jastadd

About
Download
Documentation
Concept overview
Tutorial
Reference manual
Examples
Concurrent Attributes
ExtendJ: The JastAdd Extensible Java Compiler
Tool support
Applications
Contact

Reference Manual for JastAdd 2.3.5

Click here to read the JastAdd 2.1.13 manual.

Index

Quick syntax overview

AST Specification Syntax

Syntax for AST class declarations in `.ast` files:

Syntax	Meaning
`A;`	AST class
`B: S;`	AST subclass (B is a subclass of S)
`abstract A;`	AST class, abstract
`B ::= Y;`	Child component `Y`.
`B ::= MyY:Y;`	Child component `MyY` of type `Y`.
`X ::= C*;`	List component `C`, containing `C` nodes.
`X ::= MyC:C*;`	List component `MyC`, containing `C` nodes.
`Y ::= [D];`	Optional component `D`.
`Y ::= [MyD:D];`	Optional component `MyD` of type `D`.
`Z ::= <E>;`	Token component `E` of type `String`.
`Z ::= <F:Integer>;`	Token component `F` of type `Integer`.
`U ::= /V/;`	NTA component `V`.
`U ::= /G:V/;`	NTA component `G` of type `V`.

Aspect declarations

Syntax for attribute declarations in `.jrag` and `.jadd` files:

Declaration	Meaning
`aspect N { decl* }`	Aspect declaration
`syn T A.c();`	Synthesized attribute
`syn T A.c() = exp;`	Synthesized attribute, default equation
`syn T A.c() { stmt* }`	Synthesized attribute, method body
`syn lazy T A.c() { stmt* }`	Synthesized attribute, cached
`syn X A.c() circular [bot] = exp;`	Synthesized attribute, circular
`syn nta X A.c() = exp;`	Nonterminal attribute (NTA)
`eq B.c() = exp;`	Synthesized equation
`eq B.c() { stmt* }`	Synthesized equation, method body
`inh X A.i();`	Inherited attribute
`inh lazy X A.i();`	Inherited attribute, cached
`eq B.getChild().i() = exp;`	Inherited equation, broadcast
`eq B.getChild().i() { stmt* }`	Inherited equation, broadcast, method body
`eq B.getA().i() = exp;`	Inherited equation for the `A` child of `B` nodes.
`eq B.getDecl(int index).i() = exp;`	Inherited equation for the `Decl` list component of `B` nodes.
`coll LinkedList<B> A.c() [new LinkedList<B>()] root A;`	Collection attribute
`coll LinkedList<B> A.c();`	Collection attribute, short form
`B contributes exp when cond to A.c() for targetexp;`	Collection contribution
`B contributes exp when cond to A.c();`	Collection root contribution
`refine FileName thing = exp;`	Refine an attribute equation (synthesized or inherited)
`refine FileName thing { stmt* }`	Refine an attribute or intertype method
`rewrite A { when condition to B { stmt* } }`	AST node rewrite
`uncache A.x();`	Never cache `A.x()`
`cache A.y(int a);`	Please cache `A.y(int)`
`public void A.m(int x) { stmt* }`	Intertype declared method
`public String A.f = exp;`	Intertype declared field

Abstract syntax

Abstract grammars are specified in `.ast` files and are used to generate a Java class hierarchy. The classes in an abstract grammar are referred to as AST classes. The AST classes are used by the parser to build Abstract Syntax Trees (ASTs).

Predefined AST classes

The AST classes include user declared classes in the abstract grammar, as well as a few predefined AST classes that are implicitly generated. The predefined AST classes are described in the table below.

Predefined AST class	Purpose	Accessing children
`ASTNode`	This is the base class which all other AST classes extend. Children are numbered from `0` to `getNumChild() - 1`.	Children are accessed using the generated methods `getNumChild()` and `getChild(int)`. `public ASTNode<T extends ASTNode> implements Cloneable {` `  int getNumChild();` `  ASTNode getChild(int index);` `  ASTNode getParent();` `  Iterable<T> astChildren();` `  Iterator<T> astChildIterator();` `}`
`List`	Contains elements of list components in AST classes.	`getNumChild()` and `getChild(int)` are inherited from `ASTNode`. The enhanced `for` statement can be used on the list since `List` implements `Iterable<ASTNode>`. `public List<T extends ASTNode> extends ASTNode<T> implements Iterable<T> {` `}`
`Opt`	Used to implement optional components in AST classes. Has 0 or 1 child.	`getNumChild()` and `getChild(int)` inherited from `ASTNode` can be used when accessing `Opt` nodes directly. `public Opt<T extends ASTNode> extends ASTNode<T> {` `}`

See also "About Lists and Opts".

Abstract syntax constructs

The table below documents the syntax used to declare user declared AST classes in `.ast` files.

