An Introductory Tutorial on JastAdd Attribute Grammars

Görel Hedin

Lund University, Sweden
gorel@cs.lth.se

Preprint, May 1, 2010. To appear in Postproceedings of GTTSE 2009.
The final publication will be available at www.springerlink.com

Abstract. JastAdd is an open-source system for generating compilers and other language-based tools. Its declarative specification language is based on reference 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, and how to use them when solving key problems in building language-based tools. A simple state machine language is used as a running example, showing the essence of name analysis, adding graphs to the abstract syntax tree, and computing circular properties like reachability. Exercises are included, and code for the examples is available online.

Keywords: attribute grammars, language-based 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 support for reference-valued attributes is of fundamental importance to JastAdd, because they allow explicit definition of graph properties of a program. Examples include linking identifier uses to their declaration nodes, and representing call graphs and dataflow graphs. AST nodes are objects, and the resulting data structure, including attributes, is in effect an object-oriented graph model, rather than only a simple syntax tree.

While there are many technical papers on individual JastAdd mechanisms and advanced applications, this is the first tutorial paper. The goal is to give an introduction to JastAdd and its core attribute grammar mechanisms, to explain how to program and think declaratively using this approach, and to illustrate how key problems are solved.

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.

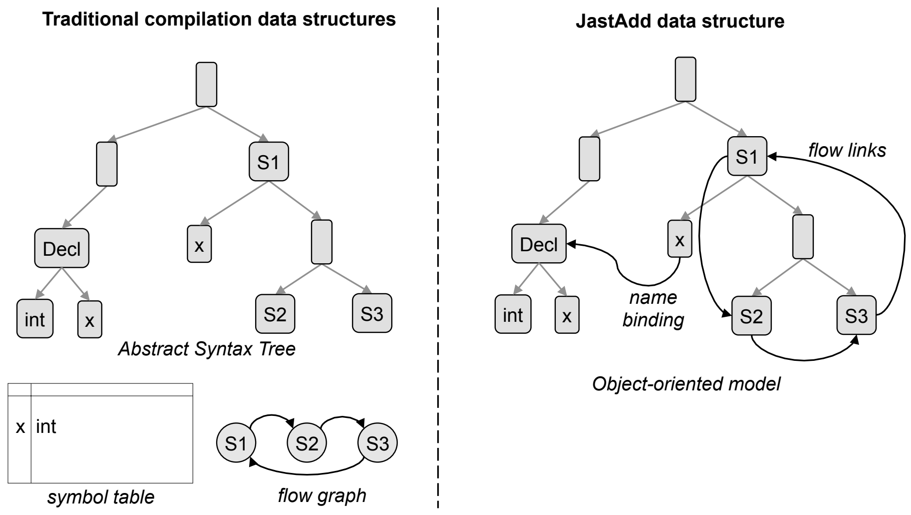


Fig. 1. In JastAdd, compilation data structures are embedded as reference attributes in the AST, resulting in an object-oriented model of the program.

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 [EH07b], 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 [EH07a], 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 toy languages like the state machine language that will be used in this tutorial, to full-blown general-purpose programming languages like Java. Because of the modularization support, it is particularly attractive to use JastAdd to build extensible languages and tools.

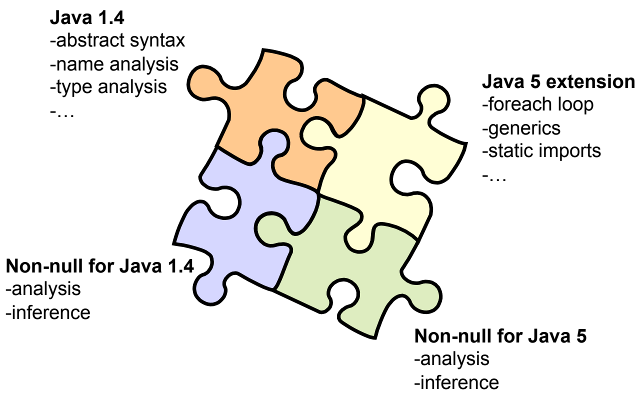


Fig. 2. Each component has modules containing abstract syntax rules, attributes, and equations. To construct a compiler supporting non-null analysis and inference for Java 5, all modules in the four components are used.

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.

