Tutorial:
Generating Language Tools with JastAdd

Görel Hedin

Lund University, Sweden
gorel@cs.lth.se, Draft Oct 29, 2009

Abstract. JastAdd is an open-source system for generating compilers and other language-based tools. Its declarative specification language is based on attribute grammars and object-orientation. This allows tools to be implemented as composable extensible modules, as exemplified by JastAddJ, a complete extensible Java compiler. This tutorial gives an introduction to JastAdd and its core attribute grammar mechanisms. In particular, we cover reference attributes, parameterized attributes, collection attributes, and circular attributes. A simple state machine language is used as a running example.

Key words: attribute grammars, extensible language tools, reference attributes, object-oriented model

1 Introduction

JastAdd is a metacompilation system for generating language-based tools such as compilers, source code analyzers, and language-sensitive editing support. It is based on a combination of attribute grammars and object-orientation. The key feature of JastAdd is that it allows properties of abstract syntax tree nodes to be programmed declaratively. These properties, called *attributes*, can be simple values like integers, composite values like sets, and reference values which point to other nodes in the abstract syntax tree (AST). The reference values allows graph properties to be defined. For example, linking identifier uses to their declaration nodes, or representing call graphs and dataflow graphs. AST nodes are objects, and the resulting data structure, including attributes, is in effect an object-oriented model, rather than only a simple syntax tree.

1.1 Object-oriented model

Figure 1 illustrates the difference from a traditional compiler where important data structures like symbol tables, flow graphs, etc., are typically separate from the AST. In JastAdd, these data structures are instead embedded in the AST, using attributes, resulting in an object-oriented model of the program. JastAdd is integrated with Java, and the resulting model is implemented using Java classes, and the attributes form a method API to those classes.

Attributes are programmed declaratively, using *attribute grammars*: Their values are stated using *equations* that may access other attributes. Because of this declarative programming, the user does not have to worry about in what order to evaluate the attributes. The user simply builds an AST, typically using a parser, and all attributes will then automatically have the correct values according to their equations, and can be accessed using the method API. The actual evaluation of the attributes is carried out automatically and implicitly by the JastAdd system.

The attribute grammars used in JastAdd go much beyond the classical attribute grammars defined by Knuth [Knu68]. In this tutorial, we particularly cover reference attributes [Hed00], parameterized attributes [Hed00,Ekm06], circular attributes [Far86,MH07] and collection attributes [Boy96,MEH09].

An important consequence of the declarative programming is that the object-oriented model in Fig. 1 becomes extensible. The JastAdd user can simply add new attributes, equations, and syntax rules. This makes it easy to extend languages and to build new tools as extensions of existing ones.

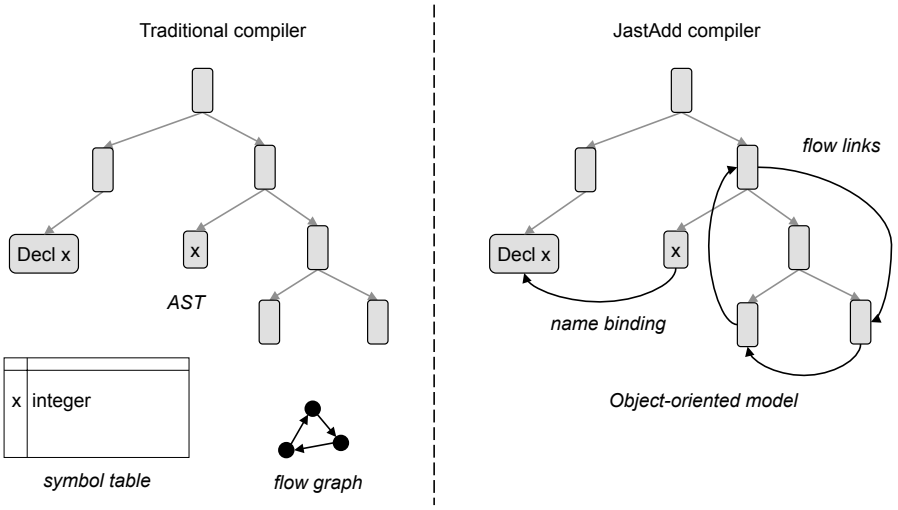


Fig. 1. In JastAdd, compilation data structures are embedded as reference attributes in the AST, resulting in an object-oriented model of the program.

1.2 Extensible languages and tools

In JastAdd, the order of defining attributes and equations is irrelevant—their meaning is the same regardless of order. This allows the user to organize rules into modules arbitrarily, to form modules that are suitable for reuse and composition. Sometimes it is useful to organize modules based on compilation problems, like name analysis, type analysis, dataflow analysis, etc. Other times it

can be useful to organize according to language constructs. As an example, in JastAddJ, an extensible Java compiler built using JastAdd [EH07a], both modularization principles are used, see Figure 2. Here, a basic compiler for Java 1.4 is modularized according to the classical compilation analyses: name analysis, type analysis, etc. In an extension to support Java 5, the modules instead reflect the new Java 5 constructs: the foreach loop, static imports, generics, etc.. Each of those modules contain equations that handle the name- and type analyses for that particular construct. In yet further extensions, new computations are added, like non-null analysis [EH07b], separated into one module handling the Java 1.4 constructs, and another one handling the Java 5 constructs.

JastAdd has been used for implementing a variety of different languages, from small domain-specific languages like the state machine language that will be used in this tutorial, to full-blown general-purpose languages like Java. Because of the modularization support, it is particularly attractive to use JastAdd to build extensible languages and tools.

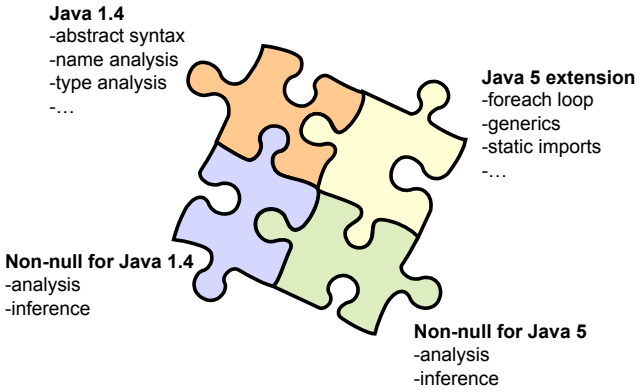


Fig. 2. Each component has modules containing abstract syntax rules, attributes, and equations. To construct a compiler supporting non-null analysis and inference for Java 5, all modules in the four components are used.

