URL: http://www.elsevier.nl/locate/entcs/volume44.html 20 pages

JastAdd—a Java-based system for implementing front ends

Görel Hedin [1] Eva Magnusson [2]

Dept of Computer Science
Lund University
Lund, Sweden

Abstract

We describe JastAdd, a Java-based system for specifying and implementing the parts of compiler front ends that follow parsing. The system is built on top of a traditional Java parser generator which is used for parsing and tree building. JastAdd adds facilities for specifying and generating object-oriented abstract syntax trees with both declarative behavior (using Reference Attributed Grammars (RAGs)) and imperative behavior (using ordinary Java code). The behavior can be modularized into different aspects, e.g. name analysis, type checking, code generation, etc., that are woven together into classes. The class weaving technique has many advantages over the often used Visitor pattern in that it provides a safe and much simpler interface for many applications and in that it supports modularization of not only methods, but also data. Class weaving is also easy and useful to combine with Visitor techniques in order to achieve the best of both. We also describe the implementation of the RAG evaluator (optimal recursive evaluation) which is implemented very conveniently using Java classes, interfaces, and virtual methods.

1 Introduction

Most existing compiler-compilers are focused on scanning and/or parsing and have only rudimentary support for further front end processing. Often, the support is limited to simple semantic actions and tree building during parsing. Systems supporting more advanced front end processing are usually based on dedicated formalisms like attribute grammars and algebraic specifications. These systems often have their own specification language and 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

[1] Email: gorel.hedin@cs.lth.se
[2] Email: eva.magnusson@cs.lth.se

paper we describe JastAdd, a simple yet flexible system which allows static-semantic 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 woven together into the classes of the abstract syntax tree, using techniques related to aspect-oriented programming [12] and subject-oriented programming [5]. A common alternative modularization technique is to use the Visitor design pattern [4]. However, class weaving 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 the front end of a translator, 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 modules are written in ordinary Java code. For declarative modules, JastAdd supports Reference Attributed Grammars (RAGs) [7]. 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 AST itself as a symbol table, and to establish direct connections between identifier use sites and declaration sites. 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 [9]. 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 [2] 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 generated, 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 [3] and JTB [10] are other Java-based system 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 class weaving 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 [15], Jalili [8], and Jourdan [11]: access to attribute values are replaced by functions that compute the semantic function for the value and then cache the computed value for further accesses. A cache flag is used to keep track of if the value has been computed before and is cached, and a cycle flag is used to keep track of attributes involved in an evaluation so that cyclic dependencies can be detected at evaluation time. Our implementation differs 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. An additional difference is that different language constructs can sometimes have the same text representation. For instance, in some languages parameter declarations and variable declarations look the same. A parsing grammar can be more compact by exploiting this

similarity, whereas in an abstract grammar it is important to represent the two constructs as different types since they need to be treated differently in the compilation.

Most parsing systems that support ASTs make use of various automatic rules and annotations in order to support abstraction of the parsing grammar. However, for JastAdd, the abstract grammar is the primary specification and the parsing system subordinate and we do not wish to rely on any specific parsing system for defining the abstract grammar. Therefore, JastAdd has its own abstract grammar specification, and it is the responsibility of the parser to produce abstract syntax trees that follow the abstract 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 pattern of implementation is obvious to any experienced programmer, and documented as the *Interpreter* pattern in [4].

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 [6], 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. Depending on what kind of subcomponents a class has, it is categorized as one of the following typical kinds: *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), and *aggregate* (the class has a set of named 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 an 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). The modifier `"abstract"` is not strictly necessary, but is useful in order to allow behavior in the form of Java method interfaces to be added to the class, without having to supply default implementations.

From this abstract grammar, the Jadd tool 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 subcomponent of the same type, the components are automatically numbered, as for the class `ASTAdd`.

2.4 Additional superclasses

Behavior (methods and fields) is added in separate files called *Jadd modules*, described later in section 3. When adding behavior it is often found that certain behavior is relevant for several classes that 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 shar-

```
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. Tiny.ast – specification of an abstract grammar for Tiny

```
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

ing 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. Additional superclasses are introduced in the abstract grammar. Typically, it is useful to introduce a superclass `Any` that is the superclass of all other AST classes. This is 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.

Such additional 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 this behavior to `Any`. AST classes that introduce new scopes, e.g. `Block`, can then override this behavior.