The tutorial includes exercises, and solutions are provided in the appendix. We recommend that you try to solve the exercises on your own before looking at the solutions. The code for the state machine language and the related exercise solutions is available for download at `http://jastadd.org`. We recommend that you download it, and run the examples and solutions as you work through the tutorial. Test cases and the JastAdd tool are included in the download. See the README file for further instructions.

1.4 Brief historical notes

After Knuth's seminal paper on attribute grammars [Knu68], the area received an intense interest from the research community. A number of different evaluation algorithms were developed, for full Knuth-style AGs as well as for subclasses thereof. One of the most influential subclasses is Kastens' *ordered attribute grammars* (OAGs) [Kas80]. OAGs are powerful enough for the implementation of full programming languages, yet allow the generation of efficient static attribute evaluators. Influential systems based on OAGs include the *GAG* system which was used to generate front ends for Pascal and Ada [KHZ82,UDP^{+}82], and the *Synthesizer Generator* which supports incremental evaluation and the generation of interactive language-based editors [RT84]. For surveys covering this wealth of research, see Deransart *et al.* [DJL88], and Paakki [Paa95].

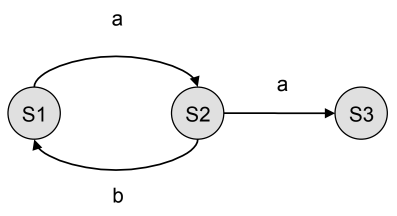
JastAdd belongs to a newer generation of attribute grammar systems based on reference attributes. Support for reference-like attributes were developed independently by a number of researchers: Hedin's reference attributes [Hed94,Hed00], Poetzsch-Heffter's occurrence attributes [PH97], and Boyland's remote attributes [Boy96].

Other landmark developments of strong importance for JastAdd include Jourdan's dynamic evaluation algorithm [Jou84], Farrow's circular attributes [Far86], Vogt, Swierstra and Kuiper's higher-order attributes [VSK89], and Boyland's collection attributes [Boy96].

In addition to JastAdd, there are several other current systems that support reference attributes, including Silver [WBGK10], Kiama [SKV09], and ASTER [KSV09]. While these systems use quite different syntax than JastAdd, and support a partly different set of features, this tutorial can hopefully be of value also to users of these systems: the main ideas for how to think declaratively about reference attributes, and how to solve problems using them, still apply.

2 Running example: A state machine language

As a running example, we will use a small state machine language. Figure 3 shows a sample 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. A sample state machine and its textual representation.

2.1 Abstract grammar

Given the textual representation of a 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 parsing the text into a syntax tree representation, and then add reference attributes to represent the graph properties. Fig. 4 shows a typical EBNF context-free grammar for the textual representation.

A corresponding abstract grammar, written in JastAdd syntax, is shown in Fig. 5. The nonterminals and productions are here written as *classes*, replacing alternative productions by subclassing: `StateMachine` is a class containing a list of `Declarations`. `Declaration` is 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. An AST consists of a tree of objects of these classes. A parser that builds the AST from a text can be generated using an ordinary parser generator, building the AST in the semantic actions.

```
<statemachine> ::= <declaration>*;
<declaration> ::= <state> | <transition>;
<state> ::= "state" ID ";"
<transition> ::= "trans" ID ":" ID "->" ID ";";
ID = [a-zA-Z][a-zA-Z0-9]*
```

Fig. 4. EBNF context-free grammar for the state machine language

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

Fig. 5. JastAdd abstract grammar for the state machine language

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 6 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.

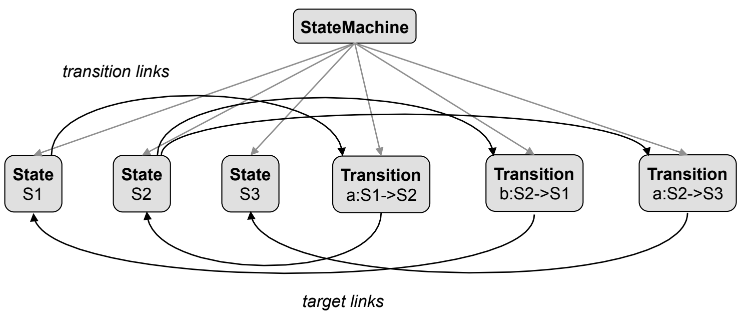


Fig. 6. The state machine graph is embedded in the object-oriented model.

