Draft, December 18, 2000.

JastAdd—a Java-based system for implementing frontends

Görel Hedin, Eva Magnusson
Dept of Computer Science, Lund University, Sweden.
{gorel | eva}@cs.lth.se

Abstract We describe JastAdd, a Java-based system for specifying and implementing the frontend parts of a compiler 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, etc., that are woven together into classes. This combination of object-oriented ASTs and aspect-modularized behavior results in a system which is easier and safer to use than solutions based on, e.g., the Visitor pattern. 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 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, intermediate code generation, etc., that are woven together into the classes of the abstract syntax tree, using techniques related to aspect-oriented programming [9] and subject-oriented programming [3]. A common alternative modularization technique is to use the Visitor design pattern [2]. However, aspect weaving has many advantages over the Visitor pattern, including full type checking of method parameters and return values, and the possibility to associate not only methods but also fields to classes.

When implementing the front-end of a translator, it is often desired to use a combination of declarative and imperative code, allowing results computed by declarative modules to be accessed by imperative modules. 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) [5]. 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 parser generator JavaCC [7]. However, it 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 CUP or ANTLR.

The tree building support in JavaCC is a preprocessor to JavaCC called JJTree and allows easy specification of what AST nodes to generate during parsing. JJTree also supports automatic generation of AST classes. However, it does not merge handwritten code in these classes with the generated code, so it relies on the programmer to modify the generated code and update it by hand after changes to the grammar. This is a very error-prone procedure and is completely avoided in JastAdd. In JastAdd, there is a distinction between generated and handwritten modules, and the programmer never has to modify generated code. The AST classes are generated by JastAdd, rather than relying on the JJTree facility for this. SableCC [1] is another Java-based system that has a similar distinction between generated and handwritten modules, but SableCC relies on the Visitor pattern for adding behavior, and supports only imperative specification of the behavior (using ordinary Java code).

The attribute evaluator used in JastAdd is an optimal recursive evaluator that can handle arbitrary acyclic attribute dependencies. If the dependencies contains 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 [11], Jalili [6], and Jourdan [8]: 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 gives an overview by describing the overall architecture of the system. Section 3 describes the object-oriented ASTs used in JastAdd. Section 4 describes how imperative code can be modularized according to different aspects of compilation and woven together into complete classes. Section 5 describes how RAGs can be used in JastAdd and section 6 how they are translated to Java. Section 7 discusses related work and Section 8 concludes the paper.

2 System architecture

JastAdd contains two separate tools: *Jrag* and *Jadd*. *Jadd* generates abstract syntax tree classes based on a specification in an *.ast* file and weaves imperative aspect code written in *.jadd* files into the generated AST classes. *Jrag* compiles and weaves RAG modules written in *.jrag* files into Java implementations that contain the recursive evaluator. The output from *Jrag* is a *.jadd* file that is woven into the AST classes by the *Jadd* tool. The traditional parser generator (in this case JJTree/JavaCC) is used for creating a parser that creates instances of the AST classes. Additional Java code is written in order to put together the system, i.e., to call the parser and the various imperative