Basic constructs

Construct	Meaning	Generated API
`abstract A;`	`A` is an abstract AST class. `A` corresponds to a nonterminal in the context free grammar.	`abstract class A extends ASTNode { }`
`B: A ::= ...;`	`B` is a concrete subclass of `A`. `B` corresponds to a production of `A` in the context-free grammar.	`class B extends A { }`
`C: A ::= A B C;`	C has three children of types A, B, and C. The API supports typed traversal of the children.	`class C extends A {` `  A getA();` `  B getB();` `  C getC();` `}`
`D: A;`	`D` has no children. `D` corresponds to an empty production of `A`.	`class D extends A { }`
`E1 ::= A;`	`E1` has a child of type `A`.	`class E1 {` `  A getA();` `}`
`E2 ::= [B];`	`E2` has an optional component of type `B`.	`class E2 {` `  boolean hasB();` `  B getB();` `}`
`E3 ::= C*;`	`E3` has a list component of zero or more `C` nodes.	`class E3 {` `  int getNumC();` `  C getC(int);` `  List<C> getCList();` `}`
`E4 ::= <D>;`	`E4` has a token component of type `D`. The set method is intended to be used only by the parser to set the token value. The token value should not be changed after tree construction, and the set method should not be used by an attribute equation as it has side effects.	`class E4 {` `  String getD();` `  void setD(String);` `}`
`A ::= /D/;`	`A` has a nonterminal attribute (NTA) component `D`. The `D` component is not created by the parser, it is instead computed on demand, using an attribute equation. See specifying NTAs for more info.	`class A {` `  D getD();` `}`

Naming children

Construct	Meaning	Generated API
`F ::= Foo:A Bar:B;`	It is possible to give components custom names. *Note!* If there is more than one child of the same type, they *must* be named.	`class F {` `  A getFoo();` `  B getBar();` `}`
`G ::= Thing:B*;`	List components can be named.	`class G {` `  int getNumThing();` `  B getThing(int);` `  List<B> getThingList();` `}`
`H ::= [Foo:X];`	Optional components can be named.	`class H {` `  boolean hasFoo();` `  X getFoo();` `}`

Typed tokens

Tokens are implictly `String` typed. But you can also give a token an explicit type:

Construct	Meaning	Generated API
`A ::= <T>;`	Here, `T` is a token of the type `String`.	`class A {` `  String getT();` `  void setT(String);` `}`
`A ::= <T:String>;`	This is equivalent to the example above.	
`A ::= <T:int>;`	Here, T is a token of the Java primitive type int.	`class A {` `  int getT();` `  void setT(int);` `}`
`A ::= <Ref:B>;`	Here, Ref is an intra-AST reference to a node of type B. This is a static reference to another node in the AST. The reference is not computed, rather it is set once during tree building.	`class A {` `  B getRef();` `  void setRef(B);` `}`

Inheriting children

AST class children are inherited by subtypes.

Construct	Meaning	Generated API
`abstract A ::= B C;` `D: A;` `E: A;`	`D` and `E` are subclasses of `A` and inherit the children of `A`.	`abstract class A extends ASTNode {` `  B getB();` `  C getC();` `}` `class D extends A { }` `class E extends A { }`
`A ::= B C;` `D: A ::= F;`	A subclass declaration can add children, but not remove children from the superclass. Here, `D` has the children `B`, `C`, `F`.	`class A extends ASTNode {` `  B getB();` `  C getC();` `}` `class D extends A {` `  F getF();` `}`
`A ::= B C;` `D: A ::= C F B;`	Subclasses can repeat superclass children to change the child order. Here, the order of children in `D` is: `C`, `B`, `F`. This affects the generated constructors of `D` and the children accessed by `getChild(int)`. See below for more info about generated constructors.	Same as above.

List and Opt components

JastAdd generates accessor methods to access optional and list components. The generated methods can be used instead of accessing the `Opt` and `List` container directly. The generated accessor methods are listed in the table below. The `Opt` and `List` nodes can be accessed by `getXOpt()` and `getXList()`, if needed.

Construct	Generated API	Example use
`A ::= [B];`	`class A {` `  boolean hasB();` `  B getB();` `  Opt<B> getBOpt();` `}`	`A a = ...;` `if (a.hasB()) {` `  B b = a.getB();` `  ...` `}`

`C ::= D*;`	`class C {` `  int getNumD();` `  D getD(int index);` `  List<D> getDList();` `}`	`C c = ...;` `for (D d : c.getDList()) {` `  ...` `}`

Building AST nodes

Use the following constructor API to build the AST. Typically you build the AST in the action routines of your parser. But you can of course also create an AST by coding it explicitly, e.g., in a test case. If you use JavaCC and JJTree, see below.

AST declaration	Generated constructor	Comment
`A ::= B C [D] E* <G>;`	`A(B, C, Opt<D>, List<E>, String)`	The constructor parameter order is same as the child order.
`A ::= A /B/ C;`	`A(A, C)`	Nonterminal attributes are not initialized via the constructor.

The predefined AST classes `Opt` and `List` have some default constructors to help with building trees:

Predefined AST class	Generated constructors	
`List<T>`	• `List()` create an empty list. • `List(T...)` this constructor accepts a variable number of AST nodes as arguments, and adds the arguments as children of the constructed list. • `List(Collection<T>)` adds all AST nodes in a collection to the list.	The constructor parameter order is same as the child order.
`Opt<T>`	• `Opt()` creates an empty optional. • `Opt(T)` creates an optional containing the given AST node.	Nonterminal attributes are not initialized via the constructor.