1.3 Tutorial outline

This tutorial gives an introduction to JastAdd and its core attribute grammar mechanisms. Section 2 presents a language for simple state machines that we will use as a running example. It is shown how to program towards the generated API for a language: constructing ASTs and using attributes. Basic attribution mechanisms are presented in Section 3, including synthesized and inherited attributes [Knu68], reference attributes [Hed00], and parameterized

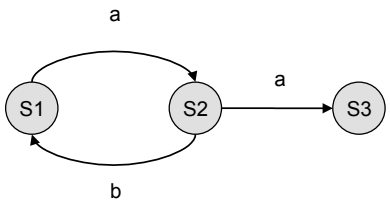
attributes[Hed00,Ekm06]. We show how name analysis can be implemented using these mechanisms. This section also briefly presents the underlying execution model.

The two following sections present more advanced mechanisms. Section 4 discusses how to define composed properties like sets using *collection attributes* [Boy96,MEH09]. These attributes are defined by the combination of values contributed by different AST nodes. We illustrate collection attributes by defining an explicit graph representation for the state machine, with explicit edges between state and transition objects. Section 5 discusses how recursive properties can be defined using *circular attributes* which are evaluated using fixed-point iteration [Far86,MH07]. This is illustrated by the computation of reachability sets for states.

Finally, Section 6 concludes the tutorial. Both the JastAdd metacompilation system and the JastAddJ extensible Java compiler are available as open-source tools at `http://jastadd.org`. The examples in this tutorial are tested on jastadd2.jar version R20090610. [1]

2 Running example: A state machine language

As a running example, we will use a small state machine language. Figure 3 shows an example state machine depicted graphically, and a possible textual representation of the same machine, listing all its states and transitions.




```
state S1;
state S2;
state S3;
trans a:S1->S2;
trans b:S2->S1;
trans a:S2->S3;
```

Fig. 3. An example state machine and its textual representation.

2.1 Abstract grammar

In order to do various computations on the state machine, we would like to construct an object-oriented model of it that explicitly captures its graph properties. We can do this by first defining an abstract grammar that gives a tree representation corresponding to the textual representation, and then add reference attributes to represent the graph properties. Consider the abstract grammar in Fig. 4, written in JastAdd syntax.

[1] See `http://www.cs.lth.se/~gorel/2009-gttse-statemachine.html`. The final version will be made available at `http://jastadd.org`.

```
StateMachine ::= Declaration*;
abstract Declaration;
State : Declaration ::= <Label:String>;
Transition : Declaration ::=
  <Label:String> <SourceLabel:String> <TargetLabel:String>;
```

Fig. 4. Abstract grammar for the state machine language

This grammar models a state machine as a list of declarations which can be either states or transitions. From a grammar perspective, we can think of Declaration as a nonterminal and State and Transition as productions, and StateMachine serving the role as both nonterminal and production. In JastAdd, we view the grammar from an object-oriented perspective where all four are classes. Declaration is here an abstract class, and State and Transition are its subclasses. The entities Label, etc. represent tokens of type String, and can be thought of as fields of the corresponding classes.

2.2 Attributing the AST

To obtain an explicit object-oriented model of the graph, we would like to link each state object to the transition objects that has that state object as its source, and to link each transition object to its target state object. This can be done using reference attributes. Figure 5 shows the resulting object-oriented model for the example machine in Figure 3. We see here how the edges between state and transition objects are embedded in the AST, using reference attributes. Given this object-oriented model, we might be interested in computing, for example, reachability. The set of reachable states could be represented as an attribute in each State object. In sections 3, 4, and 5 we will see how these attributes can be defined.

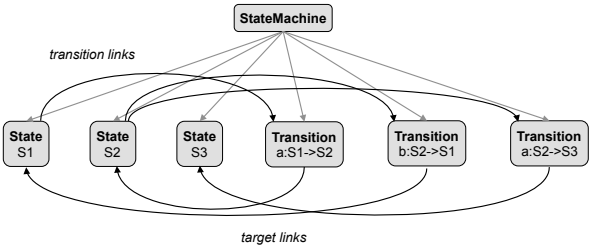


Fig. 5. The state machine graph is embedded in the object-oriented model.

Exercise 1. In Figure 5, the objects are laid out visually to emphasize the AST structure. Make a new drawing that instead emphasizes the state machine graph.

Draw only the State and Transition objects and the links between them, mimicking the layout in Figure 3.

2.3 Building and using the AST

From the abstract grammar, JastAdd generates a Java API with constructors for building AST nodes and methods for traversing the AST. This API is furthermore augmented with methods for accessing the attributes. Figure 6 shows part of the generated API for the state machine language, including the attributes `target`, `transitions`, and `reachable` that will be defined in the coming sections.

```
class StateMachine{
  StateMachine(); // AST construction
  void addDeclaration(Declaration node); // AST construction
  List<Declaration> getDeclarations(); // AST traversal
  Declaration getDeclaration(int i); // AST traversal
}

abstract class Declaration{
}

class State extends Declaration{
  State(String theLabel); // AST construction
  String getLabel(); // AST traversal
  Set<Transition> transitions(); // Attribute access
  Set<State> reachable(); // Attribute access
}

class Transition extends Declaration{
  Transition(String theLabel, theSourceLabel, theTargetLabel); // AST construction
  String getLabel(); // AST traversal
  String getSourceLabel(); // AST traversal
  String getTargetLabel(); // AST traversal
  State target(); // Attribute access
}
```

Fig. 6. API to the state machine model

Suppose we want to print out the reachable states for each state. For the small example in Figure 3, we would like to obtain the following output:

```
S1 can reach {S1, S2, S3}
S2 can reach {S1, S2, S3}
S3 can reach {}
```

meaning that all three states are reachable from S1 and S2, but no states are reachable from S3.

To program this we simply need to build the AST for the state machine, and then call the `reachable` attributes to print out the appropriate information. We do not need to do anything to attribute the AST—this is handled implicitly and automatically. To program the traversal of the AST in order to call the `reachable` attributes, it would be useful to add some ordinary Java methods to the AST classes. This can be done as a separate module using a JastAdd *aspect* as shown in Fig. 7.

```
aspect PrintReachable {
  public void StateMachine.printReachable() {
    for (Declaration d : getDeclarations()) d.printReachable();
  }

  public void Declaration.printReachable() { }

  public void State.printReachable() {
    System.out.println(getLabel() + " can reach {" +
        listOfReachableStateLabels() + "}");
  }

  public String State.listOfReachableStateLabels() {
    boolean insideList = false;
    StringBuffer result = new StringBuffer();
    for (State s : reachable()) {
      if (insideList)
        result.append(", ");
      else
        insideList = true;
      result.append(s.getLabel());
    }
    return result.toString();
  }
}
```

Fig. 7. An aspect defining methods for printing the reachable information for each state.

The aspect uses *inter-type declarations* to add methods to existing classes. For example, the method `void StateMachine.printReachable() ...` means that the method `void printReachable() ...` is added to the class `StateMachine`.[2]

[2] This syntax for inter-type declarations is borrowed from AspectJ. Note, however, that JastAdd aspects support only static aspect-orientation in the form of these inter-type declarations. Dynamic aspect-orientation like pointcuts and advice are not supported.

