JastAdd—an aspect-oriented compiler construction system

Görel Hedin * Eva Magnusson

Department of Computer Science, Lund University, Sweden

Abstract

We describe JastAdd, a Java-based system for compiler construction. JastAdd is centered around an object-oriented representation of the abstract syntax tree where reference variables can be used to link together different parts of the tree. JastAdd supports the combination of declarative techniques (using Reference Attributed Grammars) and imperative techniques (using ordinary Java code) in implementing the compiler. The behavior can be modularized into different aspects, e.g. name analysis, type checking, code generation, etc., that are woven together into classes using aspect-oriented programming techniques, providing a safer and more powerful alternative to the Visitor pattern. The JastAdd system is independent of the underlying parsing technology and supports any non-circular dependencies between computations, thereby allowing general multi-pass compilation. The attribute evaluator (optimal recursive evaluation) is implemented very conveniently using Java classes, interfaces, and virtual methods.

Key words: reference attributed grammars, aspect-oriented programming, compiler construction, visitor pattern, Java

1 Introduction

Many existing parser generators have only rudimentary support for further compilation. Often, the support is limited to simple semantic actions and tree building during parsing. Systems supporting more advanced processing are usually based on dedicated formalisms like attribute grammars and algebraic specifications. These systems often have their own specification language and

* Corresponding author
Email addresses: gorel.hedin@cs.lth.se (Görel Hedin), eva.magnusson@cs.lth.se (Eva Magnusson).

can be difficult to integrate with handwritten code, in particular when it is desired to take full advantage of state-of-the-art object-oriented languages like Java. In this paper we describe JastAdd, a simple yet flexible system which allows compiler behavior to be implemented conveniently based on an object-oriented abstract syntax tree. The behavior can be modularized into different aspects, e.g., name analysis, type checking, code generation, etc., that are combined into the classes of the abstract syntax tree. This technique is similar to the *introduction* feature of aspect-oriented programming in AspectJ [15]. A common alternative modularization technique is to use the Visitor design pattern [6], [24]. However, the aspect-oriented technique has many advantages over the Visitor pattern, including full type checking of method parameters and return values, and the ability to associate not only methods but also fields to classes.

When implementing a compiler, it is often desirable to use a combination of declarative and imperative code, allowing results computed by declarative modules to be accessed by imperative modules and vice versa. For example, an imperative module implementing a print-out of compile-time errors can access the error attributes computed by a declarative module. In JastAdd, imperative code is written in aspect-oriented Java code modules. For declarative code, JastAdd supports Reference Attributed Grammars (RAGs) [9]. This is an extension to attribute grammars that allows attributes to be references to abstract syntax tree nodes, and attributes can be accessed remotely via such references. RAGs allow name analysis to be specified in a simple way also for languages with complex scope mechanisms like inheritance in object-oriented languages. The formalism makes it possible to use the Abstract Syntax Tree (AST) itself as a symbol table, and to establish direct connections between identifier use sites and declaration sites by means of reference attributes. Further behavior, whether declarative or imperative, can be specified easily by making use of such connections. The RAG modules are specified in an extension to Java and are translated to ordinary Java code by the system.

Our current version of the JastAdd system is built on top of the LL parser generator JavaCC [11]. However, its design is not specifically tied to JavaCC: the parser generator is used only to parse the program and to build the abstract syntax tree. The definition of the abstract syntax tree and the behavior modules are completely independent of JavaCC and the system could as well have been based on any other parser generator for Java such as the LALR-based system CUP [4] or the LL-based system ANTLR [1].

The JavaCC system includes tree building support by means of a preprocessor called JJTree. JJTree allows easy specification of what AST nodes to generate during parsing, and also supports automatic generation of AST classes. However, there is no mechanism in JJTree to update AST classes once they have been generated, so if the AST classes need more functionality than is gener-

ated, it is up to the programmer to modify the generated classes by hand and to update the classes after changes in the grammar. In JastAdd, this tedious and error-prone procedure is completely avoided by allowing handwritten and generated code to be kept in separate modules. JastAdd uses the JJTree facility for annotating the parser specification with tree-building actions, but the AST classes are generated directly by JastAdd, rather than relying on the JJTree facility for this. SableCC [5] and JTB [12] are other Java-based systems that have a similar distinction between generated and handwritten modules. While both SableCC and JTB support the Visitor pattern for adding behavior, neither one supports aspect-oriented programming nor declarative specification of behavior like attribute grammars.

The attribute evaluator used in JastAdd is an optimal recursive evaluator that can handle arbitrary acyclic attribute dependencies. If the dependencies contain cycles, these are detected at attribute evaluation time. The evaluation technique is in principle the same as the one used by many earlier systems such as Madsen [20], Jalili [10], and Jourdan [13]: an access to an attribute value is replaced by a function call which computes the appropriate semantic function for the value and then caches the computed value for future accesses to the same attribute. A cache flag is used to keep track of whether the value has been computed before and is cached. A cycle flag is used to keep track of attributes involved in an evaluation so that cyclic dependencies can be detected at evaluation time. While these earlier systems used this evaluation algorithm for traditional attribute grammars, it turns out that this algorithm is also applicable to reference attribute grammars [9]. Our implementation in JastAdd differs from earlier implementations in its use of object-oriented programming for convenient coding of the algorithm.

The rest of the paper is outlined as follows. Section 2 describes the object-oriented ASTs used in JastAdd. Section 3 describes how imperative code can be modularized according to different aspects of compilation and woven together into complete classes. Section 4 describes how RAGs can be used in JastAdd and Section 5 how they are translated to Java. Section 6 discusses related work and Section 7 concludes the paper.

2 Object-oriented abstract syntax trees

2.1 Connection between abstract and parsing grammars

The basis for specification in JastAdd is an abstract context-free grammar. An abstract grammar describes the programs of a language as typed trees rather than as strings. Usually, an abstract grammar is essentially a simplification

of a parsing grammar, leaving out the extra nonterminals and productions that resolve parsing ambiguities (e.g., terms and factors) and leaving out tokens that do not carry semantic values. In addition, it is often useful to have fairly different structure in the abstract and parsing grammars for certain language constructs. For example, expressions can be conveniently expressed using EBNF rules in the parser, but are more adequately described as binary trees in the abstract grammar. Also, parsing-specific grammar transformations like left factorization and elimination of left recursion for LL parsers are undesirable in the abstract grammar.

