LUND UNIVERSITY

Compiling Java for Real-Time Systems

Nilsson, Anders

2004

[Link to publication](#)

Citation for published version (APA):
Nilsson, A. (2004). *Compiling Java for Real-Time Systems.* [Licentiate Thesis, Department of Computer Science]. Department of Computer Science, Lund University.

Total number of authors:
1

Compiling Java for Real-Time Systems

Anders Nilsson

Department of Computer Science, Lund University
Box SE-221 00 Lund
Sweden
anders.nilsson@cs.lth.se

Abstract. The tools used for constructing compilers have not changed much during the last decade. Typically, a parser generator (for example lex/yacc, or bison) is used for constructing a parser, and the remaining parts (semantic analysis, optimizations, and code generation) need to be done by hand.

Advances in compiler construction research have resulted in new object-oriented tools based on reference attributed grammars and aspect oriented programming, which could be very beneficial for developing compilers for modern object-oriented languages.

We have used the JastAdd compiler construction tool for developing a Java compiler for real-time systems. In this process, we could also evaluate the possible benefits from using the JastAdd tool in a complete real-world programming language compiler.

The usage of reference attributed grammars and aspect-oriented programming renders a rather compact, yet apprehensible and elegant compiler specification where code analysis, refactorings, and optimizations can be conveniently described as aspects performing computations and transformations on the AST.

1 Introduction

Constructing a compiler for a modern Object-Oriented (OO) language, such as Java, using standard compiler construction tools is normally a large, tedious, and error prone task. Work on compiler construction within our research group has resulted in new ideas and new compiler construction tools [1], which with the aim of this work represent state of the art. The representation of the language within that tool is based on Attribute Grammars (AGs). AG-based research tools have been available for a long time, but there are no known compiler implementations for a complete object-oriented language, so this topic also by itself forms a research issue.

The tools used for constructing compilers have not changed much during the last decade. Typically, the concrete grammar for the to-be-compiled language is specified according to the parser generator of preference (for example lex/yacc, or bison). The generated parser parses the source code and builds an Abstract Syntax Tree (AST). Hand-crafted code is then typically used for performing static-semantic analysis on the AST, generate some kind of intermediate code,

perform various optimizations on the intermediate code, and finally generate assembly- or machine code—possibly with a final optimization pass. Due to the large amount of hand-crafted code needed for the analysis-, code generation-, and optimization phases, the task of constructing a compiler, from scratch, for a modern OO language can be overwhelming.

1.1 Aspect-Oriented Programming

In 1997, Kiczales et al. published a paper [3] describing Aspect-Oriented Programming (AOP) as an answer to many programming problems, which do not fit well in the existing programming paradigms. The authors have found that certain design decisions are difficult to capture—in a clean way—in code because they cross-cut the the basic functionality of the system. As a simple example, one can imagine an image manipulation application in which the developer wants to add conditional debugging print-outs just before every call to a certain library matrix function. Finding all calls is tedious and error-prone, not to mention the task of, at a possible later time, removing all debug print-outs again. These print-outs can be seen as an *aspect* on the application, which is cross-cutting the basic functionality of the image manipulation application.

By introducing the concept of programming in *aspects*, which are woven into the basic application code at compile-time, two good things are achieved; the basic application code is kept free from disturbing add-ons (conditional debugging messages in the example above), and, the aspects themselves can be kept in containers of their own with good overview by the developers of the system.

The tool *aspectj* [4] was released in 2001 to enable Aspect-Oriented Programming (AOP) in Java. There is also a web site for the annual aspect oriented software development conference[1] where links to useful information and tools regarding AOP are collected.

1.2 Reference Attributed Grammars

Ever since Donald Knuth published the first paper [5] on Attribute Grammar (AG) in 1968, the concept has been widely used in research for specifying static semantic characteristics of formal (context-free) languages. The AG concept has though never caught on for use in production code compilers.