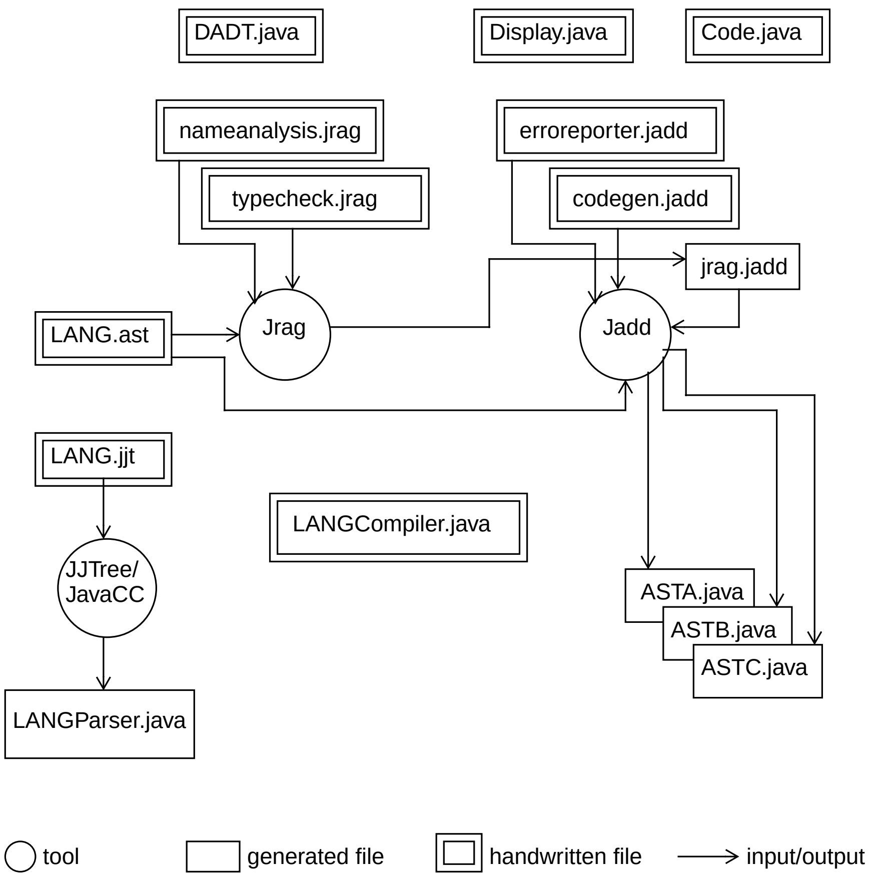

Fig. 1. JastAdd architecture

behaviors. The declarative behaviors need not be called explicitly, since any code using the defined attributes will implicitly start the recursive evaluation of those attributes. Additional Java code can also be written that is used by the *.jrag* and *.jadd* modules.

Figure 1 shows an example of generating a compiler for a language *LANG*, using the JastAdd system. Handwritten modules are shown with double borders and generated modules with single borders. The abstract syntax for *LANG* is specified in the *LANG.ast* module. Declarative behavior is specified in two separate RAG modules: *nameanalysis.jrag* that specifies name analysis, and *typecheck.jrag* that specifies type checking. Both these modules make use of *DADT.java*, an ordinary Java module which contains various abstract data types useful for name and type analysis. These data types are written to have declarative interfaces (no visible side-effects) in order to not conflict with the declarative semantics of the RAG modules. There are two imperative modules: one for error reporting, *erroreporter.jadd*, and one for code generation, *codegen.jadd*. Both these modules make use of attributes computed by the RAG modules.