We can now write the main program that constructs the AST and prints the reachable information, as shown in Fig. 8. Here we have constructed the AST manually, using the construction API. An alternative would be to write a parser that uses the construction API in its semantic actions. Any Java-based parser generator could be used, provided it allows you to place arbitrary Java code in the semantic actions so the appropriate AST can be built. In earlier projects we have used, for example, the LR-based parser generators *CUP* and *beaver*, and the LL-based parser generator *JavaCC*. For parser generators that automatically provide their own AST representation, a possibility is to write a trivial visitor that traverses the parser-generator-specific AST and builds the corresponding JastAdd AST.

```
public class MainProgram {
  public static void main(String[] args) {
    // Construct the AST
    StateMachine m = new StateMachine();
    m.addDeclaration(new State("S1"));
    m.addDeclaration(new State("S2"));
    m.addDeclaration(new State("S3"));
    m.addDeclaration(new Transition("a", "S1", "S2"));
    m.addDeclaration(new Transition("b", "S2", "S1"));
    m.addDeclaration(new Transition("a", "S2", "S3"));

    // Print reachable information for all states
    m.printReachable();
  }
}
```

Fig. 8. A main program that builds an AST and then accesses attributes.

Exercise 2. Write an aspect that traverses a state machine and prints out information about each state, stating if it is on a cycle or not. Hint: You can use the call `s.contains(o)` to find out if the set `s` contains a reference to the object `o`. What is your output for the state machine in Fig. 3? What does your main program look like?

3 Basic attribution mechanisms

We will now look at the two basic mechanisms for defining properties of AST nodes: *synthesized* and *inherited attributes*, which were introduced by Knuth in 1968 [Knu68]. Loosely speaking, synthesized attributes propagate information upwards in the AST, whereas inherited attributes propagate information downwards. The term *inherited* is used here for historical reasons, and its meaning is different from and unrelated to that within object-orientation.

3.1 Synthesized and inherited attributes

An attribute a is defined by an equation whose right-hand side is a function of other attributes, for example, $a = f(b, c, d, ...)$. In JastAdd, attributes and equations are declared in AST classes, so we can think of each AST node as having a set of declared attributes, and a set of equations. Each *synthesized* attribute must be defined by an equation in the node itself, whereas the *inherited* attributes must be defined in an ancestor node.

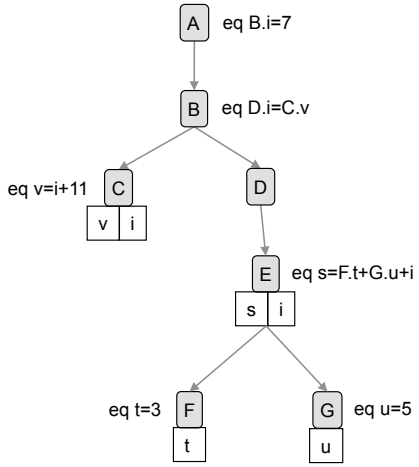


Fig. 9. The attributes `E.s`, `F.t`, `G.u`, and `C.v` are synthesized and have equations in the node they belong to. The attributes `C.i` and `E.i` are inherited. The equation in `B` applies only to the `D` subtree, and thereby to the `E.i` attribute. The equation in `A` applies to the `C.i` attribute, but not to the `E.i` attribute since it is shadowed by the equation in `B`.

Most attributes we introduce will be synthesized attributes. In the equation defining the attribute, we will use information in the node itself, say E, or by accessing its children, say, F and G. However, once in a while, we will find that the information we need is located in the context of the E node, i.e., in its parent, or further up in the AST. In these cases, we will introduce an inherited attribute in E, capturing this information. It is then the responsibility of all nodes that could have an E child, to provide an equation for that inherited attribute. In JastAdd, the equation does not have to be in the immediate parent of E, but there must be an equation in some ancestor of E, on the way from the parent up to the root of the AST. If several of these nodes have an equation for the inherited attribute, the closest one to E will apply. See Fig. 9.

Exercise 3. What will be the values of the attributes in Fig. 9?

Exercise 4. An equation in node n for an inherited attribute i applies to the subtree of one of n's children, say c. All the nodes in this subtree do not need to actually have an i attribute, so the equation applies only to those nodes that actually do. Which nodes in Fig. 9 are within the scope of an equation for `i`, but do not have an `i` attribute?

Exercise 5. In a correctly attributed AST, the attributes will have values so that all equations are fulfilled. How can the correct attribute values be computed? What different algorithms can you think of? (The actual way attributes are evaluated in JastAdd is discussed in Section 3.7.)

3.2 Reference attributes

In JastAdd, synthesized and inherited attributes are generalized in several ways, as compared to the original formulation by Knuth. The most important generalization is that an attribute is allowed to be a *reference* to an AST node. In this way, attributes can connect different AST nodes to each other, forming a graph. Furthermore it is allowed to use reference attributes inside equations, and to access the attributes of their referenced objects. This allows *non-local dependencies*: an attribute in one node can depend directly on attribute values in distant nodes in the AST. The dependencies do not have to follow the tree structure like in a classical Knuthian AG. For example, if each use of an identifier has a reference attribute that points directly to the appropriate declaration node, information about the type can be propagated directly from the declaration to the use node.

Reference attributes thus allows an AST to be extended to a graph in a declarative way. Also cyclic graphs can be defined, as in the example in Figure 10 (see also exercise 6). The example shows several possibilities for equations to access nodes and attributes, e.g.,

- `this`, meaning a reference to the node itself
- `k.v`, accessing the `v` attribute of the node referred to by `k`
- `i.G.t`, accessing the `t` attribute of the `G` child of the node referred to by `i`

Exercise 6. Draw the remaining reference attribute values in Figure 10. In what way is the graph cyclic? What are the values of the ordinary (non-reference) attributes? Give an example of non-local dependencies.

3.3 Parameterized attributes

A second generalization in JastAdd is that attributes may have parameters. A parameterized attribute will have an unbounded number of values, one for each possible combination of parameter values. For example, we may define an attribute `lookup(String)` whose values are references to declarations, different for different String values.

By calling a parameterized attribute via a reference attribute, complex computations can easily be delegated from one node to another. This is useful in, e.g., name analysis, where lookup can be delegated from a method to its enclosing class, and further on to superclasses, following the scope rules of the language.

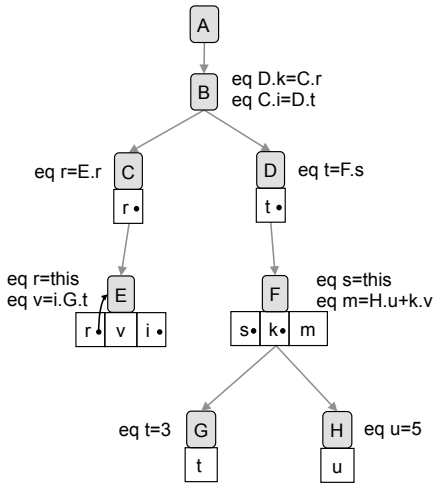


Fig. 10. Reference attributes are indicated with a small dot after the name. An arrow from a dot to a node illustrates the value of the reference attribute: `E.r` refers to `E`, due to the equation `r=this` in `E`.