Exercise 1. In Figure 6, 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 7 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. 7. Parts of the 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. 8.

```
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. 8. 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`.[1]

We can now write the main program that constructs the AST and prints the reachable information, as shown in Fig. 9. For illustration, we have used the construction API directly here to manually construct the AST for a particular test program. For real use, a parser should be integrated. This is straightforward: build the AST in the semantic actions of the parsing grammar, using the same JastAdd construction API. Any Java-based parser generator can be used, provided it allows you to place arbitrary Java code in its semantic actions. 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 straightforward solution is to write a 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. 9. A main program that builds an AST and then accesses attributes.

Exercise 2. Given the API in Fig. 7, 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?

[1] This syntax for inter-type declarations is borrowed from AspectJ [KHH$^+$01]. 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.

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

The value of an attribute a is defined by a directed equation $a = e(b_1, ..., b_n)$, where the left-hand side is an attribute and the right-hand side is an expression e over zero or more attributes b_k in the AST. In JastAdd, the 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. Attributes are declared as either *synthesized* or *inherited.* A synthesized attribute is defined by an equation in the node itself, whereas an inherited attribute is defined by an equation in an ancestor node.

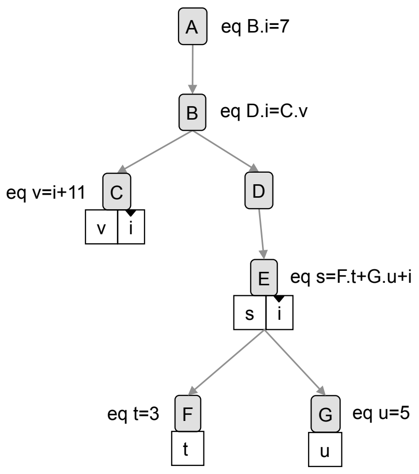


Fig. 10. The attributes `C.v`, `E.s`, `F.t`, and `G.u` are synthesized and have defining equations in the node they belong to. The attributes `C.i` and `E.i` are inherited (indicated by a downward-pointing black triangle), and are defined by equations in `A` and `B`, respectively. For `E.i`, the equation in `B` shadows the one in `A`, see the discussion below.

Most attributes we introduce will be synthesized. 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, a shorthand is used so that the equation defining an inherited attribute, say `E.i`, does not have to be located in the immediate parent of `E`, but can be in any ancestor of `E`, on the way from the parent up to the root. If several of these nodes have an equation for `i`, the closest one to `E` will apply, shadowing equations further up in the AST. See Fig. 10. Thus, an equation $child.i = expression$ actually defines the inherited i attribute for *all* nodes in the *child* subtree that declare the i attribute. This shorthand makes it possible to avoid cluttering the grammar with so called *copy rules*, i.e., equations that merely copy a value from a node to its children. Most attribute grammar systems have some kind of shorthand to avoid such copy rules. There are additional shorthands for this in JastAdd, for example allowing a single equation to be used to define an inherited attribute of *all* its children subtrees.

Exercise 3. What will be the values of the attributes in Fig. 10?

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. 10 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? (This is a difficult exercise, but worth thinking about.)

3.2 Reference attributes

In JastAdd, synthesized and inherited attributes are generalized in several ways, as compared to the classical 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 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 allow an AST to be extended to a graph in a declarative way. Also cyclic graphs can be defined, as in the example in Figure

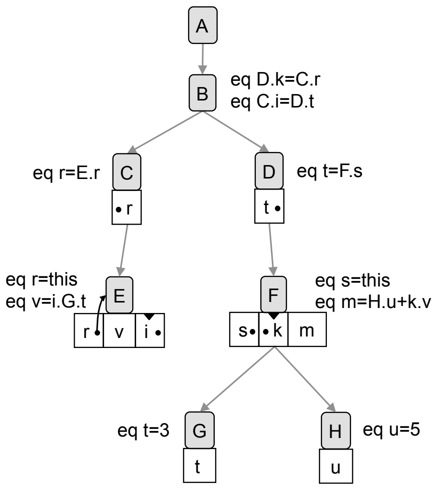


Fig. 11. Reference attributes are indicated with a small dot beside the name. An arrow from a dot to a node illustrates the value of the reference attribute: `E.r` refers to `E`, due to the equation `r=this` in `E`.

11 (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 11. 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, typically different for different parameter values. Conceptually, there is one value of `lookup` for each possible `String` value. In practice, only a few of these `lookup` values will actually be computed, because attribute evaluation is performed on demand (see Section 3.8).

By accessing 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.

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 attributes, 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, i.e., if the node itself should define the attribute, or if the definition is delegated to an ancestor. If all the information needed is available inside the subtree rooted by the node, the attribute should be synthesized. If all the information is instead available outside this subtree, make the attribute inherited. Finally, if both information inside and outside are needed, make the attribute synthesized, and introduce one or more inherited attributes to capture the information needed from outside.

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 synthesized 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. The definition is in this case delegated to an ancestor node.

3.5 Integration with Java

The JastAdd specification language builds on top of Java. 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, the right-hand side is written either as a Java expression or as a Java method body.

Although ordinary Java code is used for the right-hand side of an equation, an important requirement is that it must not introduce any externally visible side effects such as changing fields of AST nodes or changing global data. I.e., its effect should be equivalent to the evaluation of a side-effect-free expression. The reason for this restriction is that equations represent definitions of values, 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 memoize the values, i.e., run an equation just once, and store the value for subsequent accesses. And if a particular attribute is not accessed, its equation might not be run at all. Therefore, introducing externally visible side effects within the equations will not have a well-defined behavior, and may lead to very subtle bugs. The current JastAdd version (R20100416) does not check for side-effects in equations, but leaves this responsibility to the user. In principle, a number of static checks for this could be added, but this is an area of future work.

3.6 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. 12 shows the resulting grammar. We use a JastAdd aspect to introduce the attributes and equations using inter-type declarations.

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.