By utilizing Reference Attribute Grammars (RAGs) [6], it is also possible to specify in a declarative way the static semantic characteristics of object-oriented languages with many non-local grammar production dependencies.

The compiler construction toolkit, *JastAdd*, which we are using for developing a Java compiler is based on the Reference Attribute Grammar (RAG) concept.

Using the *JastAdd* [1] compiler construction toolkit, developed at our department, we are developing a compiler for real-time Java. The back-end generates C code according to the ideas described in Chapter [7], which is compiled and linked against the Garbage Collector Interface (GCI) and runtime, as described

---

[1] `http://aosd.net`

in Chapter [8], to produce a time predictable executable suitable also for hard real-time-systems.

Jast Add. The JastAdd system [1, 9] is based on current research on Reference Attribute Grammars (RAGs) and Aspect-Oriented Programming (AOP). The goal of the JastAdd system is to provide compiler developers with a better tool for AST manipulations than those available today.

Using the RAG technique makes it possible to declare semantic equations that state how to compute attributes from an AST. These equations present a convenient way of implementing name- and type analysis, and can also be used for rewriting subtrees of the AST on demand while computing attributes.

AOP is a good help for separation of concerns in the compiler implementation. Different aspects can be kept in separate source code modules which enhances readability, and also makes it possible to add or remove specific aspect modules during implementation or debugging. It is also possible to integrate ordinary Java code modules, if desired, using the JastAdd system.

Input to JastAdd is divided in two parts; an abstract grammar definition of the language, and a set of aspects which will be woven into the AST node classes. The abstract grammar defines both the context-free grammar of a language, and the inheritance hierarchy of the node classes comprising an AST. The other part of a JastAdd system is a set of aspects, usually a mix of ordinary Java code with semantic equations, which are woven in as Java code into the node classes.

The JastAdd tool does not include support for building a concrete parser which generates the AST. Any parser generator capable of constructing a parser which can build Java ASTs may be used as a front-end. JavaCC [10] is used in our Java compiler implementation, but CUP [11] has also been used in other JastAdd experiments.

2 Architecture and Overview

The architecture of our compiler differs from most available compilers, in that there is no explicit symbol table, nor will it generate internal intermediate code. Instead, all operations are implemented as methods on the AST nodes using the JastAdd system. A concrete grammar description is used to create a parser, while an abstract grammar describes the AST node class hierarchy. A collection of aspects, including name- and type analysis, optimizations and code generation, are woven into the node classes. The parser, node classes, and auxiliary handwritten Java code makes up the compiler, see Fig. 1.

Abstract Grammar. The purpose of the abstract grammar definition is twofold; at the same time as it specifies the node relations of an AST, it also specifies the class hierarchy for the node classes. As an example, consider the small excerpt from our Java grammar in List. 1.1. The corresponding node class inheritance hierarchy is shown in Fig. 2.
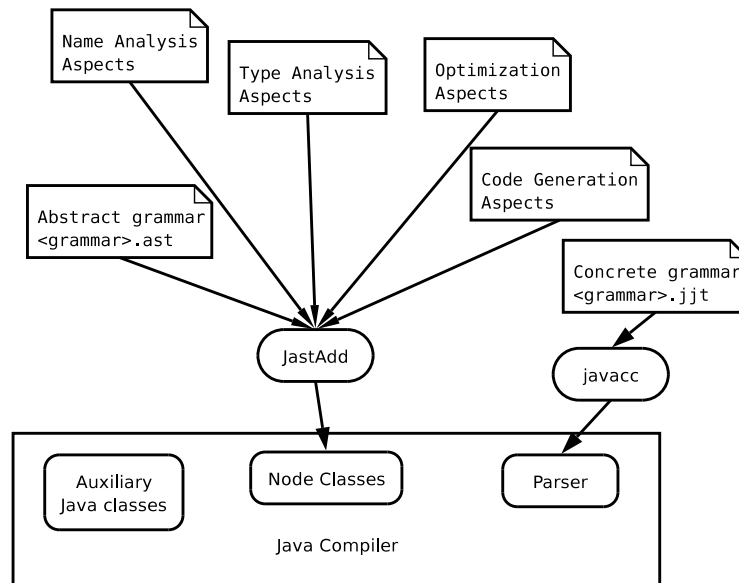
Fig. 1. Overview of the Java compiler architecture.