3.4 Thinking declaratively

When writing an attribute grammar, you should try to *think declaratively*, rather than to think about in which order things need to be computed. Think first what properties you would like the nodes to have to solve a particular problem. In the case of type checking, it would be useful if each expression node had a `type` attribute. The next step is to write equations defining these attributes. In doing so, you will need to solve subproblems that call for the addition of more properties, and so on.

For example, to define the `type` attribute of an identifier expression, it would be useful to have an attribute `decl` that refers to the appropriate declaration node. You could then simply define the identifier's `type` as equal to the `type` of its declaration. The next problem is now to define the `decl` attribute. This problem would be easy to solve if all identifiers had a parameterized attribute `lookup(String)`, which returns a reference to the appropriate declaration node when supplied with the name of the identifier. The next problem is now in defining `lookup(String)`, and so on.

In adding a new attribute, you need to decide if it should be synthesized or inherited. If the definition will use any information in the node itself, or its subtree, the attribute should be synthesized. If all the information needed is outside the node and its subtree, the attribute should be inherited.

As an example, consider the `type` attribute for expressions. Since the type will depend on what kind of expression it is, e.g., an identifier or an add node, the attribute should be synthesized. Similarly, the `decl` attribute should be synthe-

sized since it depends on the identifier's name. The `lookup(String)` attribute, on the other hand, should be inherited since there is no information in the identifier node that is relevant for the definition of this attribute. When introducing an inherited attribute we can view this as delegating the definition to an ancestor node.

In using attributes, with or without parameters, we can view them as methods of AST nodes. Attributes are similar to abstract methods, and equations are similar to method implementations. In fact, when accessing attributes, we will use Java method call syntax, e.g., `a()`, and when we write an equation, we can write the right-hand side as a Java method body. The similarity is particularly apparent for synthesized attributes, which are defined in the AST node itself, just like methods. Inherited attributes are different in that the attribute is declared in one node, but defined in another (in an ancestor).

Although an equation can be written as a Java method body, an important difference from an ordinary Java method is that it must not have any externally visible side effects. The equation code may use local variables and assignments to compute some value, but it may not change any global information, for example fields of AST nodes or global data. The reason for this is that equations represent *definitions*, and not effects of execution. As soon as an AST has been created, all its attributes automatically contain the correct values, according to their defining equations. The underlying attribute evaluator that accomplishes this will run the equation code, but the user does not have any explicit control over in what order the equations are run, or how many times they are run. For efficiency, the underlying machinery may run an equation just once, and store the value for subsequent accesses, using memoization. And if a particular attribute is not accessed, its equation might not be run at all. Therefore, any externally visible side effects within the equations will not have a well-defined behavior.

3.5 Example: Name analysis for state labels

In section 2 we discussed an attribute `target` for Transition objects, that should point to the appropriate target State object. This can be seen as a name analysis problem: We can view the states as declarations and the transitions as uses of those declarations. In addition to the `target` attribute we will define an analogous `source` attribute which points to the appropriate source State object. We start by declaring `target` and `source` as synthesized attributes of Transition. This definition would be easy if we had a parameterized attribute `State lookup(String label)` that would somehow find the appropriate State object for a certain label. Since we don't have enough information in Transition to define `lookup`, we make it an inherited attribute. In fact, we will declare `lookup` as an attribute of the superclass Declaration, since it might be useful also to the State subclass, as we will see in exercise 8. By looking at the abstract grammar, we see that the StateMachine node can have children of type Declaration, so it is the responsibility of StateMachine to define `lookup`. (In this case, StateMachine will be the root of the AST, so there are no further ancestors to which the definition can be delegated.)

In StateMachine, we can define `lookup` simply by traversing the declarations, locating the appropriate state. To do this we will introduce a synthesized attribute `State localLookup(String label)` for Declarations. Fig. 11 shows the resulting grammar. We use a JastAdd aspect to introduce the attributes and equations using inter-type declarations.

```
aspect NameAnalysis {
  syn State Transition.source() = lookup(getSourceLabel()); // R1
  syn State Transition.target() = lookup(getTargetLabel()); // R2
  inh State Declaration.lookup(String label); // R3

  eq StateMachine.getDeclaration(int i).lookup(String label) { // R4
    for (Declaration d : getDeclarationList()) {
      State match = d.localLookup(label);
      if (match != null) return match;
    }
    return null;
  }

  syn State Declaration.localLookup(String label) = null; // R5

  eq State.localLookup(String label) = // R6
    (label==getLabel()) ? this : null;
}
```

Fig. 11. An aspect binding each Transition to its source and target States.

There are a few things to note about the notation used:

syn, inh, eq The keywords `syn` and `inh` indicate declarations of synthesized and inherited attributes. The keyword `eq` indicates an equation defining the value of an attribute.

in-line equations Rules R4 and R6 define equations using the `eq` keyword. But equations can also be given in-line as part of the declaration of a synthesized attribute. This is the case in rules R1, R2, and R5.

equation syntax Equations may be written either using value syntax as in R1, R2, R5, and R6:

`attr = expr`,

or using method syntax as in R4:

`attr { ... return expr;}`

In both cases, full Java can be used to define the attribute value. However, there must be no external side-effects resulting from the execution of that Java code.

equations for inherited attributes R4 is an example of an equation defining an inherited attribute. The left-hand side of such an equation has the general form `Class.getChild().attr()`. This means that the equation is located in

`Class`, i.e., it can directly access attributes in `Class`. It defines the attribute `attr` of all nodes in the subtree `getChild()`, which is a method in the traversal API. In R4, the child is a list, and the traversal method therefore has an argument: `int i`, meaning that it applies to the `i`th Declaration in the list. In R4, the argument `i` is not used, because all the Declaration children should have the same value for `lookup`.

default and overriding equations Default equations can be supplied in superclasses and overridden in subclasses. R5 is an example of a default equation, applying to all Declaration nodes, unless overridden in a subclass. R6 is an example of overriding this equation for the State subclass.

Exercise 7. Consider the following state machine:

```
state S1;
state S2;
trans a: S1 -> S2;
```

Draw a picture similar to Fig. 9, but for this state machine, i.e., indicating the location of all attributes and equations, according to the grammar in Fig. 11. Draw also the values of the `source` and `target` attribute values. Check that these values agree with the equations.

Exercise 8. In a well-formed state machine AST, all State objects should have unique labels. Define a boolean attribute `alreadyDeclared` for State objects, which is true if there is a preceding State object of the same name.