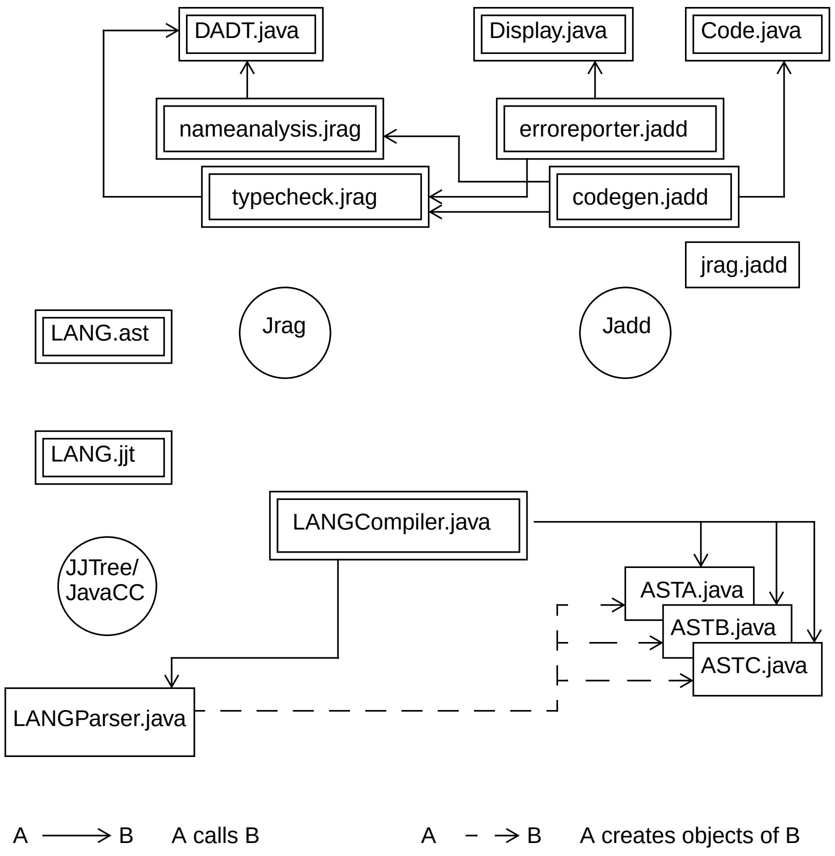

Fig. 2. Module dependencies

They also make use of the auxiliary Java modules *Display.java* (for displaying error messages) and *Code.java* (classes for modelling the instructions). The *Jrag* tool reads the AST and RAG specifications and produces a file *jrag.jadd* as output which implements the behavior specified in the RAGs using the recursive evaluation algorithm. It also produces a number of Java interfaces, *I*.java*, that are used within the evaluation. The *Jadd* tool reads the AST specification and all the *.jadd* files (including the one generated by *Jrag*), and generates a number of AST classes, *AST*.java*, which include the behavior given in the *.jadd* files. The parse syntax for *LANG* is specified in the format required by the traditional parser generator tool, in this case in a file called *LANG.jjt*. The parser generator generates *LANGParser.java*, a parser for *LANG* implemented in Java. The *LANG.jjt* file also includes directives for what AST objects to create during different stages in parsing. Finally, a Java module called *LANGCompiler.java*, functions as the main program for the compiler and calls the parser and the imperative behaviors. Figure 2 gives an overview of how the different modules depend on each other.

3 Object-oriented abstract syntax trees

The basis for specification in JastAdd is an abstract context-free grammar. An abstract grammar is essentially a simplification of a parsing grammar, leaving out the extra nonterminals and productions that resolve parsing ambiguities 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 unnecessary in the abstract grammar. Things important to abstract grammars (but unimportant to parsing grammars) include that behavior should be easy to add to the AST and that traversal and access to subcomponents of the AST should be easy and type safe.

In JastAdd, the abstract grammar is specified separately from the parsing grammar in order to allow each of them to be designed as needed with regard to their different purposes. This also makes JastAdd independent of the underlying parsing system used.

JastAdd makes use of an object-oriented notation for the abstract grammar. A nonterminal is modelled as a superclass and the productions for the nonterminals are named and modelled as subclasses. For example, a nonterminal *Decl* with productions *Decl* → int *ID* and *Decl* → boolean *ID* can be modelled by a superclass Decl with subclasses IntDecl and BoolDecl. For nonterminals with a single production, both the nonterminal and production are modelled by the same class. E.g., a nonterminal *Program* with a single production *Program* → *Block* can be modelled by a single class Program. In order to support easy traversal and access to subcomponents of an abstract syntax tree node, JastAdd supports four different kinds of right-hand-sides for a production class: *list* (a list of components of the same type), *optional* (a single component which is optional), *token* (a component which is a token with a semantic value), and *aggregate* (a set of named components which can be of different types).

3.1 An example: Tiny

Figure 3 shows Tiny.ast, a specification of an abstract grammar for Tiny, a small example block-structured language. (The line-numbers written out to the left allow are not part of the actual specification.) Class IfStmt on line 5 is an example of an aggregate class with three subcomponents (of types Exp, Stmt, and OptStmt respectively). Class CompoundStmt on line 8 is an example of a list class: a CompoundStmt node will have zero or more subcomponents, all of type Stmt. Class OptStmt on line 6 is an example of an optional class: an OptStmt node will either have no subcomponent, or a single subcomponent of type Stmt. Class BoolDecl on line 10 is an example of a token class: a BoolDecl node will have a single subcomponent which is a token of type ID. Class Stmt on line 3 is an example of a class without a right-hand side, corresponding to an ordinary nonterminal. Class Stmt serves as a superclass of the classes BlockStmt, IfStmt, AssignStmt, and CompoundStmt on lines 4, 5, 7, and 8. Class Stmt also has a modifier “abstract”. This means that the corresponding generated Java class will also have the modifier “abstract”. This modifier is not strictly necessary, but is useful is order to allow behavior in the form of Java method interfaces to be added to the class, without having to supply default implementations.