Most parsing systems that support ASTs make use of various automatic rules and annotations in order to support abstraction of the parsing grammar. In JastAdd, the abstract grammar is independent of the underlying parsing system. The parser is simply a front end whose responsibility it is to produce abstract syntax trees that follow the abstract grammar specification.

2.2 Object-oriented abstract grammar

When using an object-oriented language like Java, the most natural way of representing an AST is to model the language constructs as a class hierarchy with general abstract classes like Statement and Expression, and specialized concrete classes like Assignment and AddExpression. Methods and fields can then be attached to the classes in order to implement compilation or interpretation. This design pattern is obvious to any experienced programmer, and documented as the *Interpreter* pattern in [6].

Essentially, this object-oriented implementation of ASTs can be achieved by viewing nonterminals as abstract superclasses and productions as concrete subclasses. However, this two-level hierarchy is usually insufficient from the modelling point of view where it is desirable to make use of more levels in the class hierarchy. For this reason, JastAdd makes use of an explicit object-oriented notation for the abstract grammar, similar to [8], rather than the usual nonterminal/production-based notation. This allows nonterminals with a single production to be modelled by a single class. It also allows additional superclasses to be added that would have no representation in a normal nonterminal/production grammar, but are useful for factoring out common behavior or common subcomponents. Such additional superclasses would be unnatural to derive from a parsing grammar, which is yet another reason for supplying a separate specification of the abstract grammar.

The abstract grammar is a class hierarchy augmented with subcomponent information corresponding to production right-hand sides. For example, a class Assignment typically has two subcomponents: an Identifier and an Expression.

Tiny.ast

```
1  Program ::= Block;
2  Block ::= Decl Stmt;
3  abstract Stmt;
4  BlockStmt : Stmt ::= Block;
5  IfStmt : Stmt ::= Exp Stmt OptStmt;
6  OptStmt ::= [Stmt];
7  AssignStmt : Stmt ::= IdUse Exp;
8  CompoundStmt : Stmt ::= Stmt*;
9  abstract Decl;
10 BoolDecl: Decl ::= <ID>;
11 IntDecl : Decl ::= <ID>;
12 abstract Exp;
13 IdUse : Exp ::= <ID>;
14 Add : Exp ::= Exp Exp;
   ...
```

Fig. 1. Abstract grammar for Tiny

Depending on what kind of subcomponents a class has, it is categorized as one of the following typical kinds (similar to many other systems):

list The class has a list of components of the same type.
optional The class has a single component which is optional.
token The class has a semantic value extracted from a token.
aggregate The class has a set of components which can be of different types.

The subcomponent information is used for generating suitable access methods that allow type safe access to methods and fields of subcomponents.

2.3 An example: Tiny

We will use a small toy block-structured language, *Tiny*, as a running example throughout this paper. Blocks in Tiny consist of a single variable declaration and a single statement. A statement can be a compound statement, an if statement, an assignment statement, or a new block.

Figure 1 shows the object-oriented abstract grammar for Tiny. (The line numbers are not part of the actual specification.) All the different kinds of classes are exemplified: An aggregate class `IfStmt` (line 5), a list class `CompoundStmt` (line 8), an optional class `OptStmt` (line 6), and a token class `BoolDecl` (line 10). The classes are ordered in a single-inheritance class hierarchy. For example, `BlockStmt`, `IfStmt`, `AssignStmt`, and `CompoundStmt` (lines 4, 5, 7, and 8) are all subclasses to the abstract superclass `Stmt` (line 3).

From this abstract grammar, the JastAdd system generates a set of Java classes with access methods to their subcomponents. Figure 2 shows some of the generated classes to exemplify the different kinds of access interfaces to different kinds of classes. Note that for an aggregate class with more than one

```
abstract class ASTStmt {
}
class ASTIfStmt extends ASTStmt {
   ASTExp getExp() { ... }
   ASTStmt getStmt() { ... }
   ASTOptStmt getOptStmt() { ... }
}
class ASTOptStmt {
   boolean hasStmt() { ... }
   ASTStmt getStmt() { ... }
}
class ASTCompoundStmt extends ASTStmt {
   int getNumStmt() { ... }
   ASTStmt getStmt(int k) { ... }
}
class ASTBoolDecl extends ASTDecl {
   String getID() { ... }
}
class ASTAdd extends ASTExp {
   ASTExp getExp1() { ... }
   ASTExp getExp2() { ... }
}
```

Fig. 2. Access interface for some of the generated AST classes

subcomponent of the same type, the components are automatically numbered, as for the class `ASTAdd`.

Behavior can be added to the generated classes in separate aspect-oriented modules. Imperative behavior is added in `Jadd` modules that contain methods and fields as described in Section 3. Declarative behavior is added in `Jrag` modules that contain equations and attributes as described in Section 4. Figure 3 shows the Jastadd system architecture. The `jadd` tool generates AST classes from the abstract grammar and weaves in the imperative behavior defined in `Jadd` modules. The `jrag` tool translates the declarative `Jrag` modules into an imperative `Jadd` module, forming one of the inputs to the `jadd` tool. This translation is described in more detail in Section 5.