As can be seen in the listing and figure, the abstract class `Expr` is inherited by all other expression classes, which enables elegant implementations of methods common to all expressions, as exemplified later in this section. The abstract classes `Binary` and `Unary` are analogously inherited by all respective concrete classes.
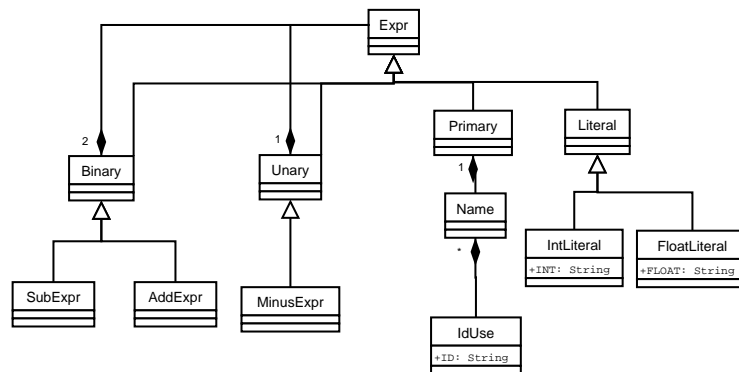

Fig. 2. Node class relations in simple JastAdd example.

Using the grammar shown in List. 1.1, we can build an AST representation of the code snippet

Listing 1.1. A small example of the JastAdd abstract grammar definition.

```
abstract Expr;
abstract Binary:Expr ::= Left:Expr Right:Expr;
abstract Unary:Expr ::= Expr;

AssignExpr : Expr ::= Dest:Expr Source:Expr;

AddExpr:Binary;
SubExpr:Binary;

MinusExpr:Unary;

Primary:Expr ::= Name;
Name ::= IdUse*;

IdUse ::= <ID>;

abstract Literal:Expr;
IntLiteral:Literal ::= <INT>;
FloatLiteral:Literal ::= <FLOAT> ;
```

```
a.b = c.d + e - 2;
```

as shown in Fig. 3. Taking advantage of the inheritance hierarchy of the AST node classes, we can now define aspects operating on this AST representation of a program.

JastAdd Aspects. Aspects in the JastAdd system are used for implementing the operations to be performed on the generated AST. They can be implemented either as normal Java code, methods and attributes woven into the AST node classes, or as RAG semantic equations which are translated into Java code by JastAdd, and then woven into the AST classes.

Considering the Java assignment expression above, the two types of aspects can be illustrated with two of the operations a compiler would typically perform on code; type checking, and code generation. Implementing type checking for this subset of Java expressions is conveniently done using semantic equations, and is shown in List. 1.2 below.