From the .ast specification, the Jadd tool generates a set of Java classes with access interface to their subcomponents. Figure 4 shows some of the generated classes to exemplify the different kinds of access interfaces to classes with different kinds of subcomponents. 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.

3.2 Additional superclasses

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

```
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;
15   ...
```

Fig. 3. 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 {
    int getNumStmts() { ... }
    ASTStmt getStmt(int k) { ... }
}

class ASTBoolDecl extends ASTDecl {
    String getID() { ... }
}

class ASTAdd extends ASTExp {
    ASTExp getExp1() { ... }
    ASTExp getExp2() { ...}
}
```

Fig. 4. Access interface for some of the AST classes generated from the .ast specification

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. Additional superclasses can be added in the .ast specification. 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 .ast specification 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 files and will be discussed in Section 4.

3.3 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 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 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 code using aspect weaving. 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.

4 Weaving jadd files

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 3. Behavior can be implemented easily by introducing abstract methods on nonterminal classes and implement 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.

4.1 The Visitor pattern

The Visitor design pattern is one (partial) solution to this problem [2]. 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 visitC method for each AST class C. All AST nodes have knowledge of an abstract Visitor class to which they can delegate to the appropriate visitC method. For example, for type checking, one could use this technique to implement a TypeCheckingVisitor as a subclass to the abstract Visitor class and implement type checking in the visit methods. There are several limitations to this approach, however. One is that only methods can be factored out; fields must still be declared directly in the classes. For example, in type checking, one typically needs a field `type` for each applied identifier, and this cannot be handled by the Visitor pattern. Another drawback with 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 visitC(Object arg)
```

to conform to the generic visit method interface.

4.2 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 module, an .jadd module. The Jadd tool reads all the .jadd modules and inserts the fields and meth-

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. 5. Jadd files for typechecking and unparsing

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. 6. Woven complete AST class

ods 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 files 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 .ast specification. Figure 5 shows an example. One .jadd module performs typechecking 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 3. An example of a complete AST class generated by Jadd is shown in Figure 6.

4.3 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 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 files, 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 module can add a field for the activation record offset for each variable declaration, and a method for computing that field.

5 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 with declarations. In a Java-based RAG system, the type of a reference attribute can be either a class or an interface. This enables type-safe yet flexible access to attributes in remote objects. The interface mechanism gives a high degree of flexibility. For example, a reference attribute outerScope of an interface type Scope can be used for linking together language constructs that introduce new scopes. Each such language construct, e.g. Block, Method, Class, and so on, simply implements the Scope interface, which may include, e.g., a function lookup for looking up declarations in that scope.

RAG 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 RAG 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. 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 (that was described in Section 3), 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).

5.1 An example: name analysis

Figure 7 shows an example of a RAG 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.

```
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. RAG module for name analysis

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. 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 could be done. The copy equations for env could be factored out into a common superclass Any, thereby making the specification substantially more concise. Rather than using Block as the type of env, an interface Env with a function lookup could be introduced. This would allow the language to be easily extended with other block constructs, e.g. procedures with parameters. Instead of using null to represent the empty environment, a static object for the empty environment could be defined. Similar changes could be done for the myDecl attribute. A more realistic language would also allow several declarations per block,

```
class Decl {
    syn String type;
}
class BoolDecl {
    type = "boolean";
};
class IntDecl {
    type = "int";
};
class Exp {
    syn String type;
};
class IdUse {
    type = (myDecl()==null) ? null
                 : (myDecl().type()==null) ? null
                     : myDecl().type()
};
class Stmt {
    syn boolean typeError;
};
class AssignStmt {
    typeError = !getIdUse().type().equals(getExp().type());
};
....
```

Fig. 7. A RAG module for type checking

rather than a single one as in Tiny. The solution is easily generalized to take care of that.

5.2 Example continued: type checking

Figure 7 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 (line 14). The types of expressions are then used as usual to do type checking as shown for the AssignStmt (line 22).

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 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 RAG type checker with the imperative one sketched in Section 4.2. By not having to code the order of computation the specification becomes much more concise and simpler to read than the imperative type checker.

6 Translating RAG modules

The Jrag tool translates RAG modules to ordinary Java code, weaving together the code of all RAG 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.

6.1 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 7

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

This attribute declaration is translated to