The `List` node constructor that takes no arguments can be used together with the `add` method which returns the list itself, so you can chain multiple list additions after creating a new node, like this:

```
A1 a = ...;
A2 a = ...;
List<A> list = new List<A>().add(a1).add(a2);
```

Below is an example of building an AST based on the grammar

```
A ::= B*;
B ::= C;
C ::= <ID>;
```

An example AST for this grammar can be built like this:

```
A = new A(new List<B>(new B(new C("foo")), new B(new C("bar"))));
```

Building ASTs using JJTree

If you use JJTree, the tree building code is generated by JJTree. You can use the "#X" notation in the JJTree specification to guide the node creation.

JJTree maintains a stack of created nodes. The "#X" notation means:

1. Create a new object of type X.
2. Pop the nodes that were created during this parse method and insert them as children to the new X node.
3. Push the new X node.

You need to explicitly create `List` and `Opt` nodes. When the parsing structure does not fit the abstract tree, e.g. when parsing expressions, you need to use some additional tricks. You also need to set token values explicitly.

Aspects

JastAdd aspects support *intertype declarations* for AST classes. An intertype declaration is a declaration that appears in an aspect file, but that actually belongs to an AST class. The JastAdd system reads the aspect files and weaves intertype declarations into the target AST classes.

The kinds of intertype declarations that can occur in an aspect include ordinary Java declarations like methods and fields, and attribute grammar declarations like attributes, equations, and rewrites.

An aspect file can contain import declarations and one or more aspects, e.g.:

```
import java.lang.util.*;
aspect A {
  abstract public void Stmt.m();
  public void WhileStmt.m() { ... }
  public void IfStmt.m() { ... }
  ...
}
aspect B {
  private boolean Stmt.count = 0;
}
```

The aspect syntax is similar to that of AspectJ, but in contrast JastAdd aspects are not real language constructs. The JastAdd system simply reads the aspect files and inserts the intertype declarations into the appropriate AST classes. For example, the method `m()` and its implementations are inserted into classes `Stmt`, `WhileStmt`, and `IfStmt`. And the declaration of the field `count` is inserted into the class `Stmt`. Import declarations are inserted into all AST classes for which there are

intertype declarations in the aspect. So, the import of `java.lang.util.*` is inserted into `Stmt.java`, `WhileStmt.java`, and `IfStmt.java`. For a more detailed discussion on the similarities and differences between JastAdd aspects and AspectJ, see below.

The aspect names, e.g., A and B above, do not show up in the woven Java code, other than in generated documentation comments for woven attributes and intertype declarations. Aspect names are used for refine declarations.

Aspect names are a way to indicate the purpose of the aspect. A common idiom for naming aspects is to have one aspect per aspect file, and give the aspect the same name as the filename sans the extension.

JADD and JRAG files

An aspect file can have the suffix `.jadd` or `.jrag`. The JastAdd system does not differ between these two types of files, but we recommend the following use:

- Use `.jrag` files for declarative aspects, i.e., where you add attributes, equations, and rewrites to the AST classes
- Use `.jadd` files for imperative aspects, i.e., where you add ordinary fields and methods to the AST classes

It is perfectly fine to not follow this convention, i.e., to mix both imperative and declarative features in the same aspect, but we try to follow the convention in our examples in order to enhance the readability of a system.

Example imperative aspect (JADD)

Here is an example *imperative* aspect that adds pretty printing behavior to some AST classes. Typically, this file would be named `PrettyPrint.jadd`:

```
aspect PrettyPrint {
  void WhileStmt.pp() {
    System.out.format("while %s do %s%n", getExp().pp(), getStmt().pp());
  }
  void IfStmt.pp() { ... }
  void Exp.pp() { ... }
}
```

Example declarative aspect (JRAG)

Here is an example *declarative* aspect that adds type checking to some AST classes. Typically, this file would be named `TypeCheck.jrag`:

```
import TypeSystem.Type;
aspect TypeCheck {
  syn Type Exp.actualType();
  eq LogicalExp.actualType() = Type.boolean();
  eq IdUse.actualType() = decl().getType();
  ...
  inh Type Exp.expectedType();
  eq WhileStmt.getExp().expectedType() = Type.boolean();
  syn boolean Exp.typeError() = !(actualType().equals(expectedType());
}
```

Supported AOP features

Feature	Comment
Intertype declaration of AST fields, methods, and constructors.	See the prettyprinting example above. The declarations are inserted into the corresponding AST classes by the AST weaver. Any modifiers (public, private, static, etc.), are interpreted in the context of the AST class. I.e., not as in AspectJ where the public/private modifiers relate to the aspect.
Intertype declaration of attributes, equations, and rewrites.	See the type checking example above. For more details, see Attributes. Note that access modifiers (public, private, etc.) are not supported for attributes. All declared attributes generate public accessor methods in the AST classes.
Declare additional interfaces for AST classes.	In an aspect you can write `WhileStmt implements LoopInterface;` This inserts an "implements LoopInterface" clause in the generated `WhileStmt` class.
Declare classes and interfaces in an aspect.	In an aspect you can write `interface I { ... }` `class C { ... }` This is equivalent to declaring the interface and class in separate ordinary Java files. The possibility to declare them inside an aspect is just for convenience.
Refine a method declared in another aspect.	For extensibility it is often useful to be able to replace or refine methods declared in another aspect. This can be done using a "refine" clause. In the following example, the aspect A declares a method `m()` in the class `C`. In the aspect `B`, the method is replaced, using a "refine" clause. This is similar to overriding a method in a subclass, but here the "overridden" method is in the same class, just defined in another aspect. Inside the body of the refined method, the original method can be called explicitly. This is similar to a call to super for overriding methods. `aspect A {` `  void C.m() { ... }` `}` `aspect B {` `  refine A void C.m() { // Similar to overriding.` `  ...` `  refined(); // Similar to call to super.`

	`` ... } } ``` Note that the refine clause explicitly states which aspect is refined (A in this case). Additional aspects may further refine the method. For example, an aspect C can refine the method refined in B. In most situations, the modifiers, type parameters, and return type of the original declaration should be re-stated in the refinement. The original method can be called using the keyword "refined". JastAdd replaces all occurrences of this keyword with the new name of the refined method. Be careful with how you use refined - even occurrences in string literals or comments are replaced!