Exercise 9. If there are two states with the same name, the first one will have `alreadyDeclared`=false, whereas the second one will have `alreadyDeclared`=true. Define another boolean attribute `multiplyDeclared` which will be true for both state objects, but false for uniquely named state objects.

3.6 More advanced name analysis

The name analysis for the state machine language is extremely simple, since there is only one global name space for state labels. However, the principle, using parameterized attributes like `lookup`, scales up to full programming languages. For example, to deal with block-structured scopes, the lookup attribute of a block can be defined to first look among the local declarations, and, if not found there, to delegate to the context, using the inherited lookup attribute of the block node itself. Similarly, object-oriented inheritance can be handled by delegating to a lookup attribute in the superclass. This general technique, using lookup attributes and delegation, is used in the implementation of the JastAddJ Java compiler. See [EH06] for details. That paper also covers type analysis for Java, also making use of parameterized attributes.

3.7 Attribute evaluation and caching

As mentioned earlier, the JastAdd user does not have to worry about in which order attributes are given values. The evaluation is carried out automatically. Given a well-defined attribute grammar, once the AST is built, all equations will hold, i.e., each attribute will have the value given by the right-hand side of its defining equation. From a performance or debugging perspective, it is, however, useful to know how the evaluation is carried out.

The evaluation algorithm is a very simple dynamic recursive algorithm, first suggested for ordinary Knuthian AGs [Jou84], but which works also in the presence of reference attributes. The basic idea is that equation right-hand sides are implemented as recursive functions, and when an attribute is called, its defining equation is run. The function call stack takes care of evaluating the attributes in the right order.

The use of object-orientation, as in JastAdd, makes the implementation of the algorithm especially simple, representing both attributes and equations as methods: For synthesized attributes, ordinary object-oriented dispatch takes care of selecting the appropriate equation method. For inherited attributes, there is some additional administration for looking up the appropriate equation method in the parent, or further up in the AST.

Two additional issues are taken care of during evaluation. First, attribute values can be cached for efficiency. If the attribute is cached, its value is stored the first time it is accessed. Subsequent accesses will return the value directly, rather than calling the equation method. In JastAdd, attributes are explicitly declared to be cached by adding the modifier `lazy` to their declaration. Attributes that involve heavy computations and are accessed more than once (with the same arguments, if parameterized) are the best candidates for caching. For the example in Fig. 11 we could define `source` and `target` as cached if we expect them to be used more than once by an application:

```
...
syn lazy State Transition.source() = ...
syn lazy State Transition.target() = ...
...
```

The second issue is dealing with circularities. In a well-defined attribute grammar, ordinary attributes must not depend on themselves, directly or indirectly. If they do, the evaluation would end up in an endless recursion. Therefore, the evaluator keeps track of attributes under evaluation, and raises an exception at runtime if a circularity is found. Due to the use of reference attributes, there is no general algorithm for finding circularities by analyzing the attribute grammar statically [Boy05].

4 Composite attributes

It is often useful to work with composite attribute values like sets, lists, maps, etc. In JastAdd, these composed values are often sets of node references. An example

is the `transitions` attribute of State, discussed in Section 2. It is possible to define composite attributes using normal synthesized and inherited attributes. However, often it is simpler to use *collection* attributes. Collection attributes allow the definition of a composite attribute to be spread out in several different places in an AST, each contributing to the complete composite value. Collection attributes can be used also for scalar values like integers and booleans, but using them for composite values, especially sets, is more common.

4.1 Representing composite attributes by immutable objects

We will use objects to represent composite attribute values like sets. However, when accessing these attributes, care must be taken to treat them as *immutable objects*, i.e., to only use their non-mutating operations. During the construction of the value, it is, however, fine to use mutating operations. For example, an equation can construct a set value by successively adding elements to a freshly created set object. Figure 12 shows a simplified[3] part of the API of the Java class `HashSet`.

```
class HashSet<E> implements Set{
  public HashSet(); // Constructor, returns a new empty set.

  // Mutating operations
  public void add(E e); // Adds the element e to this object.
  public void addAll(Set<E> s); // Adds all elements in s to this object.

  // Non-mutating operations
  public boolean contains(T e); // Returns true if this set contains e.
  public boolean equals(Set<E> s); // Returns true if this set has the
                                   // same elements as s.
}
```

Fig. 12. Simplified API for the Java class HashSet

4.2 A collection attribute: `transitions`

A *collection attribute* [Boy96,MEH09] has a composite value that is defined as a combination of *contributions*. The contributions can be located anywhere in the AST. If we would use ordinary equations, we would need to define attributes that in effect traverse the AST to find the contributions. With collection attributes, the responsibility is turned around: each contributing node declares its contribution to the appropriate collection attribute.

Fig. 13 shows how to define the `transitions` attribute as a collection.

[3] The actual API for HashSet has more general types for some parameters and returns booleans instead of void for some operations, and has many additional operations.

```
coll Set<Transition> State.transitions()       // R1
  [new HashSet<Transition>()] with add;

Transition contributes this                    // R2
  when source() != null
  to State.transitions()
  for source();
```

Fig. 13. Defining `transitions` as a collection attribute.

Rule R1 declares that `State` objects have a collection attribute `transitions` of type `Set<Transition>`. Its initial value (enclosed by square brackets) is `new HashSet<Transition>()`, and contributions will be added with the method `add`.

Rule R2 declares that `Transition` objects contribute themselves (`this`) to the `transitions` collection attribute of the `State` object `source()`, but only when `source()` is not equal to `null`.

We can note that the definition of `transitions` involves only the two node classes `State` and `Transition`. If we had instead used ordinary synthesized and inherited attributes to define `transitions`, we would have had to propagate information through StateMachine, using additional attributes. The collection attribute solution thus leads to a simpler solution, as well as less coupling between syntax node classes.

Exercise 10. Define an attribute `altTransitions` that is equivalent to `transitions`, but that uses ordinary synthesized and inherited attributes instead of collection attributes. Compare the definitions.

Via the `transitions` attribute, we can easily find the successor states of a given state. To obtain direct access to this information, we define an attribute `successors`. Figure 14 shows the definition of `successors` as an ordinary synthesized attribute, making use of `transitions`. An alternative definition would have been to define `successors` independently of `transitions`, using a collection attribute.

```
syn Set<State> State.successors() {
  Set<State> result = new HashSet<State>();
  for (Transition t : transitions()) {
    if (t.target() != null) result.add(t.target());
  }
  return result;
}
```

Fig. 14. Defining `successors`, by using `transitions`.

Exercise 11. Define an attribute `altSuccessors` that is equivalent to `successors`, but that uses a collection attribute. Compare the definitions.

