occurring as part of a `ClassInstanceExpression`; `ID` is the name of the class it extends, whereas `BodyDecl*` are its body declarations.

A `LocalClassDeclStmt` represents the declaration of a local class within a constructor, method, or initialiser.

Another important node type is `Variable`, declared in file `VariableDeclaration.jrag`: It is an interface implemented by `VariableDeclaration`, `ParameterDeclaration`, and `FieldDeclaration` that provides a common API for different kinds of variables.

1.2 Name and Type Analysis

The code pertaining to name and type analysis is mainly contained within the following files:

- `AccessControl.jrag` defines attributes to implement accessibility rules. In particular, it provides a family of attributes `accessibleFrom`, where `x.accessibleFrom(y)` holds if the entity `x` can be accessed from node `y`.

- `LookupConstructor.jrag` defines a method `lookupConstructor` to look up a constructor for a `TypeDecl` by its signature. It also provides an attribute `decl()` for classes `ConstructorAccess` and `ClassInstanceExpression` that computes the `ConstructorDecl` they resolve to.

- `LookupMethod.jrag` defines an attribute `lookupMethod` to look up a method by name. It also provides an attribute `decl()` on class `MethodAccess` that computes the `MethodDecl` the call resolves to.
- `LookupType.jrag` defines an attribute `lookupType` to look up a type by its simple or qualified name. It also provides an attribute `decl()` on class `TypeAccess` that computes the `TypeDecl` the access refers to.
- `LookupVariable.jrag` defines an attribute `lookupVariable` to look up a variable (i.e., field, parameter, or local variable) by name. It also provides an attribute `decl()` on class `VarAccess` that computes the `Variable` it binds to.
- `NameCheck.jrag` implements the name checks performed by the compiler frontend.
- `TypeAnalysis.jrag` handles the different kinds of implicit conversions in Java, and defines an attribute `type()` on expression nodes that computes their type.
- `TypeCheck.jrag` and `TypeHierarchyCheck.jrag` implement the type checks performed by the compiler frontend.

The two most important attributes defined in this part of the frontend are `decl()` and `type()` to access a type's declaration, and an expression's type.

1.3 Definite Assignment

The file `DefiniteAssignment.jrag` checks that a program adheres to Java's definite assignment rule, which stipulates that every local variable must be provably assigned before it is used the first time.

Two very useful attributes defined in this file are `isDest()` and `isSource()`, both defined on node type `Expr`, which indicate whether an expression is an lvalue or an rvalue. Some expressions can be both, for example the `i` in `i++`.

2 Control Flow Analysis

The control flow analysis framework is not part of the compiler frontend proper; it resides in the project `ControlFlowGraph`. Its most important file is `ControlFlowGraph.jrag`, which defines two attributes `pred()` and `succ()` that compute an AST node's control flow predecessors and successors, respectively.

Control flow is tracked at the expression level, hence it makes sense to ask for the control flow successor of `x` in `x + y`. Since in general a node can have more than one control flow successor or predecessor, the attributes return sets of nodes. Note that the analysis is intra-procedural, i.e. only predecessors and successors within the same method (or constructor or initialiser) are computed.

For efficiency reasons, a custom implementation of sets called `SmallSet` is used. It is defined in `Sets.jrag`.

A JastAdd Syntax

A.1 AST Node Types

AST node types are usually defined in files with the extension `.ast`. Every node type is given a (single) production, e.g.

```
TryStmt : Stmt ::= Block CatchClause* [Finally:Block];
```

This declares the node type `TryStmt` as extending the node type `Stmt`. It has three children: one of type `Block`, a child of type `List<CatchClause>`, which holds a (potentially empty) list of nodes of type `CatchClause`, and a child of type `Opt<Block>`, which holds either nothing or a node of type `Block`. The third child is additionally given the name `Finally`.

From this declaration, JastAdd generates the skeleton of a Java class with getter methods for the children that looks somewhat like this:

```
class TryStmt extends Stmt {
  public TryStmt(Block p0, List<CatchClause> p1, Opt<Block> p2) {
    ...
  }

  public Block getBlock() { ... }
  public List<CatchClause> getCatchClauseList() { ... }
  public Opt<Block> getFinallyOpt() { ... }

  public int getNumCatchClause() { ... }
  public CatchClause getCatchClause(int i) { ... }

  public boolean hasFinally() { ... }
  public Block getFinally() { ... }
}
```

Observe in particular that for list children and optional children there are additional convenience methods: for the former, we can query the number of children in the list, and access a particular one given its position; for the latter, we can determine whether the child is present, and retrieve it directly.

Node types may also have terminal children, as in

```
VarAccess : Access ::= <ID:String>;
```

JastAdd creates getter methods for them just like for node children.