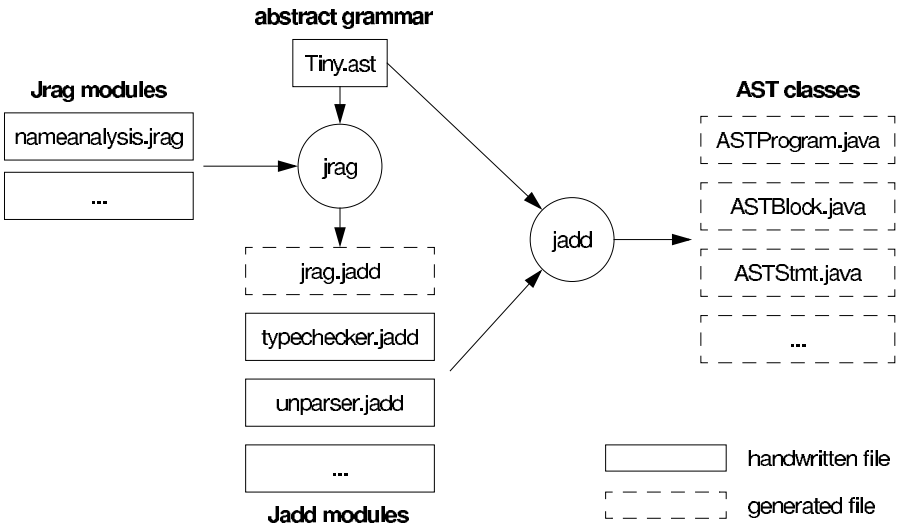

Fig. 3. Architecture of the JastAdd system

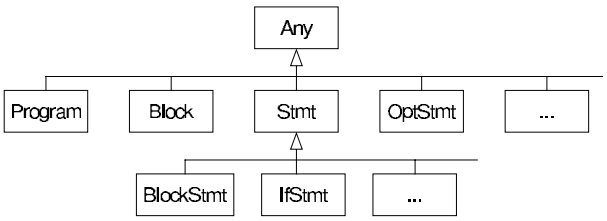

Fig. 4. Class Diagram after adding the superclass `Any`

2.4 Superclasses and interfaces

When adding behavior it is often found that certain behavior is relevant for several classes although the classes are unrelated from a parsing point of view. For example, both `Stmt` and `Exp` nodes may have use for an `env` attribute that models the environment of visible identifiers. In Java, such sharing of behavior can be supported either by letting the involved classes inherit from a common superclass or by letting them implement a common interface. JastAdd supports both ways. Common superclasses are specified in the abstract grammar. Typically, it is useful to introduce a superclass `Any` that is the superclass of all other AST classes. For the example in Figure 1, this would be done by adding a new class `"abstract Any;"` into the abstract grammar and adding it as a superclass to all other classes that do not already have a superclass. Figure 4 shows the corresponding class diagram.

Such common superclasses allows common default behavior to be specified and to be overridden in suitable subclasses. For example, default behavior for all nodes might be to declare an attribute `env` and to by default copy the `env` value from each node to its components by adding an equation to `Any`. AST classes that introduce new scopes, e.g. `Block`, can then override this behavior by supplying a different equation.

Java interfaces are more restricted in that they can include only method interfaces and no fields or default implementations. On the other hand, they are also more flexible, allowing, e.g., selected AST classes to share a specific interface orthogonally to the class hierarchy. Such selected interface implementation is specified as desired in the behavior modules and will be discussed in Section 3.4.

2.5 Connection to the parser generator

Building the tree

JastAdd relies on an underlying parsing system for parsing and tree-building. The abstract grammar is not tied to any specific parsing grammar or parsing

algorithm and there is thus normally a gap between these grammars that must be bridged. To aid the compiler writer, the JastAdd system generates a method `syntaxCheck()` which can be called to check that the built tree actually follows the abstract grammar.

Currently, JastAdd uses JavaCC/JJTree as its underlying parsing and tree-building system. JJTree allows easy specification of what AST nodes to generate during parsing. A stack is used to give the programmer control over the order in which to insert the individual nodes, so that the structure of the constructed AST does not have to match the structure of the parse. For example, expressions that are parsed as a list can easily be built as a binary AST. In this way, JJTree allows the gap between the parsing and abstract grammars to be bridged fairly easily.

Token semantic values

When building the AST, information about the semantic values of tokens needs to be included. To support this, JastAdd generates a set-method as well as a get-method for each token class. For example, for the token class `BoolDecl` in Figure 1, a method `void setID(String s)` is generated. This method can be called as an action during parsing in order to transmit the semantic value to the AST.

3 Adding imperative behavior

Object-oriented languages lend themselves very nicely to the implementation of compilers. It is natural to model an abstract syntax tree using a class hierarchy where nonterminals are modelled as abstract superclasses and productions as specialized concrete subclasses, as discussed in Section 2. Behavior can be implemented easily by introducing abstract methods on nonterminal classes and implementing them in subclasses. However, a problem is that to make use of the object-oriented mechanisms, the class hierarchy imposes a modularization based on language constructs whereas the compiler writer also wants to modularize based on aspects in the compiler, such as name analysis, type checking, error reporting, code generation, and so on. Each AST class needs to include the code related to all of the aspects and in traditional object-oriented languages it is not possible to provide a separate module for each of the aspects. This is a classical problem that has been discussed since the origins of object-oriented programming.

The Visitor design pattern is one (partial) solution to this problem [6]. It allows a given method that is common to all AST nodes to be factored out into a helper class called a Visitor containing an abstract `visit(C)` method for each AST class `C`. To support this programming technique, all AST classes are equipped with a generic method `accept(Visitor)` which delegates to the appropriate `visit(C)` method in the Visitor object. For example, a `Visitor` subclass `TypeCheckingVisitor` can implement type checking in its `visit` methods. Type checking of a program is started by calling `accept` on the root node with the `TypeCheckingVisitor` as a parameter.

There are several limitations to the Visitor pattern, however. One is that only methods can be factored out; fields must still be declared directly in the classes, or be handled by a separate mechanism. For example, in type checking it is useful to associate a field `type` with each applied identifier, and this cannot be handled by the Visitor pattern. Another drawback of the Visitor pattern is that the parameter and return types can not be tailored to the different visitors – they must all share the same interface for the visit methods. For example, for type checking expressions, a desired interface could be

```
Type typecheck(Type expectedType)
```

where `expectedType` contains the type expected from the context and the `typecheck` method returns the actual type of the expression. Using the Visitor pattern, this would have to be modelled into visit methods

```
Object visit(C node, Object arg)
```

to conform to the generic visit method interface.

3.2 Aspect-oriented programming

A more powerful alternative to the Visitor pattern is to introduce an explicit modularization mechanism for aspects. This is the approach used in JastAdd. Our technique is similar to the *introduction* feature of the aspect-oriented programming system AspectJ [15].