Java interfaces are more restricted in that they can include only method interfaces and no default implementation. 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 Jadd modules and will be discussed in Section 3.

2.5 Connection to JavaCC

Building the tree

To connect easily to JavaCC/JJTree, the AST classes generated by JastAdd are made subclasses of a class `SimpleNode` that is generated by JJTree. This

allows JJTree to create AST nodes but use its own implementation in `SimpleNode` to connect the nodes into a tree. However, the resulting tree is untyped in the sense that there is no check that the number and types of components of a given node are consistent with the abstract grammar. The compiler writer must be careful to specify tree building actions in JJTree to build the tree in the right way. If the tree is built in the wrong way, AST access methods will fail since they use the primitive `SimpleNode` access methods and then cast the result to the appropriate type. Specification of tree-building actions is error-prone, in particular if the parsing and abstract tree differ in structure. To aid the compiler writer, JastAdd generates a method `syntaxCheck()` for each AST class which can be called to check that a given tree actually follows the abstract grammar.

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`, 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.

Visitors
JJTree includes support for generating `"accept"` methods in AST classes to support programming using the Visitor pattern. JastAdd generates the same methods so that programming in this way is supported although JastAdd rather than JJTree is used for AST class generation. A compiler writer may use this facility as well, although most behavior is easier to implement using class weaving, as will be described in Section 3. The visitor facility has been useful for bootstrapping, implementing JastAdd itself. It is also useful when migrating an existing JJTree-based system to use JastAdd.

3 Weaving Jadd modules

Object-oriented languages lend themselves very nicely to 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 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.

3.1 The Visitor pattern

The Visitor design pattern is one (partial) solution to this problem [4]. 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 the `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 Class weaving

JastAdd uses class weaving for modularizing compiler aspects. For each aspect, the appropriate fields and methods for the AST classes are written in a separate file, a *Jadd module*. The Jadd tool reads all the Jadd modules and inserts the fields and methods into the appropriate classes during the generation of the AST classes. This class weaving approach does not support separate compilation of individual Jadd modules, but, on the other hand, it allows a suitable modularization of the code and does not have the limitations of the Visitor pattern.

The Jadd modules use normal Java syntax. Each file 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 3 shows an example.

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");
    if (expectedType != "int")
      typeError = true;
    else
      typeError = false;
   }
}
...
```

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. 3. Jadd files for typechecking and unparsing.

One Jadd module performs type checking for expressions and another Jadd module implements an unparser which also reports type-checking errors. The Jadd modules may use fields and methods in each other. This is illustrated by the unparser module which uses the error fields 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 Jadd is shown in Figure 4. 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 makes it convenient to use another approach where the AST itself is used as a symbol table. For fast lookup, it is useful to add a hash table or some other fast collection structure. However, the items in that collection can be references to declaration nodes in the AST. Once lookup is done, the appropriate

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");
      if (expectedType != "int")
         typeError = true;
      else
         typeError = false;
   }
   // From unparser.jadd
   void unparse(Display d) {
      ...
      if (typeError)
         d.showError("type mismatch");
      ...
   }
}
```

Fig. 4. Woven complete AST class

declaration reference can be stored in a field of the applied identifier. To access the type of a declaration, simply add a method to all declarations, returning the type of the declaration. Often, it is useful to allow such references to denote both AST nodes and other ordinary Java objects. For example, a missing declaration can be modelled by a static object `MissingDeclaration`. This is easy to handle in Java by letting both the appropriate AST objects and the other objects implement a common interface.

Using the AST itself as a symbol table is particularly powerful in combination with the class weaving modularization technique. Often, different aspects or phases in compilation need different information in the symbol table. Using Jadd modules, the fields and methods can be added to the "symbol table items" (i.e., AST classes) in separate modules, as desired. For example, a Jadd module for code generation can add a field for the activation record offset for each variable declaration, and a method for computing that field.