4.3 Collection attribute syntax

Figure 15 shows the general syntax used for declaring collection attributes and contributions. For the *collection-attribute-declaration*, the *initial-object* should be a Java expression that creates a new object of type *type*. The *contributing-method* should be a one-argument method that mutates the *initial-object*. It must be *commutative*, i.e., the order of calls should be irrelevant and result in the same final value of the collection attribute. Optionally, a *rootclass* can be supplied, limiting the contributions to occur in the AST subtree rooted at the closest *rootclass* object above or at the *nodeclass* object in the AST. If no *rootclass* is supplied, contributions can be located anywhere in the AST.

In the *contribution-declaration*, the *expr* should be a Java expression that has the type of the argument of the *contributing-method*, as declared in the corresponding collection declaration (the one for *collection-nodeclass.attr()*). In the example, there is an `add` method in `Set<Transition>` which has the argument type `Transition`, so this condition is fulfilled. There can be one or more such contributions, separated by commas, and optionally they may be conditional, as specified in a `when` clause. The expression *ref-expr* should be a reference to a *collection-nodeclass* object. Optionally, the contribution can be added to a whole set of collection attributes by using the `each` keyword, in which case *ref-expr* should be a set of *collection-nodeclass* objects, or more precisely, it should be an object implementing Java's interface Iterable, and contain objects of type *collection-nodeclass*.

collection-attribute-declaration ::=
  '`coll`' *type nodeclass* '`.`' *attr* '`()`'
  '`[`' *initial-object* '`]`'
  '`with`' *contributing-method*
  ['`root`' *rootclass*]

contribution-declaration ::=
  contributing-nodeclass '`contributes`'
  (*expr* ['`when`' *cond*] , '`,`')$^+$
  '`to`' *collection-nodeclass* '`.`' *attr* '`()`'
  '`for`' ['`each`'] *ref-expr*

Fig. 15. Syntax for collection attributes and contributions

Exercise 12. Given the `successors` attribute, define a `predecessors` attribute for State, using a collection attribute. Hint: use the `for each` construct in the contribution.

Exercise 13. Collection attributes can be used not only for sets, but also for other composite types, like maps and bags, and also for scalar types like integers. Primitive types, like int and boolean in Java, need, however, to be wrapped in objects. Define a collection attribute that computes the number of transitions in a state machine.

Exercise 14. Define a collection attribute `errors` for StateMachine, to which different nodes in the AST can contribute objects describing static-semantic errors. The `errors` attribute could be a set of ErrorMessage objects, where ErrorMessage is an ordinary Java class. Give a suitable API for ErrorMessage. Transitions referring to missing source and target states are obvious errors. What other kinds of errors are there? Write a collection declaration and suitable contributions to define the value of `errors`.

For more examples of collection attributes, see the Metrics example, available at `jastadd.org`. This example implements Chidamber and Kemerer's metrics for object oriented programs [CK94]. The implementation is done as an extension to the JastAddJ compiler, and makes heavy use of collection attributes for computing the different metrics. Collection attributes are also used in the Flow Analysis example at `jastadd.org`, as described in [NNEHM09]. Here, *predecessors* in control-flow graphs, and *def* and *use* sets in dataflow, are defined using collection attributes.

4.4 Evaluation of collection attributes

When accessing a collection attribute, JastAdd automatically computes its value, based on the existing contribution declarations. In general, this involves a complete traversal of the AST to find the contributions, unless the scope of the collection is restricted, using a `root` clause in the collection declaration. To improve performance, several collection attributes can be computed in the same traversal, either completely or partially. Given that a particular instance c_i of a collection attribute c is accessed, the default behavior of JastAdd is to partially compute all instances of c, so that further traversal of the AST is unnecessary when additional instances of c are accessed. The algorithm used is called *two-phase joint evaluation* [MEH09]. It is sometimes possible to achieve further performance improvements by using other algorithm variants. For example, the evaluation of several different collection attributes can be grouped, provided that they do not depend on each other. See [MEH09] for more details.

5 Circular attributes

Sometimes, the definition of a property is circular, depending ultimately on itself: When we write down a defining equation for the property, we find that we need the same property to appear at the right-hand side of the equation, or in equations for attributes used by the first equation. In this case, the equations cannot be solved by simple substitution, as for normal synthesized and inherited

attributes, but a fixed-point iteration is needed. The variables of the equations are then initialized to some value, and assigned new values in an iterative process until a solution to the equation system is found, i.e., a fixed point.

The `reachable` attribute of State is an example of such a circularly defined property. In this section we will first look at how this property can be formulated and solved mathematically, and then how it can be programmed using JastAdd.

5.1 Circularly defined properties

To define reachability for states mathematically, suppose first that the state machine contains n states, $s_1..s_n$. Let $succ_k$ denote the set of states that can be reached from s_k through one transition. The set of reachable states for s_k, i.e., the set of states that can be reached via any number of transitions from s_k can then be expressed as follows:

$$reachable_k = succ_k \cup \bigcup_{s_j \in succ_k} reachable_j$$

We will have one such equation for each state $s_k, 1 \leq k \leq n$. If there is a cycle in the state machine, the equation system will be cyclic, i.e., there will be some *reachable* set that (transitively) depends on itself. We can compute a solution to the equation system using a least fixed point iteration. I.e., we use one *reachable* variable for each state, to which we initially assign the empty set. Then we interpret the equations as assignments, and iterate these assignments until no *reachable* variable changes value. We have then found a solution to the equation system. The iteration is guaranteed to terminate if we can place all possible values in a lattice of finite height, and if all the assignments are monotonic, i.e., if they never decrease the value of any *reachable* variable.

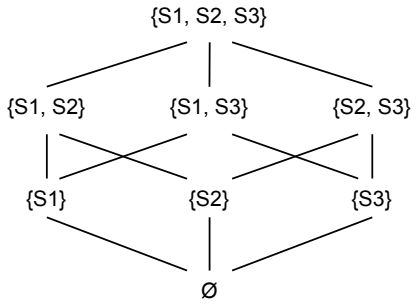


Fig. 16. The sets of states for the state machine of Fig. 3 are arranged in a lattice.

In this case, the values are sets of states, and they can be arranged in a lattice with the empty set at the bottom and the set of all states in the state machine

at the top. Fig. 16 shows the lattice for the state machine of Fig. 3. The lattice will be of finite height since the number of states in the state machine is finite. The assignments will be monotonic since the union operator can only lead to increasing values in the lattice. Because we start at the bottom (the empty set), we are furthermore guaranteed to find the *least* fixed point, i.e., the variables will stay at the lowest possible points in the lattice. If we have a cycle in the state machine, there may be additional uninteresting fixed points, for example by assigning the full set of states to *reachable* for all states on the cycle.

Exercise 15. For the state machine of Fig. 3, write down all the equations for *reachable*. Which are the variables of the equation system?

Exercise 16. What is the (least) solution to this equation system? Are there any more (uninteresting) solutions?

Exercise 17. Construct a state machine for which there is more than one solution to the equation system. What would be the least solution? What would be another (uninteresting) solution?