Similarities and differences from AspectJ

The aspect concept in JastAdd was developed in parallel to the AspectJ development, and we have gradually adopted the AspectJ syntax, for features that are similar. The important similarity between JastAdd aspects and AspectJ aspects is the intertype declarations. In addition, JastAdd aspects support attribute grammar features which AspectJ does not. Note, however, that JastAdd supports intertype declarations only for the AST classes, not for classes in general as AspectJ does. There are many other features of AspectJ that are not supported in JastAdd, for example:

- Fields and methods private to an aspect are not supported.
- Declaration of additional parent classes is not supported.
- Dynamic features like AspectJ's pointcuts or advice are not supported.

Idiom for private fields and methods

As mentioned, JastAdd does not support fields and methods that are private to an aspect. As a workaround idiom, such fields and methods can be implemented as (non-private) static fields and methods in class `ASTNode`. As an example, consider the pretty printer. We might want to parameterize the pretty printer methods so that it can pretty print to any `PrintStream` object, not only on `System.out`. Here is how you could write this in AspectJ and the corresponding JastAdd implementation:

AspectJ code

```
aspect PrettyPrinter {
  private PrintStream ppStream = null;
  public void prettyprint(ASTNode n, PrintStream s) {
    ppStream = s;
    n.pp("");
    ppStream = null;
  }
  void ASTNode.pp(String indent) { }
  void WhileStmt.pp(String indent) {
    ...
    ppStream.println(...);
    ...
  }
  ...
}
```

JastAdd code

```
aspect PrettyPrinter {
  static PrintStream ASTNode.ppStream = null;
  public void ASTNode.prettyprint(PrintStream s)
    ppStream = s;
    pp("");
    ppStream = null;
  }
  void ASTNode.pp(String indent) { }
  void WhileStmt.pp(String indent) {
    ...
    ppStream.println(...);
    ...
  }
  ...
}
```

Attributes

Attributes are specified in JastAdd aspect files.

Different kinds of attributes are documented in the following sections.

Synthesized attributes

Syntax	Meaning
`syn T A.x();`	Declares a synthesized attribute `x` of type `T` in class A. There must be equations defining x in A (if A is concrete) or in all concrete subclasses of A (if A is abstract). *Note!* Synthesized attributes are conceptually equivalent to abstract virtual functions (without side-effects). The main difference is that their values may be cached (see below). They can be accessed in the same way as virtual functions. I.e., the declaration generates the following Java API: `T A.x();`
`eq A.x() = Java-expr;`	The equation defines the value of the synthesized attribute x of AST nodes of type A. The Java-expression that defines the value must be free from externally visible side-effects. The context of the expression is the class A, and any part of the class A's API may be used in the computation, including accesses to other attributes. *Note!* Equations defining synthesized attributes are conceptually equivalent to virtual method implementations (without side-effects).
`eq B.x() = Java-expr;`	Suppose B is a subclass of A. This equation overrides the corresponding (default) equation for A.x(). *Note!* This is equivalent to overriding method implementations.
`syn T A.x() = Java-expr;`	The declaration of a synthesized attribute and the (default) equation for it can be written in one clause. So the clause to the left is equivalent to: `syn T A.x();` `eq A.x() = Java-expression;`

Inherited attributes

Inherited attributes propagate information down in the AST. When an inherited attribute is evaluated, the evaluation code first searches upward in the AST for a node that can compute the inherited attribute. The equation may be on the parent of the current node, or an ancestor, even the root of the tree.

Syntax	Meaning
`inh T A.y();`	y is an inherited attribute in class A and of type T. There must be equations defining y in all classes that have children of type A. If a class has several children of type A, there must be one equation for each of them. Inherited attributes can be accessed in the same way as synthesized attributes. I.e., the declaration generates the following Java API: `T A.y();` *Note!* Inherited attributes differ from ordinary virtual functions in that their definitions (equations/method implementations) are located in the *parent* AST node, rather than in the node itself. *Note!* The concept of *inherited* attributes in this Attribute Grammar sense is completely different from object-oriented inheritance. Both attribute grammars and object-orientation were invented in the late 60's and the use of the same term "inheritance" is probably a mere coincidence: In AGs, inheritance takes place between nodes in a syntax tree. In OO, inheritance takes place between classes in a class hierarchy.
`eq C.getA().y() = Java-expr;`	This equation defines the value of the inherited attribute y() of the A child of a C node. *Note!* The Java-expression executes in the context of C. *Note!* The equation is similar to a method implementation. *Note!* The equation actually applies to all inherited attributes y in the subtree rooted at A, provided that they declare the y attribute. See below under broadcast attributes.
`eq D.getA().y() = Java-expr;`	Suppose D is a subclass of C. In this case, the equation overrides the previous one. *Note!* This is analogous to overriding a virtual method implementation.