For each aspect, the appropriate fields and methods for the AST classes are written in a separate file, a *Jadd module*. The JastAdd system is a class weaver: it reads all the Jadd modules and weaves the fields and methods into the appropriate classes during the generation of the AST classes. This approach does currently not support separate compilation of individual Jadd modules, but, on the other hand, it allows a suitable modularization of the code and

typechecker.jadd

```
...
class IfStmt {
  void typeCheck() {
    getExp().typeCheck("Boolean");
    getStmt().typeCheck();
    getOptStmt().typeCheck();
  }
}
class Exp {
  abstract void typeCheck(String expectedType);
}
class Add {
  boolean typeError;
  void typeCheck(String expectedType) {
    getExp1().typeCheck("int");
    getExp2().typeCheck("int");
    typeError = expectedType != "int";
   }
}
...
```

unparser.jadd

```
import Display;

class Stmt {
  abstract void unparse (Display d);
}
class Exp {
  abstract void unparse (Display d);
}
class Add {
  void unparse (Display d) {
    ...
    if (typeError)
       d.showError("type mismatch");
  }
}
...
```

Fig. 5. Jadd modules for typechecking and unparsing.

does not have the limitations of the Visitor pattern.

The Jadd modules use normal Java syntax. Each module simply consists of a list of class declarations. For each class matching one of the AST classes, the corresponding fields and methods are inserted into the generated AST class. It is not necessary to state the superclass of the classes since that information is supplied by the abstract grammar. Figure 5 shows an example. The `typechecker.jadd` module performs type checking for expressions and computes the boolean field `typeError`. The `unparser.jadd` module implements an unparser which makes use ot the field `typeError` to report type-checking errors.

The Jadd modules may use fields and methods in each other. This is illustrated by the unparser module which uses the `typeError` field computed by the type checking module. The Jadd modules may freely use other Java classes. This is illustrated by the unparsing module which imports a class `Display`. The import clause is transmitted to all the generated AST classes. Note also that the Jadd modules use the generated AST access interface described in Section 2. An example of a complete AST class generated by the JastAdd system is shown in Figure 6. In the current JastAdd system, the names of the generated classes are by default prefixed by the string "AST" as in the JavaCC/JJTree system.

3.3 Using the AST as a symbol table

In traditional compiler writing it is common to build symbol tables as large data structures, separate from the parse tree. The use of object-oriented ASTs

ASTAdd.java

```
class ASTAdd extends ASTExp {
   // Access interface
   ASTExp getExp1() { ... }
   ASTExp getExp2() { ...}
   // From typechecker.jadd
   boolean typeError;
   void typeCheck(String expectedType) {
      getExp1().typeCheck("int");
      getExp2().typeCheck("int");
      typeError = expectedType != "int";
   }
   // From unparser.jadd
   void unparse(Display d) {
      ...
      if (typeError)
         d.showError("type mismatch");
      ...
   }
}
```

Fig. 6. Woven complete AST class

makes it convenient to use another approach where the AST itself is used as a symbol table, connecting each AST node that serves as an applied identifier to the corresponding AST node that serves as the declaration. This technique is particularly powerful in combination with aspect-oriented programming. Each part of the compiler that computes a certain part of the "symbol table" can be separated into a specific aspect, imperative or declarative.

Consider the language Tiny in Figure 1. Name analysis involves connecting each applied identifier (`IdUse` node) to its corresponding declared identifier (`Decl` node). For example, taking an imperative approach, this can be implemented by declaring a field `Decl myDecl` in class `IdUse` and by writing methods that traverse the AST and set each such field to the appropriate `Decl` node. Typically, this computation will make use of some efficient representation of the declarative environment, e.g. a hash table of references to the visible `Decl` nodes. But once the `myDecl` fields are computed, the hash table is no longer needed.

Other aspects can add fields and methods to the `Decl` nodes and access that information from the `IdUse` nodes via the `myDecl` field. For example, a type analysis aspect can add a type field to each `Decl` node and access that field from each `IdUse` node during type checking. A code generation aspect can add a field for the activation record offset to each `Decl` node and access that field from each `IdUse` node for generating code.

More complex type information such as structured and recursive types, class hierarchies, etc. is available more or less directly through the `myDecl` fields. For example, a class declaration node will contain a subnode that is an applied identifier referring to the superclass declaration node. More direct access to the superclass can easily be added as an extra field or method of the class

declaration nodes. In this way, once the `myDecl` fields are computed, the AST itself serves as the symboltable.

The different compiler aspects can be implemented as either imperative or declarative aspect modules. Section 4 describes how to implement the name analysis declaratively, defining `myDecl` as a synthesized attribute rather than as a field and specifying its value using equations rather than computing it with imperative methods.

3.4 Adding interface implementations to classes

As mentioned in Section 2.4, aspect modules may add interface implementations to the AST classes. One use of this is to relate AST classes that are syntactically unrelated. As an example, consider implementing name analysis for a language which has many different block-like constructs, e.g. class, method, compound-statement, etc. Each of these block-like constructs should have a method `lookup` which looks up a name among its local declarations, and if not found there, delegates the call to some outer block-like construct. This can be implemented in a name analysis aspect by introducing an interface `Env` with the abstract method `lookup` and adding this interface implementation to each of the involved AST classes.

Another use of interfaces is to relate AST classes to other externally defined classes. One use of this is in order to apply the *Null pattern* for references within the AST. The Null pattern recommends that null references are replaced by references to real (but usually empty) objects, thereby removing the need for specific handling of null references in the code [25]. For example, in the case of an undeclared identifier, the `myDecl` field could refer to a special object of type `NotDeclared`, rather than being null. This can be implemented in a name analysis aspect by introducing an interface `Declaration` whose implementation is added both to the class `NotDeclared` and to the involved AST classes. Naturally, the type of `myDecl` should in this case be changed to `Declaration` as well.