3.4 Combining class weaving with visitors

Visitors have disadvantages compared to class weaving as discussed earlier. However, for certain applications, visitors are still useful, in particular when some computation over the tree can be formulated as a regular traversal, computing a single aspect of each node. In this case, it can even be slightly easier to formulate the implementation as a visitor than as a Jadd module. For example, consider a visitor that implements error checking. It can be

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

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

  void visit(...
}
```

Fig. 5. Error checking visitor

declared as a subclass of a default visitor which implements each visit method by doing nothing. The default visitor can also contain a general recursive traversal method which traverses the AST and calls `accept` for each node in the AST.

The error checking visitor only needs to implement the visit methods for those AST classes where errors can occur, and it does not need to implement the traversal. Furthermore, fields and methods that all visit methods use, for example the object that collects the error messages, can be declared directly in the visitor. Figure 5 shows a simple error checking visitor which only checks if applied identifiers are declared.

The combination of visitors and class weaving can be highly useful. Visitors can make use of methods and fields provided by Jadd modules. For example, in the error checking visitor the visit method for `IdUse` accesses the field `decl` that is supplied by a Jadd module.

Figure 6 shows the corresponding implementation using a Jadd module. Here, the traversal needs to be written explicitly (in the default implementation of `errorCheck` in class `Any`) and the error-collecting object needs to be supplied as a parameter to `errorCheck`. The implementations are very similar, but the visitor solution is slightly simpler.

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

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

class ...
```

Fig. 6. Error checking Jadd module

4 Using Reference Attributed Grammars

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 RAGs (Reference Attributed Grammars) which fit nicely with object-oriented ASTs. 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 in the AST, 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*. Jrag modules are similar to Jadd modules in that different aspects can be specified in different modules and they both consist of a list of AST class declarations that contain information to be added to the complete AST classes. The Jrag language is, however, not ordinary Java, but a slightly extended and modified language. Each class consists of a list of attribute declarations, method declarations, and equations. Attribute declarations are written like field declarations, but with an additional modifier `"syn"` or `"inh"` to indicate if the attribute is a synthesized or inherited attribute. Java method call syntax is used for accessing attributes, e.g., `a()` means access the value of the attribute `a`. Methods are written in the same way as in Java, but should contain no side effects that are visible outside the method, in order to preserve the declarative semantics of attribute grammars. Equations are written like Java assignment statements. The left-hand side can be either a synthesized attribute in the node itself (declared in its class or any superclass), or an inherited attribute of a component. For access to components, the generated access interface for ASTs is used, e.g., `getStmt()` for accessing the `Stmt` component of a node. Equations for synthesized attributes can be written directly as part of the attribute declaration (using the syntax of variable initialization in Java).

4.1 An example: name analysis

Figure 7 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 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.

```
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   name = getID();
39 };
40 class IntDecl {
41   name = getID();
42 };
43 class BoolDecl {
44   name = getID();
45 };
```

Fig. 7. A Jrag module for name analysis.

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.

The example has been 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` could 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. Instead of using `null` to represent the empty environment, a static object for the empty environment can be defined, simply by letting it implement the `Env` interface. 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 collection structure to support fast lookup of declarations.

4.2 Example continued: type checking

Figure 8 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 2). 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` (lines 11–12).

Again, the example is written to be self-contained and straight-forward to read. For a realistic language, the types would typically be represented in

```
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 =
12     !getIdUse().type().equals(getExp().type());
13 };
     ...
```

Fig. 8. A Jrag module for type checking.

way that would allow for more efficient comparison than strings, for example as integer constants or as references to type objects, maybe using imported Java code and adding more interfaces to the AST classes. The imported Java code must, however, be written with care to supply methods without externally visible side-effects. A more realistic example would also have 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 with the imperative one sketched in Section 3.2. By not having to code the order of computation the specification becomes much more concise and simpler to read than the imperative type checker.

5 Translating Jrag modules

The Jrag tool translates Jrag modules to ordinary Java code, weaving together the code of all Jrag modules and producing a Jadd file. 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 8

```
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 8.

```
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 father node. Suppose a class `X` has an inherited attribute `ia` of type `T`. This is implemented by introducing an interface `FatherOfX` 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 father node with itself as the parameter.

For example, recall the declaration of the inherited attribute `env` in class `Stmt` in Figure 7. 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 FatherOfStmt {
   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 FatherOfStmt {
   ASTBlock Stmt_env(ASTStmt theStmt) {
      return this;
   }
}
class IfStmt implements FatherOfStmt {
   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 FatherOfStmt{
   ASTBlock Stmt_env(ASTStmt theStmt) {
      if (theStmt==getStmt1())
         return env();
      else
         return null;
   }
}
```

Finally, a method is added to `Stmt` to give access to the attribute value. The method `jjtGetParent()` is generated by JJTree (in the superclass `SimpleNode`) and is used to access the parent of the `Stmt` node. The cast is safe since all AST nodes with `Stmt` components must implement the `FatherOfStmt` interface (this is checked by Jrag).