The `class TypeCheck` declaration does not, in this example, have any other semantic meaning other than being a syntactic placeholder for declarations (similar to the `aspect` declaration in AspectJ). A synthetic attribute, `syn TypeDecl type`, is declared in the `Expr` and `Name` classes with default values, and overridden in some subclasses of `Expr`. Semantic equations are written as assignments, and will be transformed to Java methods by the JastAdd system as can be noted in, for example, the declaration of `Unary.type` where the `type` attribute in its `Expr` child is evaluated by calling the generated `type()` method.
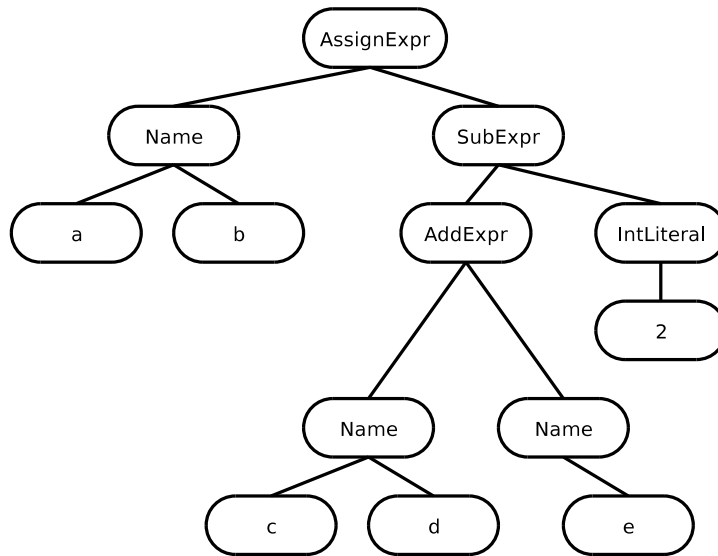
Fig. 3. AST representation of the Java expression `a.b=c.d+e-2`, according to the grammar in List. 1.1.

Analogously to the synthetic attribute shown in this example, there are also inherited attributes which are used to propagate information downwards in the AST.

Code generation from an AST representation is not equally suited to describe in the form of semantic equations, as is for example type checking. Printing is operational in nature, and it is thus more convenient to implement a code generator using imperative code, even though it would be possible to generate code by evaluating one large string attribute. As an example of using imperative code in JastAdd, consider the pretty-printer example in List. 1.3 below. As can

Listing 1.2. Type checking implemented using semantic equations in JastAdd.

```
class TypeCheck {
  syn TypeDecl Expr.type = null;
  syn TypeDecl Name.type = lookupTypeDecl();
  Binary.type = LeastCommonType(getLeft().type(),
                                getRight().type());
  Unary.type = getExpr().type();
  Primary.type = getName().type();
  IntLiteral.type = lookupType("int");
  FloatLiteral.type = lookupType("float");
}
```

Listing 1.3. Pretty-printer implemented using Java aspects in JastAdd.

```
class PrettyPrint {
  abstract void Expr.prettyPrint(PrintStream out);
  syn String Binary.operator = "";
  String AddExpr.operator = "+";
  String SubExpr.operator = "-";

  void Binary.prettyPrint(PrintStream.out) {
    getLeft().prettyPrint(out);
    out.print(operator());
    getRight().prettyPrint(out);
  }
  void MinusExpr.prettyPrint(PrintStream.out) {
    out.print("-");
    getExpr().prettyPrint(out);
  }
  void Name.prettyPrint(PrintStream.out) {
    for (int i=0; i<getNumIdUse(); i++) {
      getIdUse().prettyPrint(out);
      if (i<getNumIdUse()) out.print(".");
    }
  }
  void IdUse.prettyPrint(PrintStream out) {
    out.print(GetID());
  }
  void Literal.prettyPrint(PrintStream out) {
    out.print(GetLITERAL());
  }
}
```

be noted in the example, one can mix use of semantic equations with imperative code.

3   Simplification Transformations

Generating code from an AST representation can be rather cumbersome , depending on the AST topography[2] and the complexity of the parsed language. Especially, expressions in Java may be rather complex, as for example in the code fragment with corresponding AST in Fig.4.

Building a code generator capable of handling arbitrary expressions tends to be a very complex and error-prone task. Instead, by defining the simplest possible