5.2 Circular attributes

In JastAdd, we can program circular properties like *reachable* by explicitly declaring the attribute as `circular`. and stating what initial value to use. The attribute will then automatically be evaluated using fixed-point iteration. Fig. 17 shows the definition of the attribute `reachable` for States.

```
syn Set<State> State.reachable() circular [new HashSet<State>()]; // R1

eq State.reachable() { // R2
  HashSet<State> result = new HashSet<State>();
  for (State s : successors()) {
    result.add(s);
    result.addAll(s.reachable());
  }
  return result;
}
```

Fig. 17. Defining `reachable` as a circular attribute.

Things to note:

syntax Synthesized, inherited and collection attributes can be declared as circular by adding the keyword `circular` after that attribute name. For synthesized and inherited attributes, an initial value also needs to be supplied, surrounded by square brackets as shown in the example above. For collection attributes, the initial object is used as the initial value.

caching Circular attributes are automatically cached, so adding the keyword `lazy` has no effect.

equals method The types used for circular attributes must have a Java `equals` method that tests for equality between two attribute values.

value semantics As usual, it is necessary to treat any accessed attributes as values, and to not change their contents. In the example, we set `result` to a freshly created object, and it is therefore fine to mutate `result` inside the equation. Note that if we instead had initialized `result` to the set of successors, we would have had to be careful to set `result` to a fresh clone of the `successors` object.[4]

termination JastAdd does not currently check that the equations are monotonic, or that the values can be viewed as in a lattice of finite height. If these conditions are not fulfilled, it may result in erroneous evaluation or non-termination.

Exercise 18. Define an attribute `altReachable` that is equivalent to `reachable`, but that uses a circular collection attribute. Hint: make use of the `predecessors` attribute defined in exercise 12.

For more examples of JastAdd's circular attributes, you may look at the the Flow Analysis example at `jastadd.org` where intraprocedural control flow and dataflow is defined as an extension to JastAddJ, as described in [NNEHM09]. Here, the *in* set is defined as a circular attribute, and the *out* set as a circular collection attribute. In [MH07], there are examples of defining the properties *nullable*, $first$, and $follow$ for nonterminals in context-free grammars, using JastAdd circular attributes. The *nullable* property is defined using a boolean circular attribute, and the two others as set-valued circular attributes. A variant of $follow$ is defined in [MEH09] using circular collection attributes.

6 Conclusions

In this tutorial we have covered central attribution mechanisms in JastAdd, including synthesized, inherited, reference, parameterized, collection, and circular attributes. With these mechanisms you can address many advanced problems in compilers and other language tools. There are some additional mechanisms in JastAdd that are planned to be covered in a sequel of this tutorial:

Rewrites [EH04], allow sub ASTs to be replaced conditionally, depending on attribute values. This is useful when the AST constructed by the parser is not specific enough, or in order to normalize language constructs to make further compilation easier.

[4] In order to avoid having to explicitly create fresh objects each time a new set value is computed, we could define an alternative Java class for sets with a larger nonmutating API, e.g., including a `union` function that automatically returns a new object if necessary. Such an implementation could make use of persistent data structures[DSST86], to efficiently represent different values.

Nonterminal attributes [VSK89] allow the AST to be extended dynamically, defining new AST nodes using equations. This is useful for macro expansion and transformation problems. In JastAddJ, nonterminal attributes are used for adding nodes representing instances of generic types.

Inter-AST references [ÅEH09] allow nodes in a new AST to be connected to nodes in an existing AST. This is useful when creating transformed ASTs: nodes in the transformed AST can have references back to suitable locations in the source AST, giving access to information there.

Interfaces. Attributes and equations can be defined in interfaces, rather than in AST classes, allowing reuse of language independent computations, and connection to language independent tools.

Refine. Equations in existing aspects can be *refined* in new aspects. This is similar to object-oriented overriding, but without having to declare new subclasses. Refines are useful for adjusting the behavior of an aspect when reusing it for a new language or tool.

The declarative construction of an object-oriented model is central when programming in JastAdd. The basic structure is always the abstract syntax tree (AST), but through the reference attributes, graphs can be superimposed. In this tutorial we have seen this through the addition of the `source` and `target` edges, and the `transitions`, `successors`, and `reachable` sets. Similar techniques are used to implement compilers for programming languages like Java. Here, each use of an identifier can be linked to its declaration, each class declaration to its superclass declaration, and edges can be added to build control-flow and dataflow graphs. Once these graphs have been defined, further attribute definitions are often made in terms of those graph structures rather than in terms of the tree structure of the AST. An example was defining `transitions` in terms of `source`.

An important design advice is to focus on thinking declaratively when programming in JastAdd. Think first about what attributes you would like the AST to have. Then, in defining these attributes, think of what other attributes that would be useful, in order to make your equations simple. This will lead to the addition of new attributes. In this tutorial, we have mostly worked in the other direction, in order to present simple mechanisms before more complex ones. For a real situation, where you already know about the JastAdd mechanisms, you might have started out with the `reachable` attribute instead. In order to define it, it would have been useful to have the `successors` attribute. To define the `successors` attribute, you find that you need the `transitions` and `target` attributes, and so on.

As for normal object-oriented programming, naming is essential. Try to pick good descriptive names of both your AST classes and your attributes, so that the code you write is readable, and the APIs that the attributes produce will be simple and natural to use. For each attribute that you implement, you can write test cases that build up some example ASTs and test that the attributes get the intended values in different situations, so that you are confident that you have got your equations right.

JastAdd has been used for implementing both simple small languages and advanced programming languages. The implementation of our extensible Java compiler, JastAddJ, has been driving the development of JastAdd, and has motivated the introduction of many of the different mechanisms and made it possible to benchmark them on large programs [EH04,MH07,MEH09,NNEHM09]. Other advanced languages are being implemented as well, most notably an ongoing open-source implementation of the language Modelica which is used for describing physical models using differential equations [Mod09a,Mod09b,ÅEH09]. For more information about JastAdd, see `jastadd.org`.

Acknowledgments. The JastAdd system was jointly developed with my former PhD students Torbjörn Ekman and Eva Magnusson. For this tutorial I am particularly grateful to Torbjörn for joint work on constructing the state machine example. Thanks also to the students Nicolas Mora and Philip Nilsson whose comments helped me improve the readability of the tutorial.

References

[ÅEH09] J. Åkesson, T. Ekman, and G. Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 2009.

[Boy96] John Tang Boyland. *Descriptional Composition of Compiler Components.* PhD thesis, University of California, Berkeley, September 1996. Available as technical report UCB//CSD-96-916.

[Boy05] John Tang Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[DSST86] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, STOC 86*, pages 109–121. ACM, 1986.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 144–169. Springer, 2004.

[EH06] Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005*, volume 4143 of *LNCS*. Springer, 2006.

[EH07a] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA 2007*, pages 1–18. ACM, 2007.

[EH07b] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Proceedings of TOOLS Europe 2007, Journal of Object Technology*, 6(7), 2007.