```
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.equals(getLabel())) ? this : null;
}
```

Fig. 12. An aspect binding each Transition to its source and target States.

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, as mentioned in Section 3.5, there must be no external side-effects resulting from the execution of that Java code. Even if R4 uses the method body syntax with a loop and an assignment, it is easy to see that there are no external side-effects: only the local variables `d` and `match` are modified.

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

```
A.getC().attr()
```

meaning that it is an equation in `A` which defines the `attr` attribute in the subtree rooted at the child `C` of the `A` node. If `C` is a list, the general form includes an argument `int i`:

```
A.getC(int i).attr()
```

meaning that the equation applies to the ith child of the list. The right-hand side of the equation is within the scope of `A`, allowing the API of

`A` to be accessed directly. For example, in R4, the AST traversal method `getDeclarationList()` is accessed. The argument `i` is not used in this equation, since 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;
trans a: S1 -> S2;
state S2;
```

Draw a picture similar to Fig. 10, but for this state machine, i.e., indicating the location of all attributes and equations, according to the grammar in Fig. 12. Draw also the reference values of the `source` and `target` attributes. 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.

3.7 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, it illustrates the typical solution for name analysis in JastAdd: using inherited `lookup` attributes, and delegation to other attributes, like `localLookup`. This solution 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.

3.8 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 Knuth-style 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 can be explicitly declared to be cached by adding the modifier `lazy` to their declaration. It is also possible to cache all attributes by using an option to the JastAdd system. 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. 12 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, see Exercise 13, 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. When *accessing* these attributes, great care must be taken to treat them as *immutable objects*, i.e., to only use their non-mutating operations. However, during the *construction* of the value, it is 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 13 shows a simplified[2] 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. 13. 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. 14 shows how to define the `transitions` attribute as a collection.

[2] 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. 14. 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 15 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. 15. 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 16 shows the syntax used for declaring collection attributes and contributions. For the *collection-attribute-declaration*, the *initial-object* should be a Java expression that creates a fresh 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. 16. 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 `numberOfTransitions` 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 strings describing static-semantic errors. 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 Java 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*, i.e., 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.

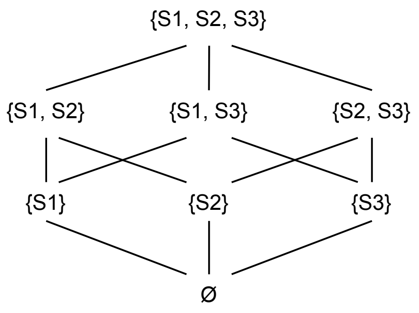


Fig. 17. The sets of states for the state machine of Fig. 3 are arranged in a lattice.