3.5 Combining visitors with aspect-oriented programming

Visitors have serious limitations compared to aspect-oriented programming as discussed earlier. They support modularization only of methods and not of fields, and they do not support type-checking of the method arguments and return values. However, there are certain applications where visitors actually may be slightly simpler to use than Jadd modules, namely when the computation can be formulated as a regular traversal and when the untyped method

visitor – ErrorChecker.java

```
class ErrorChecker extends DefaultTraversingVisitor {
  ErrorCollector errs = new ErrorCollector();

  void visit(IdUse node) {
    if (node.myDecl==null) errs.add(node, "Missing declaration");
  }

  void visit(...
}
```

Jadd module – errorchecker.jadd

```
class Any {
  void errorCheck(ErrorCollector errs) {
    for (int k=0;k<getNumChildren();k++)
      getChild(k).errorCheck(errs);
  }
}

class IdUse {
  void errorCheck(ErrorCollector errs) {
    if (myDecl==null) errs.add(this, "Missing declaration");
  }
}

class ...
```

Fig. 7. Two alternative implementations of error checking

arguments can be replaced by typed visitor instance variables. This is illustrated in Figure 7 where the visitor implementation is slightly simpler than the corresponding Jadd module. In the visitor implementation, the traversal method has been factored out into a superclass `DefaultTraversingVisitor` which can be reused for other visitors. Furthermore, the ErrorCollector object which is used by all visit methods is declared directly in the visitor, rather than supplied as an argument as in the Jadd module.

Visitors and aspect-oriented programming can be freely combined so that each subproblem is solved by the most suitable implementation technique. For example, the `visit(IdUse)` method in the visitor in Figure 7 accesses the field `myDecl` that can be supplied by a Jadd (or Jrag) module.

JastAdd stays backward compatible with JavaCC/JJTree by generating the same visitor support as JJTree (the same `"accept"` methods), thereby allowing existing JJTree projects to be more easily migrated to JastAdd. The visitor support has also been useful for bootstrapping the JastAdd system.

In addition to imperative modules it is valuable to be able to state computations declaratively, both in order to achieve a clearer specification and to avoid explicit ordering of the computations, thereby avoiding a source of errors that are often difficult to debug.

JastAdd supports the declarative formalism Reference Attributed Grammars (RAGs) which fits nicely with object-oriented ASTs. In attribute grammars, computations are defined declaratively by means of attributes and equations. Each attribute is defined by an equation and can be either *synthesized* (for propagating information upwards in the AST) or *inherited* (for propagating information downwards in the AST). An equation defines either a synthesized attribute in the same object, or an inherited attribute in a child object. An attribute can be thought of as a read-only field whose value is equal to the right-hand side of its defining equation.

The important extension in RAGs (as compared to traditional attribute grammars) is the support for reference attributes. The value of such an attribute is a reference to an object. In particular, a node q can contain a reference attribute referring to another node r, arbitrarily far away from q in the AST. This way arbitrary connections between nodes can be established, and equations in q can access attributes in r via the reference attribute. Typically, this is used for connecting applied identifiers to their declarations.

In a Java-based RAG system, the type of a reference attribute can be either a class or an interface. The interface mechanism gives a high degree of flexibility. For example, to implement name analysis, the environment of visible declarations can be represented by a reference attribute `env` of an interface type `Env`. Each language construct that introduces a new declarative environment, e.g., `Block`, `Method`, `Class`, and so on, can implement the `Env` interface, providing a suitable implementation of a function `lookup` for looking up declarations.

RAGs are specified in separate files called *Jrag* modules. The Jrag language is a slightly extended and modified version of Java. A Jrag module consists of a list of class declarations, but instead of fields and methods, each class contains attributes and equations. Ordinary methods may be declared as well and used in the equations. However, in order to preserve the declarative semantics of attribute grammars, these methods should in effect be functions, containing no side effects that are visible outside the method.

The syntax for attributes and equations is similar to Java. Attribute declarations are written like field declarations, but with an additional modifier `"syn"` or `"inh"` to indicate if the attribute is synthesized or inherited. Java method call syntax is used for accessing attributes, e.g., `a()` means access the value of

nameanalysis.jrag

```
1  class Program {
2    getBlock().env = null;
3  }
4  class Block {
5    inh Block env;
6    getStmt().env = this;
7    ASTDecl lookup(String name) {
8      return
9       (getDecl().name().equals(name))
10       ? getDecl()
11       : (env() == null) ? null
12       : env().lookup(name);
13   }
14 }
15 class Stmt {
16   inh Block env;
17 }
18 class BlockStmt {
19   getBlock().env = env();
20 }
21 class AssignStmt {
22   getIdUse().env = env();
23   getExp().env = env();
24 }
25 class Decl {
26   syn String name;
27 }
28 class Exp {
29   inh Block env;
30 }
31 class Add {
32   getExp1().env = env();
33   getExp2().env = env();
34 }
35 class IdUse {
36   inh Block env;
37   syn Decl myDecl= env().lookup(name());
38   syn String name = getID();
39 }
40 class IntDecl {
41   name = getID();
42 }
43 class BoolDecl {
44   name = getID();
45 }
```

Fig. 8. A Jrag module for name analysis.

the attribute `a`. Equations are written like Java assignment statements. Equations for synthesized attributes can be written directly as part of the attribute declaration (using the syntax of variable initialization in Java). For access to components, the generated access methods for ASTs is used, e.g., `getStmt()` for accessing the `Stmt` component of a node.

Jrag modules are aspect-oriented in a similar way as Jadd modules: they add attributes and equations to AST classes analogously to how Jadd modules add fields and methods. The JastAdd system translates the Jrag modules to Java and combines them into a Jadd module before weaving. This translation is described in Section 5.

4.1 An example: name analysis and type checking

Figure 8 shows an example of a Jrag module for name analysis of the language Tiny. (Line numbers are not part of the actual specification.) All blocks, statements, and expressions have an inherited attribute `env` representing the environment of visible declarations. The `env` attribute is a reference to the closest enclosing `Block` node, except for the outermost `Block` node whose `env` is `null`, see the equations on lines 2 and 6. All other `env` definitions are trivial copy equations, e.g., on lines 22 and 23.