Method syntax

It is possible to write the computation of an attribute value as a method body instead of as a single expression. This may be convenient when the computation is complex. Inside the method body it is possible to use ordinary imperative Java code with local variables, assignments, loops, etc. However, the net result of the computation must not have any side-effects. (Currently, JastAdd does not check the absence of such side-effects, but future versions might do so.)

Example of a method body in a synthesized attribute:

```
syn T A.x() {
  ...
  return Java-expr;
}
```

Cached attributes

An attribute can be declared *lazy* in order to speed up subsequent evaluations of the attribute. An attribute that is declared lazy will have its value is cached after the first access to it. The next time the attribute is accessed, the cached value is returned directly. We recommend that attributes that are expensive to compute and that are accessed multiple times should be declared lazy. For example, declaration bindings and type attributes are good candidates for caching. JastAdd has facilities for automatically computing good cache configurations based on profiling, but this is not yet documented here.

Example lazy attribute declaration:

```
syn lazy A.x();
```

Another way to change the caching behaviour of attributes is to use a separate cache declaration:

Syntax	Meaning
`uncache A.x();`	This prevents attribute x of class A from ever being cached, though attributes with the same name in subtypes of A can still be cached if declared lazy.
`cache A.x();`	This tells JastAdd to cache attribute x of class A.

Cache declarations take precedence over the `lazy` keyword, but conflicting cache declarations for a single attribute will cause JastAdd to report an error as there is no way to select the proper caching strategy.

Refining attributes

Equations defined in one aspect can be refined in another aspect, in the same way as methods can be refined, see JastAdd aspect files. In the example below, the equation replaces the corresponding equation declared in the aspect named `S`:

```
refine S eq B.x() = Java-expr;
```

The value of the original equation in `S` can be accessed by the expression `S.B.x()`. Older JastAdd code accessed the original equation by using the expression `refined()`.

Parameterized attributes

Attributes can have parameters. This is a bit unusual for attribute grammars, but a natural generalization when attributes are viewed as virtual functions.

Syntax	Meaning
`syn T A.x(int a);` `eq A.x(int a) {`	Here, x is a parameterized synthesized attribute. The equation is similar to a method implementation and the argument values can be used in the computation of the resulting value.

`return Java-expr;` `}`	
`inh T A.y(int a);` `eq C.getA().y(int a) {` `return Java-expr;` `}`	Here, y is a parameterized inherited attribute. The equation executes in the context of C and can in addition access the arguments (a in this case).

Broadcasting inherited attributes

Often, an inherited attribute is used in a number of places in a subtree. If basic inherited attributes are used, the value needs to be copied explicitly using inherited attributes in all the intermediate nodes. For convenience, JastAdd supports another technique called broadcasting, where an inherited attribute is available in every node of a complete subtree. An equation defining an inherited attribute actually broadcasts the value to the complete subtree of the child. By using this technique, no explicit copy attributes are needed.

Syntax	Meaning
`eq C.getA().y() = ...;` `inh T A.y();`	Here, the equation defines an inherited attribute y() declared in the A child of a C node. This equation actually applies not only to the inherited y() attribute of the A child, but to all inherited y() attributes in the whole subtree of A. In order to for a node N in the subtree to access y(), the attribute must, however, be exposed by declaring y() as an inherited attribute of N.
`inh T B.y();`	Here, the attribute y() is exposed in B by declaring it as an inherited attribute there. If there is a B node that is in the subtree rooted at the A that is a child of a C node, then the equation above will apply.

Overruling broadcast definitions

A broadcast definition of an attribute a() applies to all nodes in a subtree rooted by N. If, however, there is a node in the subtree which has another equation that defines a() for a child M, that equation will take precedence for defining a() in M and its subtree.

Differentiating between children in a list

When defining an inherited attribute of a child node that is an element of a list, it is sometimes useful to know which index the child node has. This can be done as follows:

```
C ::= E*;
eq C.getE(int index).y() = expr;
```

Here, a `C` node has a list of `E` children. When defining the `y()` attribute of a given (subtree of an) `E` child, the value might depend on the index of the child. For example, if the `E` nodes are actual arguments of a procedure, we might want to pass down the expected type of each argument.

The example equation shows how to declare the index as a parameter of the `getE()` method, and to access the index in the equation body.

Circular attributes

Attributes can be circularly defined, meaning that the value of the attribute can depend (indirectly) on itself. Circular attributes are evaluated iteratively, starting with a start value given in the declaration of the attribute. The evaluation stops when the value equals that for the previous iteration.

Circular attributes are always cached. They do not need to be declared "lazy".

It is an error if a lazy attribute is circular, but not declared as such. With the `visitCheck` and `componentCheck` options this can be detected at runtime, and an exception will be generated. To be sure that the evaluation of circular attributes will converge, the values should be arranged into lattices of finite height, the bottom values should be used as starting values, and each equation on the cycle should be monotonic with respect to the lattices.

```
syn T A.x(int a) circular [bv];
eq A.x(int a) = rv;
```

Here, the attribute x is a circular attribute. The starting value is *bv* (a Java expression).