---

[2] The parser usually does not have all the information needed for building a semantically "good" AST, but instead builds an AST syntactically close to the source code, see Fig. 4.
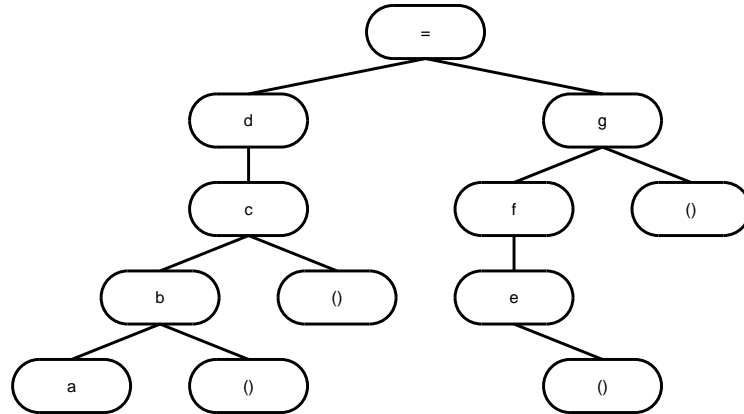
```
a.b().c().d = e().f.g();
```


Fig. 4. Java code fragment and corresponding AST.

Java language subset while not restricting the semantics, the code generator becomes much simpler and less error-prone, see [12] for a definition of such a Java language subset.

The mapping from the full Java language specification [13] to the simpler subset can be conveniently described as a set of transformation on the AST, as will be shown in the following sections.

Names. Most of the simplifying transformations needed to perform on the AST are consequences of real-time memory management, see [8, 14] for details. Memory operations on references are performed via side-effect macros, only allowing one level of indirection at each step. It is therefore necessary to transform all Java expressions with more than one level of indirection into lists of statements each containing at most one level of indirection. For example, the Java statement

```
a.b = c;
```

contains one indirection, whereas

```
a.b = c.d;
```

has two indirections, and must therefore be transformed into something like

```
tmp_1 = c.d;
a.b = tmp_1;
```

or, described as a transformation on the AST in Fig. 5, to meet the indirection level requirements.

The situation becomes a little more complicated with method calls, since arguments passed in the call may contain arbitrarily complex expressions. By studying the method call
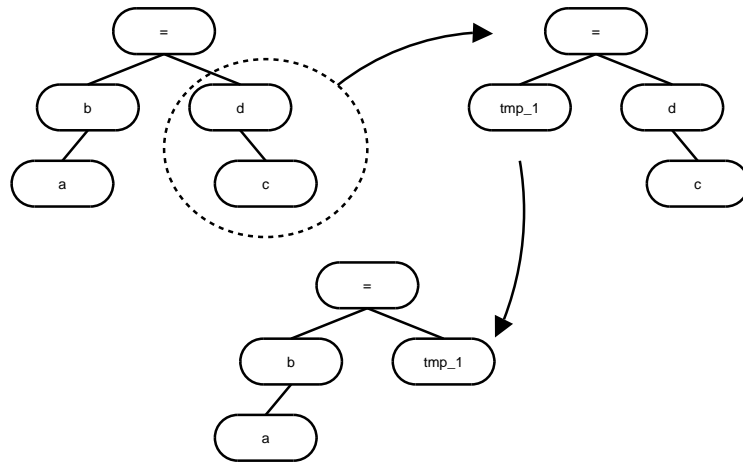
Fig. 5. Simplifying names by means of an AST transformation.

```
a(b(c()),d());
```

we may soon see that the evaluation order of the method calls must be

```
c(), b(c()), d(), a(b(c()),d())
```

A suitable simplifying transformation for the above expression, to meet the indirection requirements, could then be expressed as AST transformations or as code as in Fig. 6.

The aspect code needed for performing the simplification transformations shown in Fig.s 5 and 6 is shown below in List. A.