A final variety of children are non-terminal attributes, which are not important for the purposes of this introduction.

Just like a Java class, a node type can be declared **`abstract`**, which will make the generated class abstract as well; such a node type cannot be instantiated directly.

A.2 Attributes

Once a node type is declared, we can define attributes on it. This is done in separate files with the extension `jrag`. Such files should contain *aspect declarations* as their top-level entities. For example, the file `LookupVariables.jrag` from the Java 1.4 compiler frontend defines attribute for handling variable lookup; all these attribute definitions are inside three aspects `VariableScope`, `VariableScopePropagation`, and `Fields`:

```
aspect VariableScope {
  ...
  (attribute definitions)
  ...
}

aspect VariableScopePropagation {
  ...
  (further definitions)
  ...
}

aspect Fields {
  ...
  (further definitions)
  ...
}
```

A.2.1 Synthesised Attributes

The simplest form of attributes are *synthesised attributes*. They are declared on a node type like this:

```
  syn boolean Stmt.declaresVariable(String name) = false;
```

This declares a synthesised attribute `declaresVariable` on node type `Stmt`, which takes a single parameter `name` of type `String` and returns **`boolean`**. The declaration also provides a default implementation, which just returns **`false`**.

The above declaration is translated into the following (somewhat simplified) Java method declaration, which is inserted into the definition of class `Stmt`:

```
  public boolean declaresVariable(String name) {
    return false;
  }
```

Methods can be overridden on subclasses; likewise, synthesised attributes can be given different definitions on derived node types:

```
  eq VariableDeclaration.declaresVariable(String name)
    = name().equals(name);
```

Besides the equational definition style in this example, synthesised attributes can also be defined using Java method syntax. Indeed, the above definition could be rewritten as

```
  eq VariableDeclaration.declaresVariable(String name) {
    return name().equals(name);
  }
```

Synthesised attributes can be declared `lazy`, meaning that their value will be cached and reused on further invocations without being recomputed. Lazy attributes should normally be side-effect free, although this is not enforced by the system.

Synthesised attributes can be declared `circular`:

```
syn lazy boolean TypeDecl.isCircular() circular [true] = false;

eq ClassDecl.isCircular() { ... }
eq InterfaceDecl.isCircular() { ... }
```

This means that, for every `TypeDecl`, JastAdd performs a fixed point iteration to determine the value of `isCircular`, starting at the value `true` and iterating until no further value change is observed. The user has to ensure that the definitions given for such attributes are monotonic, and that a fixed point may be reached by iteration.

A.2.2 Inherited Attributes

While the definition of a synthesised attribute only depends on the children of a node (and the values of their attributes), an inherited attribute can make use of additional information about the location of the node with respect to its parent node.

For example, the definition

```
eq TypeDecl.getBodyDecl(int i).lookupVariable(String name) {
  (...)
}
```

defines the attribute `lookupVariable` on a `BodyDecl`, but only if it is the child of a `TypeDecl`. Additionally, the body of the declaration knows the index `i` at which the body declaration occurs within the type.

Inherited attributes have to be defined for every possible combination of parent and child nodes. If no explicit definition is given, JastAdd provides a default copy rule that recursively evaluates the attribute on the parent node.

To use an inherited attribute on a certain node type, one has to declare it visible:

```
inh lazy SimpleSet Block.lookupVariable(String name);
```

This declaration causes a method `lookupVariable(String)` to be inserted into the definition of class `Block`, that uses the definition rules of the attribute.

Like synthesised attributes, inherited attributes can also be declared `lazy` and `circular`.

A.2.3 Collection Attributes

While synthesised and inherited attributes are locally defined, collection attributes are global entities whose definition is contributed to by all the nodes in a certain subtree, or even the entire syntax tree.

For example, to collect the names of all the types declared in a Java program one could define the collection attribute

```
coll HashSet<String> Program.typeNames()
  [new HashSet<String>()] with add
  root Program;
```

This says that the collection attribute `typeNames` is declared on the program node, and is of type `HashSet<String>`. It is initialised to **`new`** `HashSet<String>()`, and contributions from individual nodes are added using method `add` in type `HashSet<String>`. Contributors to the attribute value should be searched for in nodes below `Program`, i.e. in the entire syntax tree.

Every type declaration now should contribute its own name to this global collection:

```
TypeDecl contributes name()
         to Program.typeNames()
         for getProgram();
```

This says that every type declaration contributes the value of its method `name()` [1] to the `typeNames()` attribute of the `Program` node to be found by evaluating `getProgram()`.

Contributions may also specify a condition using **`when`** `...` to indicate that they only contribute if the condition evaluates to **`true`**.

Note that the contributions may be collected in an arbitrary order.

[1] This can be an arbitrary expression composed of field values and method invocations.