```
class Stmt {
   ASTBlock env() {
      return ((FatherOfStmt) jjtGetParent()).Stmt_env(this);
   }
}
```

5.3 Generalizations

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 [8,11,15].

6 Related work

The weaving technique used here is related to current trends in object-oriented computing like aspect-oriented programming [12], subject-oriented programming [5], adaptive programming [14], and fragment modularization [13]. These techniques are all aimed at modularization of code orthogonally to the normal code structure and supporting this using general techniques. JastAdd's weaving technique relies on the same basic ideas, but is much simpler and light-weight, using an ad hoc weaver for our particular problem (modularization of ASTs).

The idea of using object-oriented ASTs is probably as old as object-orientation itself. The use of it in attribute grammars has been reported before [6]. There are several compiler tools that use OO ASTs, the metaprogramming system (MPS) of the BETA system [16] being an early representative. MPS has certain support for aspect modularization through the BETA fragment system which allows methods to be factored out into different aspects. However, fields can usually not be factored out in order to be able to handle separate compilation.

Another example of a compiler tool using OO ASTs is SableCC [3] which is a Java-based system using an LL-parser generator. The goals of SableCC are similar to those of JastAdd, including supporting a fully typed OO AST in Java and strict separation of handwritten and generated code. However, SableCC provides support only for visitor-based modularization.

The JTB tool (Java Tree Builder) [10] is another JavaCC-based compiler tool supporting visitor-based modularization. It ties these ideas to adaptive programming [14] where one of the goals is to separate traversals from the main code that does the real work in order to make the main code less sensitive to changes in the object structure. As discussed earlier, the visitor pattern and improvements of it are highly relevant to combine with the class weaving and reference attribute grammar techniques reported here.

7 Conclusion

We have presented a simple yet very flexible and safe system for writing compiler front ends in Java. Its main features are object-oriented ASTs (decoupled from parsing grammars), typed access methods, aspect-modularization for both fields and methods, and support for both imperative and declarative code, the latter by means of RAGs. We find this combination very useful for writing practical translators in an easy way. We are currently in the process of bootstrapping the system using itself, and are also using it in a course on compiler construction.

In the bootstrapping process, we started by writing an abstract grammar for the abstract grammar formalism and coding the would-be generated AST classes for that grammar by hand (only 14 classes and 15–20 lines of Java

code per class). To implement the first version of the Jadd tool we used JJTree's visitor functionality, but had great use of the access interface in the AST classes to provide safer access to components. As soon as the Jadd tool could generate AST classes we started to use it for generating its own AST classes (for the abstract grammar) and to improve them. The class weaving functionality is also implemented using visitors, and now that it is working, new enhancements are done by using Jadd modules. The old visitor code is also gradually replaced by Jadd modules. The next step in the bootstrapping process is to gradually add Jrag modules and to replace some of the Jadd modules by Jrag modules.

The implementation of the tool is working successfully but we have many improvements planned such as generation of various convenience code, error checking, etc., and we expect the bootstrapping process and the many student projects to provide even more feedback for improvements.

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 syntax modules that can be composed. Providing several Jadd files for each phase, each matching the AST classes of such a syntax module, would work trivially.

Aspect modularization is a general problem in object-oriented programming, and should be possible to support by a general-purpose mechanism. The current research in the fields of aspect- and subject-oriented programming may produce results that are applicable for our problem. Ideally, Jadd modules should be possible to compile separately, and some appropriate import and visibility mechanisms should be employed so that unintentional name conflicts do not occur as the result of composing unrelated Jadd modules.

Acknowledgements

We are grateful to Anders Ive and to the anonymous reviewers for their constructive comments. 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] CUP, LALR Parser Generator for Java,
`http://www.cs.princeton.edu/~appel/modern/java/CUP/`

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

[4] Gamma, E. et al., "Design Patterns", Addison Wesley, 1995.

[5] 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.

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

[7] Hedin, G., *Reference Attributed Grammars,* in D.Parigot and M. Mernik, eds., Second Workshop on Attribute Grammars and their Applications, WAGA'99, Amsterdam, The Netherlands, (1999), 153–172. INRIA Rocquencourt.

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

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

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

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

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

[13] Kristensen, B. B., et al. *An Algebra for Program Fragments.* In proceedings of ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments, (1985), Seattle Washington.

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

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

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