Unary Expressions. Unary expressions which as a side effect changes the value of the operand, may need to be simplified in order to meet indirection requirements. For example, the simple statements

```
a++;
b.a++;
```
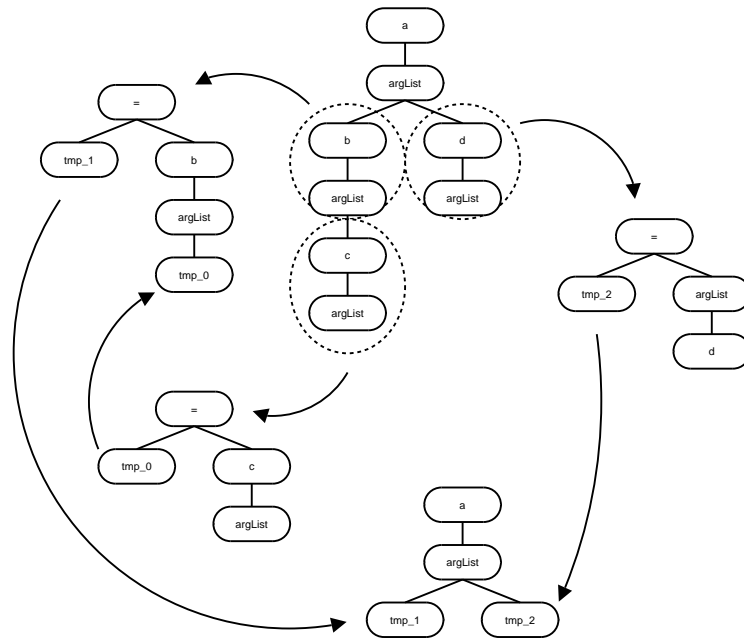
should be read as

```
a = a+1;
b.a = b.a+1;
```

which poses no problem in the first statement, with zero indirections, but the latter statement now has two indirections and must be simplified to something like

```
tmp_0 = b.a;
b.a = tmp_0+1;
```

However, things get more complicated as such unary expressions may be used inside other expressions. For example, the seemingly simple statement

```
tmp_0 = c();
tmp_1 = b(tmp_0);
tmp_2 = d();
a(tmp_1,tmp_2);
```

Fig. 6. Simplifying a complex method call.

```
a[k.i++] = b[++k.i];
```

has a non-trivial evaluation order. A simplification of the above statement which meet indirection requirements can be written as:

```
tmp_0 = k.i;
++tmp_0;
k.i = tmp_0;
tmp_1 = b[tmp_0];

tmp_2 = k.i;
k.i = tmp_2 + 1;

a[tmp_2] = tmp_1;
```

Note that the evaluation of a *PreIncrement* expression differs from the evaluation of a *PostIncrement* expression to maintain semantic correctness.

Control-Flow Statements. The expression—or expressions—which is an important part of all control-flow statements require special care in the simplifi-

cation process, so as not to alter the semantics of the program. Only the for-statement will be described here, as it is—semantically—the most complicated control-flow statement in Java.

A Java for-statement, as defined by the abstract grammar for Java, is represented by the AST subtree in Fig. 7. As defined in the Java language specifi-
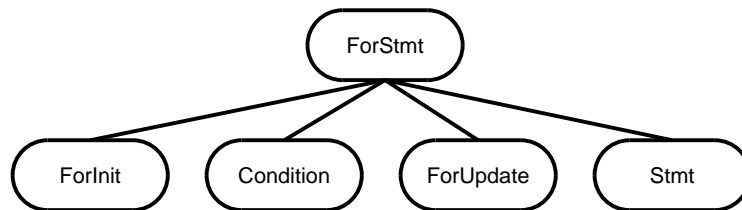
Fig. 7. Subtree representing a for-statement.

cation [13], the *ForInit* and *ForUpdate* nodes may hold arbitrary lists of *StatementExpressions* or, in the case of *ForInit*, a *variableDeclaration*. An example of a complex `for`-statement could be

```
for(a=b(c(1),d),e=f[g()];a[h++]<i;a=b(c(h++)),d)
   // code
```

The solution to simplifying complex for-statements is to, in fact, create `while`-statements by moving the *ForInit* ahead of the statement and move the *ForUpdate* last inside the *Stmt* node (which has been transformed to a *Block*). A simplified for-statement subtree is shown in Fig. 8. The resulting code after simplifying the example for-statement above would then be