[Ekm06] Torbjörn Ekman. *Extensible Compiler Construction.* PhD thesis, Lund University, Sweden, June 2006.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN symposium on Compiler Construction*, pages 85–98. ACM Press, 1986.

[Hed00] Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[Jou84] Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 1984.

[Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (1971).

[MEH09] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. Demand-driven evaluation of collection attributes. *Automated Software Engineering*, 16(2):291–322, 2009.

[MH07] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Science of Computer Programming*, 68(1):21–37, 2007.

[Mod09a] The Modelica Association, 2009. http://www.modelica.org.

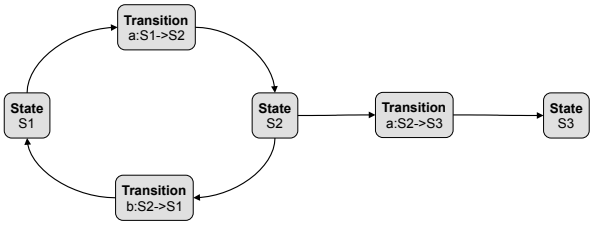
[Mod09b] Modelon AB. JModelica Home Page, 2009. `http://www.jmodelica.org`.

[NNEHM09] E. Nilsson-Nyman, T. Ekman, G. Hedin, and E. Magnusson. Declarative intraprocedural flow analysis of Java source code. *Electronic Notes in Theoretical Computer Science*, 238(5):155–171, Oct 2009.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of PLDI '89*, pages 131–145. ACM Press, 1989.

A Answers to selected exercices

Exercise 1



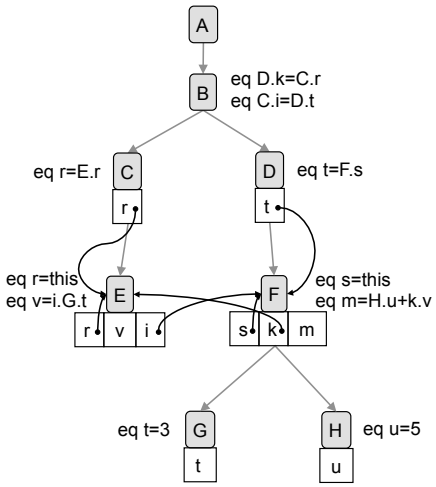
Exercise 3

```
C.v = 18
C.i = 7
E.s = 26
E.i = 18
F.t = 3
G.u = 5
```

Exercise 4

`B`, `D`, `F`, and `G`.

Exercise 6

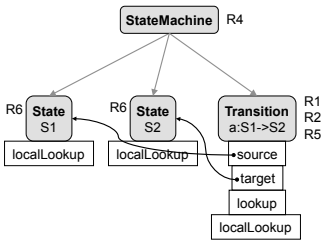


`E` and `F` are on a cycle via their attributes `i` and `k`.
Values of non-reference attributes:

```
E.v = 3
F.m = 8
G.t = 3
H.u = 5
```

Examples of non-local dependencies: the value of `E.v` depends directly on the value of `G.t`, and `F.m` depends directly on `E.v`.

Exercise 7



Exercise 10

```
syn Set<Transition> State.altTransitions() = transitionsOf(this);
inh Set<Transition> State.transitionsOf(State s);

eq StateMachine.getDeclaration(int i).transitionsOf(State s) {
  HashSet<Transition> result = new HashSet<Transition>();
  for (Declaration d : getDeclarationList()) {
    Transition t = d.transitionOf(s);
    if (t != null) result.add(t);
  }
  return result;
}

syn Transition Declaration.transitionOf(State s) = null;
eq Transition.transitionOf(State s) {
  if (source() == s)
    return this;
  else
    return null;
}
```

We see that the definition of `altTransitions` is more complex than that of `transitions`: two help attributes are needed: the inherited `transitionsOf`

and the synthesized `transitionOf`. Furthermore, we see that the definition of `altTransitions` is more coupled in that it relies on both the existence of the StateMachine nodeclass, and on its child structure.

Exercise 11

```
coll Set<State> State.altSuccessors()
  [new HashSet<State>()] with add;

Transition contributes target()
  when target() != null && source() != null
  to State.altSuccessors()
  for source();
```

In this case, the definitions using ordinary attributes and collection attributes have about the same complexity and coupling.

Exercise 12

```
coll Set<State> State.predecessors()
  [new HashSet<State>()] with add;

State contributes this
  to State.predecessors()
  for each successors();
```

Exercise 15

$$
\begin{aligned}
reachable_1 &= \{S_2\} \cup reachable_2 \\
reachable_2 &= \{S_1, S_3\} \cup reachable_1 \cup reachable_2 \\
reachable_3 &= \emptyset
\end{aligned}
$$

Exercise 16

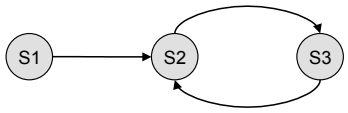
The least (and desired) solution is

$$
\begin{aligned}
reachable_1 &= \{S_1, S_2, S_3\} \\
reachable_2 &= \{S_1, S_2, S_3\} \\
reachable_3 &= \emptyset
\end{aligned}
$$

There are no additional solutions since the attributes that are circular ($reachable_1$ and $reachable_2$) have the top value in the lattice (the set of all states).

Exercise 17

This state machine has more than one solution for *reachable*.



The equation system is:

$$reachable_1 = \{S_2\} \cup reachable_2$$
$$reachable_2 = \{S_3\} \cup reachable_3$$
$$reachable_3 = \{S_2\} \cup reachable_2$$

The least (and desired) solution is:

$$reachable_1 = \{S_2, S_3\}$$
$$reachable_2 = \{S_2, S_3\}$$
$$reachable_3 = \{S_2, S_3\}$$

An additional (and uninteresting) solution also includes S_1:

$$reachable_1 = \{S_1, S_2, S_3\}$$
$$reachable_2 = \{S_1, S_2, S_3\}$$
$$reachable_3 = \{S_1, S_2, S_3\}$$

Exercise 18

```
coll Set<State> State.altReachable() circular
  [new HashSet<State>()] with addAll;

State contributes union(asSet(this),altReachable())
  to State.altReachable()
  for each predecessors();
```

In the above solution we have made use of two auxiliary functions: `asSet` and `union`. It would have been nice if these functions had already been part of the Java Set interface, but since they are not, we define them as functions in ASTNode as shown below, making them available to all AST nodes. A nicer solution can be achieved by designing new alternative Java classes and interfaces for sets.

```
Set<State> ASTNode.asSet(State o) {
  HashSet<State> result = new HashSet<State>();
  result.add(o);
  return result;
}

Set<State> ASTNode.union(Set<State> s1, Set<State> s2) {
  HashSet<State> result = new HashSet<State>();
  for (State s: s1) result.add(s);
  for (State s: s2) result.add(s);
  return result;
}
```