The equation defines x as having the value computed by the Java expression `rv`. Note that `rv` may depend (directly or indirectly) on x.

If an attribute `A.x()` that was not declared circular becomes part of a circular evaluation, for example by adding an extension aspect, then it is essential that the original attribute is never cached. This can ensured using the `uncache` declaration described above:

```
uncache A.x();
```

Attribute systems with circular attributes are well defined if at least one attribute on every possible circular dependency cycle is declared circular and the other attributes on all cycles are either also declared circular or declared uncached as above.

The --safeLazy Option

Although it is normally an error to have cached non-circular attributes in a circular evaluation, the `safeLazy` option can be used to make non-circular attributes aware of circular evaluations and safely cache their results during circular evaluation. This still requires that at least one attribute on every circular dependency cycle is declared circular.

The `safeLazy` option adds an extra cache field for each cached non-circular attribute which tracks the cycle ID on which the attribute was last evaluated. When the attribute is later re-evaluated it can reuse the cached value if the cycle ID is identical, or if it was previously cached outside of any circular evaluation.

Nonterminal attributes

Nonterminal attributes (NTAs) are nodes in the AST. Whereas normal AST nodes are built by the parser, the NTAs are viewed as attributes and are defined by equations.

- NTAs can be inherited or synthesized.
- The value in the equation should be a freshly built AST subtree. It should be complete in the sense that all its children should also be freshly created nodes (i.e., they are not allowed to be initialized to null).
- The NTA can itself have attributes that can be accessed like normal attributes.
- If the NTA has inherited attributes, there must be equations for those attributes in some ancestor, as for normal children.

Declaration Syntax

```
syn nta C A.anNTA() = new C();
```

Older syntax

In the older syntax, a nonterminal attribute is added as follows:

- Declare the NTA in the ast file, see also NTAs in the abstract syntax.
- Declare the NTA as an attribute in a jrag file. It can be declared as a synthesized or an inherited attribute. The name of the attribute should be the same as in the AST traversal API, e.g., *getX* if the NTA is called *X*.
- Add equations defining the NTA. The defining value should be a new AST of the appropriate type, created using the AST building API.

Note that if the NTA is a list or an optional node, you need to create the appropriate AST with a `List` or an `Opt` node as its root. See examples below.

Simple synthesized NTA

In an .ast file:

```
A ::= B /C/;
```

In a .jrag file:

```
syn C A.getC() = new C();
```

The NTA *C* is declared in the .ast file. It is then declared as a synthesized attribute *getC()* in the .jrag file. The equation is provided directly in the declaration and creates a new *C* node.

List NTA

In an .ast file:

```
A ::= B /C*/;
```

In a .jrag file:

```
syn C A.getCList() =
    new List()
        .add(new C())
        .add(new C());
```

The list NTA *C* is declared in the .ast file. It is then declared as a synthesized attribute *getCList()* (the same name as in the implementation level traversal API) in the `.jrag` file. The equation is provided directly in the declaration and creates a *List* node to which is added a number of *C* nodes (two in this example).

Collection attributes

Collection attributes have composite values defined by so called *contributions*. Contributions are similar to sythesized attributes which add their value to the collection attribute value. Contributions may be located in any node in the subtree of the collection root node.

Collection attributes are evaluated in two phases: first a survey phase which searches for nodes that can contribute to the collection value and then a collection phase in which all contributions are computed.

In the survey phase, the attribute evaluator first finds the collection root node and then searches the subtree of the root node to find all contributing nodes. If no root node is declared, the grammar root node is used (if one exists).

Examples

Collection attributes are commonly used to collect error messages. For example:

```
coll ArrayList<String> Program.errors();
Div contributes
    "Division by zero is not allowed!"
    when getRight().isZero()
    to Program.errors();
```

Declaration syntax

The syntax for declaring a collection attribute looks like this:

```
coll T A.c() [fresh] with m root R;
```

where

- **T** is the type of the attribute. Usually `T` is a subtype of `java.lang.Collection`.
- **A** is AST class on which the attribute is evaluated.
- The **.c()** part declares the attribute name, in this case `c`.
- (optional) **[fresh]** tells JastAdd how the intermediate collection result is initialized. The Java expression **fresh** creates an empty instance of the result type. This part is optional if `T` is a concrete type with a default constructor, if it is omitted the default constructor of the type `T` is used, i.e. `new T()`.
- (optional) **with m** specifies the name of a method to be used for updating the intermediate collection object. This part is optional and the default method `add` is used if no update method is specified. The update method must fulfill these requirements:
  - The method `m`, should be a one-argument method of `T`.

    - The method `m` should mutate the `T` object by adding one object to it.
    - The method `m` should be commutative, in the sense that the order of calling `m` for different contributions should yield the same resulting `T` value.
- (optional) The **root R** part declares the collection root type. The collection mechanism starts by finding the nearest ancestor node of type `R` for the `A` node which the collection attribute is evaluated on. The subtree rooted at that nearest `R` ancestor is searched for contributions to `A.c()`, this means that the collection is scoped to the subtree of `R`, and contributions outside that tree are not visible.