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. 17 shows the lattice for the state machine of Fig. 3.

The lattice will be of finite height since the number of states in any given 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. 18 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. 18. 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 the 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.[3]

termination For attributes declared as circular, it would be nice to have static checks for the requirements of equation monotonicity and finite-height lattices. Currently, JastAdd does not support any such checks, but leaves this as a responsibility of the user. This means that if these requirements are not met, it may result in erroneous evaluation or non-termination during attribute evaluation. Boyland's APS system provides some support in this direction by requiring circular attributes to have predefined lattice types, like *union* and *intersection* lattices for sets, and *and* and *or* lattices for booleans [Boy96]. Similar support for JastAdd is part of future work.

Attributes that are not declared as circular, but which nevertheless happen to be defined circularly, are considered erroneous. To statically check for the existence of such attributes is, in general, an undecidable problem in the presence of reference attributes [Boy05]. In JastAdd, such circularities are detected dynamically, raising an exception at evaluation time.

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 *flow analysis* example at `jastadd.org` where intraprocedural control flow and dataflow is defined as an extension to the JastAddJ Java compiler, 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.

[3] 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.

6 Conclusions and outlook

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.

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 the JastAddJ Java compiler, nonterminal attributes are used for adding nodes representing instances of generic types [EH07b].

Inter-AST references [ÅEH10] 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 supporting 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.

The use of Java as the host language for writing equations is very powerful, allowing existing Java classes to be used for data types, and for connecting the attributed AST to imperatively implemented tools. At the same time, it is extremely important to understand the declarative semantics of the attribute grammars, and to watch out to not introduce any external side-effects in the equations. In particular, when dealing with composite values that are implemented using objects, it is very important to distinguish between their mutating and non-mutating operations, so that accessed attributes are not mutated by mistake.

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 [Mod10,JMo09,ÅEH10]. 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, and to the anonymous reviewers who gave very detailed and valuable feedback on earlier drafts of this paper.

References

[ÅEH10] J. Åkesson, T. Ekman, and G. Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 73(1-2):21–38, 2010.

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

[Boy05] J. T. 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.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer Verlag, 1988.

[DSST86] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. 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] T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 144–169. Springer, 2004.

[EH06] T. Ekman and G. 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*, pages 422–436. Springer, 2006.

[EH07a] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Proceedings of TOOLS Europe 2007, Journal of Object Technology*, 6(9):455–475, 2007.

[EH07b] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA 2007*, pages 1–18. ACM, 2007.

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

[Far86] R. 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.

[Hed94] G. Hedin. An Overview of Door Attribute Grammars. In Peter A. Fritzson, editor, *5th Int. Conf. on Compiler Construction (CC' 94)*, volume 786 of *LNCS*, pages 31–51, Edinburgh, April 1994. Springer Verlag.

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

[JMo09] JModelica.org, Modelon AB, 2009. `http://www.jmodelica.org`.

[Jou84] M. 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.

[Kas80] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980. See also: Bericht 7/78, Institut für Informatik II, University Karlsruhe (1978).

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355. Springer, 2001.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science.* Springer Verlag, 1982.

[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).

[KSV09] L. C. L. Kats, A. M. Sloane, and E. Visser. Decorated attribute grammars: Attribute evaluation meets strategic programming. In *Proceedings of the 18th International Conference on Compiler Construction (CC 2009)*, Lecture Notes in Computer Science, pages 142–157, York, United Kingdom, March 2009. Springer-Verlag.

[MEH09] E. Magnusson, T. Ekman, and G. 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.

[Mod10] The Modelica Association, 2010. `http://www.modelica.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.

[Paa95] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[PH97] A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34:737–772, 1997.

[RT84] T. Reps and T. Teitelbaum. The Synthesizer Generator. In *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, pages 42–48. ACM press, Pittsburgh, PA, April 1984.

[SKV09] A. M. Sloane, L. C. L. Kats, and E. Visser. A pure object-oriented embedding of attribute grammars. In T. Ekman and J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, 2009.