The goal of the name analysis is to define a connection from each `IdUse` node to the appropriate `Decl` node (or to `null` if there is no such declaration). This

typechecker.jrag

```
1  class Decl     { syn String type; }
2  class BoolDecl { type = "boolean"; };
3  class IntDecl  { type = "int"; };
4  class Exp { syn String type; };
5  class IdUse {
6    type = (myDecl()==null)
7             ? null : myDecl().type()
8  };
9  class Stmt { syn boolean typeError; };
10 class AssignStmt {
11   typeError = !getIdUse().type().equals(getExp().type());
12 };
     ...
```

Fig. 9. A Jrag module for type checking.

is done by a synthesized reference attribute `myDecl` declared and defined at line 37. Usual block structure with name shadowing is implemented by the method `lookup` on `Block` (lines 7–13). It is first checked if the identifier is declared locally, and if not, the enclosing blocks are searched by recursive calls to `lookup`.

The `lookup` method is an ordinary Java method, but has been coded as a function, containing only a return statement and no other imperative code. As an alternative, it is possible to code it imperatively using ordinary if-statements. However, it is good practice to stay with function-oriented code as far as possible, using only a few idioms for simulating, e.g., let-expressions. Arbitrary imperative code can be used as well, but then it is up to the programmer to make sure the code has no externally visible side effects.

Figure 9 shows a type checking module that uses the `myDecl` attribute computed by the name analysis. This is a typical example of how convenient it is to use the AST itself as a symbol table and to extend the elements as needed in separate modules. The type checking module extends `Decl` with a new synthesized attribute `type` (line 1). This new attribute is accessed in `IdUse` in order to define its `type` attribute (lines 6–7). The types of expressions are then used as usual to do type checking as shown for the `AssignStmt` (line 11).

The examples are written to be self-contained and straight-forward to understand. For a realistic language several changes would typically be done. The copy equations for `env` would be factored out into a common superclass `Any`, thereby making the specification substantially more concise. The type for `env` attributes would typically also be generalized. In the example we simply used the class `Block` from the abstract grammar as the type of the `env` attribute. For a more complex language with several different kinds of block-like constructs, an interface `Env` can be introduced to serve as the type for `env`. Each different block-like construct (procedure, class, etc.) can then implement the `Env` interface in a suitable way. The Null pattern could be applied, both for the `env` and the `myDecl` attributes, in order to avoid null tests such as on line 11 in

Figure 8 and on line 6 in Figure 9. A more realistic language would also allow several declarations per block, rather than a single one as in Tiny. Typically, each block would be extended with a hash table or some other fast dictionary data type to support fast lookup of declarations. Types would be represented as objects rather than as strings, and the type checker would support better error handling, e.g., not considering the use of undeclared identifiers as type checking errors.

It is illustrative to compare the Jrag type checker in Figure 9 with the imperative one sketched in Figure 5. By not having to code the order of computation the specification becomes much more concise and simpler to read than the imperative type checker.

4.2 Combining declarative and imperative aspects

An important strength of the JastAdd system is the ease with which imperative Jadd aspects and declarative Jrag aspects can be combined. A compiler can be divided into many small subproblems and each be solved declaratively or imperatively depending on which paradigm is most suitable. For example, the name analysis and type analysis can be solved by declarative aspects that define the `myDecl` and `type` attributes. Code generation can be split into a declarative aspect that defines block levels and offsets and an imperative aspect that generates the actual code.

It is always safe for an imperative aspect to use attributes defined in a declarative aspect. Usually, this is the natural way to structure a compiler problem: a core of declarative aspects defines an attribution which is used by a number of imperative aspects to accomplish various tasks such as code generation, unparsing, etc.

In principle, it is also possible to let a declarative aspect use fields computed by an imperative aspect. However, for this to be safe it has to be manually ensured that these fields behave as constants with respect to the declarative aspect, i.e., that the computation of them is completed before any access of them is triggered. For example, it would be possible to write an imperative name analysis module that computes `myDecl` fields and let a declarative type checking module access those fields, provided that the name analysis computation is completed before any other computations start that might trigger accesses from the type checking module.

In some attribute-grammar systems, equations are allowed to call methods in order to trigger desired side-effects, e.g., code generation. This technique is used in systems with evaluation schemes that evaluate all attributes exactly once and where the order of evaluation can be predicted. In JastAdd, this

technique is not applicable because of the demand evaluation scheme used which will delay the computation of an attribute until its value is needed. This results in an order of evaluation which is not always possible to predict statically and which does not necessarily evaluate all attributes.

5 Translating declarative modules

The JastAdd system translates Jrag modules to ordinary Java code, weaving together the code of all Jrag modules and producing a Jadd module. Attribute evaluation is implemented simply by realizing all attributes as functions and letting them return the right hand side of their defining equations, caching the value after it has been computed the first time, and checking for circularities during evaluation. This implementation is particularly convenient in Java where methods, overriding, and interfaces are used for the realization. In the following we show the core parts of the translation, namely how to translate synthesized and inherited attributes and their defining equations for abstract and aggregate AST classes.

5.1 Synthesized attributes

Synthesized attributes correspond exactly to Java methods. A declaration of a synthesized attribute is translated to an abstract method declaration with the same name. For example, recall the declaration of the `type` attribute in class `Decl` of Figure 9

```
class Decl { syn String type; }
```

This attribute declaration is translated to

```
class Decl { abstract String type(); }
```

Equations defining the attribute are translated to implementations of the abstract method. For example, recall the equations defining the `type` attribute in `IntDecl` and `BoolDecl` of Figure 9.

```
class IntDecl { type = "int"; }

class BoolDecl { type = "boolean"; }
```

These equations are translated as follows.

```
class IntDecl {
   String type() { return "int"; }
```

```
}
class BoolDecl {
   String type() { return "boolean"; }
}
```

5.2 Inherited attributes

An inherited attribute is defined by an equation in the parent node. Suppose a class `X` has an inherited attribute `ia` of type `T`. This is implemented by introducing an interface `ParentOfX` with an abstract method `T X_ia(X)`. Any class which has components of type `X` must implement this interface. If a class has several components of type `X` with different equations for their `ia` attributes, the `X` parameter can be used to determine which equation should be applied in implementing the `X_ia` method. To simplify accesses of the `ia` attribute (e.g. from imperative Jadd modules), a method `T ia()` is added to `X` which simply calls the `X_ia` method of the parent node with itself as the parameter.