Contribution declaration

When JastAdd evaluates a collection attribute it first performs a "survey" of the AST, searching for contributions to the given collection attribute. Contributions are added by using contribution statements like below:

```
N1 contributes value-exp
    when cond-exp
    to N2.a()
    for N2-ref-exp;
```

Let's look at each part of the above template statement:

- **N1** is the type of AST nodes that provide this particular contribution to the target collection attribute.
- **value-exp** is a Java expression that evaluates to an object to be added to the intermediate collection of the target collection attribute.
- (optional) **when cond-exp** is an optional contribution condition: the contribution is only added to the target collection attribute if the Java expression `cond-exp` evaluates to `true`.
- **N2** is the node type where the target collection attribute is declared.
- **.a()** is the name of the target collection attribute.
- (optional) **for N2-ref-exp** gives a Java expression, `N2-ref-exp`, which evaluates to a reference to the AST node that owns the collection attribute this contribution is contributing to. This is the target expression, and it can be omitted if the target node is identical to the collection root node.

One can optionally contribute one contribution to multiple target nodes by using this syntax:

```
N1 contributes value-exp
    when cond-exp
    to N2.a()
    for each N2-ref-set;
```

where `N2-ref-set` is a Java expression that evaluates to an `Iterable<N2>` containing references for the set of contribution target nodes.

It is possible to contribute multiple values in a single contribution by using this syntax:

```
N1 contributes each value-exp
    when cond-exp
    to N2.a()
    for each N2-ref-set;
```

Note the **each** before `value-exp`. This syntax works if `value-exp` has the type `Iterable<E>` where `E` is the element type of the collection attribute. For example, if the collection attribute is declared as `coll LinkedList<String> ...` then `value-exp` should have the type `Iterable<String>`.

NTA Contributions

It is possible to add contributions from an NTA child to a collection attribute using the following variation of the `contributes` statement:

```
N1 contributes nta getMyNta() to N2.a();
```

The above statement means that the NTA child named `MyNta` in node type `N1` is also searched for contributions during the survey phase of the evaluation of the collection attribute `N2.a()`.

Custom Collection Survey

It is possible to customize the tree traversal used to search for contributions for a collection attribute. This can be done using an alternative form of the `contributes` statement, where the expression part is replaced by a code block:

```
N1 contributes {
  getA().collectContributions();
  super.collectContributions();
} to N2.a();
```

The meaning of the above code is that the `N1` node type should search its `A` child while searching contributions for the `N2.a()` collection attribute. The call to `super.collectContributions()` is needed to ensure that all regular children of `N1` are also searched for contributions.

Multiple custom collection survey blocks like this can be used, but only one of them needs to call `super.collectContributions()`. It is possible to use attributes inside the code blocks to decide when a particular subtree should be searched for contributions.

Rewrites

JastAdd has a mechanism for replacing AST nodes by a rewritten version of the node whenever the node is first accessed. Rewrites are declared using rewrite rules, described below.

Unconditional rewrite rule

```
rewrite A {
  to B {
    ...
    return exp;
  }
}
```

An A node will be replaced by the node specified in the Java expression *exp*. This will happen as soon as the A node is accessed (by a get() method from its parent), so if you traverse the tree you will only be able to access the final rewritten nodes.

A and B must be AST classes.

The *exp* must be of type B.

Let the set S be the superclasses of A (including A) that occur on right-hand sides of productions in the abstract syntax. B must be a subclass of all classes in S. This guarantees that replacing an A node by a B node does not break the rules in the abstract syntax.

The code in the body of the rewrite may access and rearrange the nodes in the subtree rooted at A, but not any other nodes in the AST. Furthermore, the code may not have any other side effects.

Conditional rewrite rule

```
rewrite A {
  when ( condition )
    to B {
      ...
      return exp;
    }
}
```

The conditional rewrite works in the same way as the unconditional one, but performs the replacement only if the boolean expression *condition* is true. The condition may access anything in the AST, e.g., attributes, other tree nodes and their attributes, etc.

Iterative rewriting

After a node has been replaced according to a rewrite rule, all conditional rewrite rules are checked again, and a new rewrite may be performed. This is iterated until no rule conditions hold.

Order of rewriting

At each iteration of rewriting, the rule conditions are evaluated in a certain order. The first condition that is true is used for rewriting in that iteration. The order in which rule condition evaluation occurs is the following:

- conditions in superclasses are evaluated before conditions in subclasses
- conditions within an aspect file are evaluated in lexical order
- conditions in different aspect files are evaluated in the order the files are listed in the jastadd command.

Confluency

If the order of rewriting of a node does not effect the final result, the rules are said to be *confluent*. This is highly desirable, since it makes the specification more readable to not have to take lexical order of rules into account. However, JastAdd cannot check that the rules are confluent. In cases where several conditions for a node are true at the same time, we recommend that you contemplate the rules and try to find out if they could be non-confluent. In that case, we recommend you to refine the conditions so that only one can apply at a time. This makes your specification independent of lexical order. Note that it is often useful to have several different rules that apply at the same time for a given node, but which are confluent.