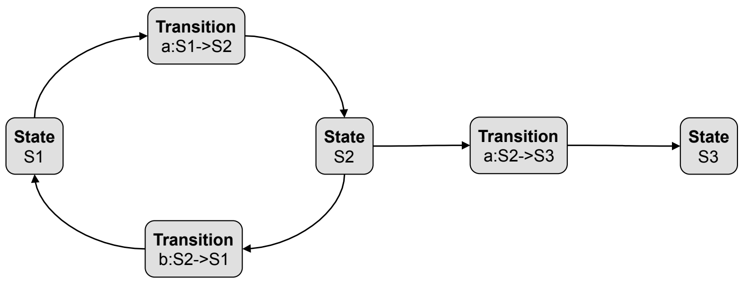
[UDP$^+$82] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein, and W. Kirchgässner. *An Attribute Grammar for the Semantic Analysis of Ada*, volume 139 of *Lecture Notes in Computer Science*. Springer Verlag, 1982.

[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.

[WBGK10] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54, 2010.

A Solutions to exercices

Exercise 1



Exercise 2

Here is an aspect defining a method `printInfoAboutCycles` for `StateMachine`:

```
aspect PrintInfoAboutCycles {
  public void StateMachine.printInfoAboutCycles() {
    for (Declaration d : getDeclarationList()) {
      d.printInfoAboutCycles();
    }
  }

  public void Declaration.printInfoAboutCycles() {}

  public void State.printInfoAboutCycles() {
    System.out.print("State "+getLabel()+" is ");
    if (!reachable().contains(this)) {
      System.out.print("not ");
    }
    System.out.println("on a cycle.");
  }
}
```

The main program parses an inputfile, then calls the `printInfoAboutCycles` method:

```
package exampleprogs;
import AST.*;
import java.io.*;

public class Compiler {
  public static void main(String[] args) {
    String filename = getFilename(args);

    // Construct the AST
    StateMachine m = parse(filename);

    // Print info about what states are on cycles
    m.printInfoAboutCycles();
  }

  public static String getFilename(String[] args) { ... }

  public static StateMachine parse(String filename) { ... }
}
```

Running the "compiler" on the input program of Fig. 3, gives the following output:

```
State S1 is on a cycle.
State S2 is on a cycle.
State S3 is not on a cycle.
```

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 5

There are many possible algorithms for computing attribute values in an AST. Here are some alternatives:

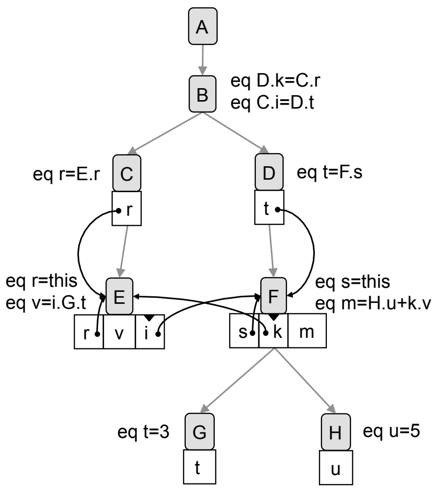
Dynamic, with explicit depedency graph Add dependency edges between all attributes in an AST according to the equations. For example, for an equation $a = f(b, c)$, the two edges (b, a) and (c, a) are added. Then run all the equations as assignments in topological order, starting with equations with no incoming edges. This kind of algorithm is called a *dynamic* algorithm because we use the dependencies of an actual AST, rather than only *static* dependencies that we can derive from the grammar.

Static Compute a conservative approximation of a dependency graph, based on the attribute grammar alone, so that the graph holds for any possible AST. Then compute a scheduling for in what order to evaluate the attributes based on this graph. This kind of algorithm is called a *static* algorithm, since it only takes the grammar into account, and not the actual AST. It might be more efficient than the dynamic algorithm since the actual dependencies in an AST do not need to be analyzed. On the other hand, it will be less general because of the approximation. There are many different algorithms that use this general approach. They differ in how they do the approximation, and thereby in how general they are.

Dynamic, with implicit dependency graph Represent each attribute by a function, corresponding to the right-hand side of its defining equation. To evaluate an attribute, simply call its function. Recursive calls to other attributes will automatically make sure the attributes are evaluated in topological order. This algorithm is also dynamic, but does not require building an explicit dependency graph.

The JastAdd system uses this latter algorithm, see Section 3.8 for details. The other two alternatives are not powerful enough to handle arbitrary reference attributes.