```
class ASTDecl {
   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 7.

```
class ASTIntDecl {
   type = “int”;


class ASTBoolDecl {
   type = “boolean”;
}
```

These equations are translated as follows.

```
class ASTIntDecl {
   String type() {
      return “int”;
   }
}
class ASTBoolDecl {
   String type() {
      return “boolean”;
   }
}
```

6.2 Inherited attributes

An inherited attribute is defined by an equation in the father node. To represent this in Java, we add an interface FatherOfX for each class X declaring an inherited attribute. Each class which has components of type X must implement this interface. The interface contains abstract methods for computing the inherited attributes. In addition, methods for the inherited attribute of X are added to X. These methods call the corresponding method of the father node in an appropriate way.

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 ASTStmt {
   inh Block env;
}
```

```
class ASTBlock {
   getStmt().env = this;
}

class ASTIfStmt {
   getStmt().env = env();
}
```

Since ASTStmt 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 ASTBlock implements FatherOfStmt {
   ASTBlock Stmt_env(ASTStmt theStmt) {
      return this;
   }
}

class ASTIfStmt implements FatherOfStmt {
   ASTBlock Stmt_env(ASTStmt theStmt) {
      return env();
   }
}
```

The parameter theStmt was not needed in this case, since both these classes had 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 jjt-GetParent() 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 ASTStmt {
            ASTBlock env() {
                return ((FatherOfStmt) jjtGetParent()).Stmt_env(this);
            }
        }
```

6.3 Generalizations

The translation described above can be easily generalized to handle lists, optionals, caching of computed values (to achieve optimal evaluation), and circularity checks (to detect cyclic attribute dependencies and thereby avoid endless recursion).

7 Related work

The weaving technique used here is related to current trends in object-oriented computing like aspect-oriented programming [9], subject-oriented programming [3], and fragment modularization [10]. 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 [4]. There are several compiler tools that use OO ASTs, e.g. the metaprogramming system (MPS) of the BETA system [12]. 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 [1] 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.

8 Conclusion

We have presented a simple yet very flexible and safe system for writing compiler frontends in Java. Its main features are Java-based OO ASTs (decoupled from parsing grammars), aspect-modularized code, 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 will also use it in a course on compiler construction in the spring of 2001.

References

1. E. Gagnon. SableCC, an Object-Oriented Compiler Framework. PhD thesis. McGill University, 1997.
2. E. Gamma et al. Design Patterns. Addison Wesley 1995.
3. W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). OOPSLA 1993 Conference Proceedings, ACM SIGPLAN Notices, 28(10), pp. 411-428, ACM Press, October 1993.
4. G. Hedin. An object-oriented notation for attribute grammars. *ECOOP'89*. BCS Workshop Series, pp 329-345, Cambridge University Press. 1989.
5. G. Hedin. Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 153-172, Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.
6. F. Jalili. A general linear time evaluator for attribute grammars. *ACM SIGPLAN Notices, Vol 18(9):35-44, September 1983*.
7. JavaCC. http://www.metamata.com/
8. M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer-Verlag, 1984.
9. G. Kiczales et al. Aspect-Oriented Programming. ECOOP'97, LNCS 1241, pp. 220-242, Springer-Verlag, June 1997.
10. B. B. Kristensen et al. An Algebra for Program Fragments. In proceedings of ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments, June 1985, Seattle, Washington
11. O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation*, pp 259-299, LNCS 94, Springer-Verlag, January 1980.
12. O. L. Madsen, C. Norgaard. An Object-Oriented Metaprogramming System, In proceedings of Hawaii International Conference on System Sciences - 21, January 5-8, 1988