For example, recall the declaration of the inherited attribute `env` in class `Stmt` in Figure 8. Both `Block` and `IfStmt` have `Stmt` components and define the `env` attribute of those components:

```
class Stmt {
   inh Block env;
}
class Block {
   getStmt().env = this;
}
class IfStmt {
   getStmt().env = env();
}
```

Since `Stmt` contains declarations of inherited attributes, an interface is generated as follows:

```
interface ParentOfStmt {
   ASTBlock Stmt_env(ASTStmt theStmt);
}
```

The `Block` and `IfStmt` classes must implement this interface. The implementation should evaluate the right-hand side of the appropriate equation and return that value. The translated code looks as follows.

```
class Block implements ParentOfStmt {
   ASTBlock Stmt_env(ASTStmt theStmt) {
      return this;
```

```
   }
}
class IfStmt implements ParentOfStmt {
   ASTBlock Stmt_env(ASTStmt theStmt) {
      return env();
   }
}
```

The parameter `theStmt` was not needed in this case, since both these classes have only a single component of type `Stmt`. However, in general, an aggregate class may have more than one component of the same type and equations defining the inherited attributes of those components in different ways. For example, an aggregate class `Example ::= Stmt Stmt` could have the following equations:

```
class Example {
   getStmt1().env = env();
   getStmt2().env = null;
}
```

The translation of `Example` needs to take the parameter into account to handle both equations:

```
class Example implements ParentOfStmt{
   ASTBlock Stmt_env(ASTStmt theStmt) {
      if (theStmt==getStmt1())
         return env();
      else
         return null;
   }
}
```

Finally, a method `env()` is added to `Stmt` to give access to the attribute value. The method `getParent()` returns a reference to the parent node. The cast is safe since all AST nodes with `Stmt` components must implement the `ParentOfStmt` interface (this is checked by the JastAdd system).

```
class Stmt {
   ASTBlock env() {
      return ((ParentOfStmt) getParent()).Stmt_env(this);
   }
}
```

The translation described above can be easily generalized to handle lists and optionals. It is also simple to add caching of computed values (to achieve optimal evaluation) and circularity checks (to detect cyclic attribute dependencies and thereby avoid endless recursion) using the same ideas as in other implementations of this algorithm [10,13,20].

6 Related work

Recent developments in aspect-oriented programming [14] include the work on AspectJ [15], subject-oriented programming [7], and adaptive programming [19].

AspectJ covers both static aspects through its *introduction* feature and dynamic aspects through its notion of *joinpoints*. The introduction feature allows fields, methods, and interface implementations to be added to classes in separate aspect modules, similar to how our Jadd modules work. Now that a stable release of AspectJ is available and seems to gain wide-spread use it would be attractive to build JastAdd on top of AspectJ rather than using our own mechanism. The focus in AspectJ is, however, on the dynamic aspects rather than the static aspects. The joinpoint model in AspectJ allows code written in aspects to be inserted at dynamically selected execution points. We do not employ such dynamic aspects in JastAdd, but it is a very interesting area of future work to investigate their benefits in compiler construction.

Subject-oriented programming supports static aspects called *subjects* where each subject provides a (possibly incomplete) perspective on a set of classes. There is a strong focus on how to merge subjects that are developed independently. Explicit composition code is used to specify how to merge subjects, allowing, e.g., different subjects to use different names for the same program entity. This approach is powerful, but also more heavy-weight than the technique used in JastAdd.

Adaptive programming focuses on factoring out traversal code and making it robust to structural changes in the class hierarchy. This separation is similar to what can be accomplished by visitors where default traversal strategies can be factored out in superclasses (as in our example in Figure 7). However, adaptive programming goes beyond visitors in several ways. In particular, they do not require the classes involved to be related in a class hierarchy, and they employ generative techniques to generate traversal code from high-level descriptions.

The *fragment system* is a technique for aspect-oriented modularization which predates the above approaches [18] [16]. It provides a general approach to static aspect modularization based on the syntax of the supported language. By using this mechanism for entities in imperative code, dynamic aspect modularization is also supported to a certain extent. The BETA language uses the fragment system as its modularization mechanism.

There are many compiler tools that generate object-oriented ASTs. An early example was the BETA meta programming system (MPS) [21] which also supported aspect modularization to a certain extent via the fragment system mentioned above. However, due to limitations of the separate compilation mechanism it was only possible to factor out methods and not fields.

The Visitor pattern is supported by many recent compiler tools including JJTree [11], SableCC [5], Java Tree Builder [12], and JJForester [17]. These systems generate AST classes and abstract visitor classes that support various traversal schemes.

There are a few other experimental systems for reference attributed grammars or similar formalisms: the MAX system by Poetzsch-Heffter [23], Boyland's prototype system for the compiler description language APS [3], and our own predecessing system Applab [2]. Similar to JastAdd, these systems stress the modularity with which specifications can be written. In contrast to JastAdd, they all have their own formal languages for specification and do not easily integrate with imperative object-oriented programming in standard languages.

7 Conclusion

We have presented JastAdd, a simple yet flexible and safe system for constructing compilers in Java. Its main features are

- object-oriented ASTs (decoupled from parsing grammars)
- typed access methods for traversing the AST
- aspect modularization for imperative code in the form of fields, methods, and interface implementations
- aspect modularization for declarative code in the form of RAG attributes and equations
- seamless combination of imperative and declarative code

We find this combination very useful for writing practical translators in an easy way. The use of object-oriented ASTs with typed access methods is a natural way of modelling the program. The aspect-modularization is easy to use and makes it easy to change and extend the compiler. We have found it very useful

to be able to combine the declarative and imperative techniques for coding a compiler, making it possible to select the most appropriate technique for each individual subproblem. While subsets of these features exist in other systems we are not aware of other systems that combine them all. In particular, we have not found other Java-based compiler tools that are based on aspect-oriented programming or reference attributed grammars.

We have quite substantial experience from using JastAdd in research and education, and also from bootstrapping the system in itself.