Exercise 6

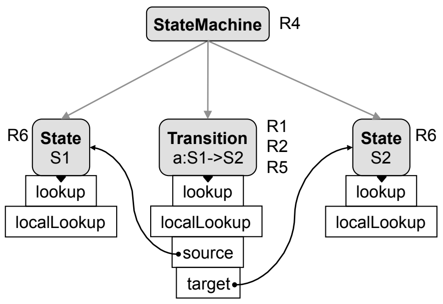


E and F are on a cycle via their attributes i and k.
Values of non-reference attributes:

```
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 8

```
syn boolean State.alreadyDeclared() =
    lookup(this.getLabel()) != this;
```

Exercise 9

A state is multiply declared if it either is declared already, or if it has a later namesake. To find out if it has a later namesake, we define a new attribute `lookupForward` that only traverses declarations *after* the state. Note that the equation for this attribute makes use of the argument `i` to start traversing at the appropriate position in the list.

```
syn boolean State.multiplyDeclared() =
    alreadyDeclared() || hasLaterNamesake();

syn boolean State.hasLaterNamesake() =
    lookupForward(getLabel()) != null;

inh State Declaration.lookupForward(String label);

eq StateMachine.getDeclaration(int i).lookupForward(String label) {
  for (int k = i+1; k<getNumDeclaration(); k++) {
    Declaration d = getDeclaration(k);
    State match = d.localLookup(label);
    if (match != null) return match;
  }
  return null;
}
```

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 13

To define the collection, we introduce a class `Counter` that works as a wrapper for Java integers. To give `Transitions` access to their enclosing state machine node, in order to contribute the value 1 to the collection, we introduce an inherited attribute `theMachine`. Finally, the synthesized attribute `numberOfTransitions` simply accesses the value of the `Counter`.

```
syn int StateMachine.numberOfTransitions() =
    numberOfTransitionsColl().value();
coll Counter StateMachine.numberOfTransitionsColl()
     [new Counter()] with add;

Transition contributes 1
  to StateMachine.numberOfTransitionsColl()
  for theMachine();

inh StateMachine Declaration.theMachine();
eq  StateMachine.getDeclaration(int i).theMachine() = this;

class Counter {
  private int value;
  public Counter() { value = 0; }
  public void add(int value) { this.value += value; }
  public int value() { return value; }
}
```

Exercise 14

Here we have simply used a set of strings to represent the error messages. In addition to missing declarations of states, error messages are added for states that are declared more than once.

```
coll Set<String> StateMachine.errors()
     [new HashSet<String>()] with add;

State contributes getLabel()+" is already declared"
when alreadyDeclared()
to StateMachine.errors()
for theMachine();

Transition contributes "Missing declaration of "+getSourceLabel()
```

```
when source() == null
to StateMachine.errors()
for theMachine();

Transition contributes "Missing declaration of "+getTargetLabel()
when target() == null
to StateMachine.errors()
for theMachine();
```

Exercise 15

$$\begin{aligned} \mathit{reachable}_1 &= \{S_2\} \cup \mathit{reachable}_2 \\ \mathit{reachable}_2 &= \{S_1, S_3\} \cup \mathit{reachable}_1 \cup \mathit{reachable}_2 \\ \mathit{reachable}_3 &= \emptyset \end{aligned}$$

Exercise 16

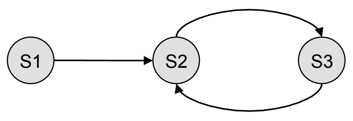
The least (and desired) solution is

$$\begin{aligned} \mathit{reachable}_1 &= \{S_1, S_2, S_3\} \\ \mathit{reachable}_2 &= \{S_1, S_2, S_3\} \\ \mathit{reachable}_3 &= \emptyset \end{aligned}$$

There are no additional solutions since the attributes that are circular ($\mathit{reachable}_1$ and $\mathit{reachable}_2$) 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:

$$\begin{aligned} \mathit{reachable}_1 &= \{S_2\} \cup \mathit{reachable}_2 \\ \mathit{reachable}_2 &= \{S_3\} \cup \mathit{reachable}_3 \\ \mathit{reachable}_3 &= \{S_2\} \cup \mathit{reachable}_2 \end{aligned}$$

The least (and desired) solution is:

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

An additional (and uninteresting) solution also includes S_1:

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

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. (The class `ASTNode` is a superclass of all node classes.) 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;
}
```