Shorthand notation

If you have several conditional rewrite rules, you may write them inside the same rewrite block. So, e.g., writing

```
rewrite A {
  when ( condition-1 )
  to B {
    ...
    return exp-1
  }
  when ( condition-2 )
  to C {
    ...
    return exp-2
  }
}
```

... is equivalent to:

```
rewrite A {
   when condition-1
   to B {
      ...
      return exp-1
   }
}
rewrite A {
  when condition-2
  to C {
    ...
    return exp-2
  }
}
```

Sometimes you don't need a block for computing the resulting node. It may be sufficient with an expression. In that case, you may simply write the expression instead of the block, e.g., as follows:

```
rewrite A {
   when ( condition-1 )
   to B exp-1
   when ( condition-2 )
   to C exp-2
}
```

... which is equivalent to:

```
rewrite A {
   when ( condition-1 )
   to B { return exp-1 }
   when ( condition-2 )
   to C { return exp-2 }
}
```

Building with JastAddGradle

One of the simplest ways to build a JastAdd project is by using the Gradle build system and the JastAddGradle plugin.

To use JastAddGradle, add the following to the plugin block in `build.gradle`:

```
plugins {
  id "org.jastadd" version "1.13.3"
}
```

To use JastAddGradle you can, for example, define a new task of type `JastAddTask`:

```
task generateAst(type: org.jastadd.JastAddTask) {
  outputDir = file("src/gen")
  sources = fileTree("lang")
  options = [ "--package=lang" ] // Options (see below).
  doFirst { file('src/gen').mkdirs() }
}
```

Remember to also include the generated sources in the Java source set:

```
sourceSets { main { java { srcDir "src/gen" } } }
```

More documentation and several examples of how to use JastAddGradle are available in the JastAddGradle GitHub repository.

Running JastAdd from the command line

JastAdd may be run from the command line using the following syntax:

```
java -jar jastadd2.jar [options] <source files>
```

Source file arguments are filepaths ending in `.ast`, `.jrag` and `.jadd`. At least one `.ast` file must be provided, otherwise JastAdd will not generate any code. Some of the available options are listed below.

Options

JastAdd has a large number of options that control code generation for attributes and also enable/disable certain kinds of attributes.

Here is a summary of available options:

Option	Purpose
`--help`	Print help text.
`--version`	Print version information.
`--package=NAME`	Optional package for generated files, default is none.
`--o=PATH`	Optional base output directory, default is current directory.
`--ASTNode=NAME`	Change name of ASTNode class to NAME.
`--List=NAME`	Change name of List class to NAME.
`--Opt=NAME`	Change name of Opt class to NAME.
`--stateClassName=NAME`	Change name of ASTState class to NAME.
`--generateImplicits=yes/no`	Enables code generation for all implicit types (ASTNode, Opt, List). Default is yes.
`--generateAnnotations=yes/no`	Enables code generation for meta annotations for attributes and classes.
`--ASTNodeSuper=NAME`	Sets supertype for ASTNode class.
`--beaver`	For compatibility with Beaver parsers. Use `beaver.Symbol` as ASTNode supertype and add setters for tokens accepting beaver symbols.
`--jjtree`	JJTree compatibility mode.
`--grammar=NAME`	In JJTree mode, generates accept methods for `NAMEVisitor`.
`--rewrite=regular`	Enable ReRAGs support.
`--visitCheck=false`	Disable circularity check for attributes.
`--cacheCycle=false`	Disable cache cyle optimization for circular attributes.
`--safeLazy`	Makes non-circular memoized attributes safe to use in circular attributes. (recommended)
`--defaultMap=EXPR`	Replaces the default map consturction expression for memoization with EXPR.
`--defaultSet=EXPR`	Replaces the default set construction expression for memoization with EXPR.
`--rewrite=none/cnta/regular`	Rewrite implementation: `cnta` is highly recommended for new projects.
`--lineColumnNumbers=yes/no`	Generate code for handling line/column numbers with Beaver parsers.
`--cacheCycle=yes/no`	Performance tuning option for circular attributes. (untested)
`--componentCheck=yes/no`	Check that circular attributes do not depend on memoized non-circular attributes. (obsolete)
`--indent=ARG`	Choose indentation in generated code, can be one of: `tab`, `2space`, `4space`, `8space`.
`--license=TEXT`	TEXT is inserted at the top of each generated Java file.
`--concurrent`	Enables concurrent attribute evaluation. See Concurrent Attributes.
`--numThreads=NUM`	Number of concurrent workers for parallelized collection attributes.
`--concurrentMap=NAME`	Name of concurrent map class used in concurrent code generation.
`--tracing=ARGS`	See Attribute Tracing.
`--cache=all|none`	Choose if all attributes are memoized.
`--incremental=ARGS`	Experimental incremental attribute evaluation option.
`--flush=ARGS`	Generate methods for flushing attribute caches.
`--dot`	Generate a DOT graph of the abstract grammar.
`--emptyContainerSingletons`	Use singleton objects for empty List or Opt nodes.
`--optimize-imports`	Remove unused imports in generated code.
`--statistics=PATH`	Output attribute statistics in CSV format to a file.
`--minListSize=NUM`	Performance tuning: minimum non-empty child list size.
`--staticState=yes/no`	Performance tuning: use a static field to store reference to AST state object.
`--inhEqCheck=yes/no`	(deprecated) Check that equations exist for inherited attributes.
`--traceVisitCheck=yes/no`	(deprecated) Print a message instead of throwing an exception on circularity error.
`--private=yes/no`	(deprecated) Use private modifier for generated methods.
`--lazyMaps=yes/no`	(deprecated) Choose if memoization maps are lazily initialized when evaluating attributes.
`--refineLegacy`	(deprecated) Enables old syntax for calling refined method.