Research projects using JastAdd include a Java-to-C compiler and a tool for integrating Java with automation languages. As a part of these projects a general name analyzer for Java has been developed as a Jrag component. Additional ongoing projects using JastAdd involve translators for robot languages and support for extensible languages.

The JastAdd system is used in our department's undergraduate course on compiler construction. The students work in pairs and use JastAdd to implement a compiler for a small procedural language of their own design and producing SPARC assembly code as output. The course has covered both visitors and aspect-oriented programming using Jadd modules, but not Jrags or attribute grammars.

JastAdd is being bootstrapped in itself. This process has proceeded in several steps. Our starting point was the JavaCC/JJTree system which generates AST classes with untyped access methods and a simple default visitor. The first step was to implement the generation of AST classes with *typed* access methods to allow us to use visitors in a safer way. This step was itself bootstrapped by starting with hand coding the would-be generated AST classes for the abstract grammar formalism (a small amount of code), allowing us right away to use the typed access methods when analyzing abstract grammars. The next step was to use this platform (JJTree-generated visitors and our own generated AST classes with typed access methods) to implement the class weaving of Jadd modules. Once this was implemented we started to use Jadd modules for further implementation, adding the translator for Jrag modules (which generates a Jadd module), and improving the system in general. We are now continuing to improve the system and are also gradually refactoring it to use Jadd and Jrag modules instead of visitors.

The implementation of the JastAdd system is working successfully but we have many improvements planned such as generation of various convenience code, better error reporting, and extensions of the abstract grammar formalism.

There are several interesting ways to continue this research. One is to support modularization not only along phases, but also along the syntax. I.e., it would be interesting to develop the system so that it is possible to supply several

abstract grammar modules that can be composed. Another interesting topic is to explore how dynamic aspect-modularization, for example using joinpoints in AspectJ, can be exploited in compiler construction. Yet another interesting direction is to investigate how emerging aspect-oriented techniques can be applied to achieve language-independent compiler aspects, e.g., name analysis and type analysis modules that can be parameterized and applied to many different abstract grammars. Work in this direction has been done by de Moor et al. for attribute grammars within a functional language framework [22]. We also plan to continue the development of reference attributed grammars and to applying them to new problem areas.

Acknowledgements

We are grateful to Anders Ive and to the anonymous reviewers for their constructive comments. Torbjörn Ekman and Anders Nilsson implemented the Java name analyzer. Many thanks also to the compiler construction students who provided valuable feedback on the system.

References

[1] ANTLR Translator Generator, `http://www.ANTLR.org/`

[2] Bjarnason, E., G. Hedin, K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing* 6(1):36–54 (1999).

[3] Boyland, J. T. *Descriptional Composition of Compiler Components*. Ph.D. thesis. University of California, Berkeley, 1996.

[4] CUP, LALR Parser Generator for Java, `http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[5] Gagnon, E. M., L. J. Hendren, *SableCC, an Object-Oriented Compiler Framework.* In Proceedings of Tools 26–USA'98. IEEE Computer Society, (1998).

[6] Gamma, E. et al., *Design Patterns*, Addison Wesley, 1995.

[7] Harrison, W., H. Ossher, *Subject-Oriented Programming (A Critique of Pure Objects)*, OOPSLA 1993 Conference Proceedings, ACM SIGPLAN Notices, ACM Press, **28(10)** (1993), 411–428.

[8] Hedin, G., *An object-oriented notation for attribute grammars*, ECOOP'89. BCS Workshop Series, Cambridge University Press (1989), 329–345.

[9] Hedin, G., *Reference Attributed Grammars,* Informatica (Slovenia) 24(3): (2000).

[10] Jalili, F., *A general linear time evaluator for attribute grammars*, ACM SIGPLAN Notices, ACM Press, **18(9)** (1983), 35–44.

[11] JavaCC, The Java Parser Generator, `http://www.metamata.com/`

[12] JTB, Java Tree Builder, `http://www.cs.purdue.edu/jtb/`

[13] Jourdan, M., *An optimal-time recursive evaluator for attribute grammars.* In M. Paul and B. Robinet, editors, International Symposium on Programming, 6th Colloquium, LNCS **167** (1984), 167–178. Springer Verlag.

[14] Kiczales, G., et al. *Aspect-Oriented Programming*, ECOOP'97, LNCS **1241** (1997), 220–242. Springer Verlag.

[15] Kiczales, G., et al. An Overview of AspectJ. In J. L. Knudsen, ed., *Proceedings of ECOOP 2001*, 327–353, Budapest, June 2001. LNCS 2072. Springer-Verlag.

[16] Knudsen. J. L. Aspect-Oriented Programming in BETA using the Fragment System. In Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99.

[17] Kuipers T., Visser J. Object-oriented tree traversal with JJForester. In *Proceedings of LDTA'01.* Genova, Italy, April 2001. Electronic Notes of Theoretical Computer Science, Elsevier.

[18] Kristensen, B. B., et al. Syntax-Directed Program Modularization. In P. Degano, E. Sandewall (eds.): *Integrated Interactive Computing Systems*, North-Holland Publishing Company, 1983.

[19] Lieberherr, K., "Adaptive Object-Oriented Software", PWS Publishing Company, 1996.

[20] Madsen, O. L. *On defining semantics by means of extended attribute grammars.* In Semantics-Directed Compiler Generation, LNCS **94** (1980), 259–299. Springer Verlag.

[21] Madsen, O. L., C. Nørgaard. *An Object-Oriented Metaprogramming System.* In proceedings of Hawaii International Conference on System Sciences **21**, (1988).

[22] de Moor, O., S. Peyton-Jones, E. van Wyk. Aspect-Oriented Compilers. In *Generative and Component-Based Software Engineering, First International Symposium*. LNCS 1799, (1999), 121-133. Springer Verlag.

[23] Poetzsch-Heffter, A. Prototyping Realistic Programming Languages Based on Formal Specifications. *Acta Informatica* 34(10):737–772 (1997).

[24] Watt, D. A., D. F. Brown. *Programming Language Processors in Java*, Prentice Hall, 2000.

[25] Woolf, B. The Null Object Pattern. In R. Martin et al. (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1997.