Fig. 8. Subtree representing a simplified for-statement.

```
tmp_0 = c(1);
a = b(tmp_0,d);
tmp_1 = g();
e = f[tmp_1];

tmp_2 = a[h++];
for ( ; tmp_2<i ; ) {
  // code

  tmp_3 = c(h++);
  a = b(tmp_3,d);

  tmp_2 = a[h++];
}
```
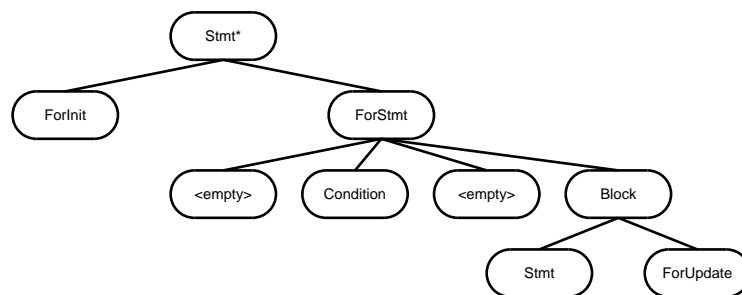
Similar techniques are used to simplify the other Java control-flow statements.

4   Optimization Transformations

Also in cases when compiling to some kind of pseudo-high-level intermediate language (such as C), there is need for some optimizations at the higher abstraction level which can not be taken care of by the intermediate language compiler. Examples of such optimizations are typical OO optimizations, such as implicit finalization of method calls, class in-lining, but also, depending on the object model, (high level) dead code elimination. Of these optimizations, only dead code elimination is currently implemented in our compiler.

4.1   Dead Code Elimination

Constructing an AST based on static dependencies between classes in an application clearly results in a set of type declarations including a subset of the Java2 Standard Edition (J2SE) standard classes. However, the J2SE is so designed that, for any application, this subset will include >200 type declarations. A static analysis of all possible execution paths of the application reveals that there exist a set of type declarations, possibly referenced during execution, which includes much fewer classes than static dependencies would suggest. It has also been shown by Tip et al. [15] that there is much to gain regarding the application size if also referenced type declarations are stripped of unused code, such as attributes, methods, and constructors.

Dead-code elimination requires static compilation of the program to be optimized, as dynamically loaded code may try to reference methods or fields which were previously unreachable. It should also be performed using whole-program analysis, since otherwise only private methods and fields may be analyzed.

Implementation. We have implemented dead code elimination in our Java compiler using JastAdd aspects to calculate the transitive closure of an application, starting from the application `main` method and all `run` methods found

in thread objects. Encountered methods and constructors are marked as *live*, as are type declarations with referenced constructors, methods, or fields. During the code generation pass only code for *live* types, constructors, and methods will be generated.

Evaluation. The dead code optimization algorithm has been tested on a couple of applications, with good results, as seen in Table 1 below. The two applications are described in Sect. 6.

Application	Without opt. (kB)	With opt. (kB)
HelloWorld	316	218
Robot Controller	1059	759

Table 1. Code size results from utilizing dead code optimization on some applications.

5   Code Generation

When the AST has been transformed, as described in Sect. 3, to reflect the simplest possible Java coding style, the task of generating intermediate code—in this case C code—becomes relatively simple.

First, a C header file is generated for each used class in the AST, containing the type declarations of the object model. Handwritten C code, such as native method implementations, can then include appropriate class headers. Then, one C file containing the actual implementations of all constructors and methods, as well as class initialization code.

Header Files. The organization of the header files is sketched below as:

<**class**>_**ClassStruct** A C struct representing the class. Has pointers to the class's super class struct, and a pointer to this class' virtual methods table. Only one instance of this struct exist in run-time.

<**class**>_**StaticStruct** A struct containing static fields of this class, and all ancestors. Only one instance exist in run-time.

<**class**>_**ObjectStruct** A struct representing an instantiated object of this class. Contains a pointer to the class struct and all non-static fields of this class (including ancestors).

<**class**>_**MethodStruct** The virtual methods table associated with objects instantiated from this class. Contains function pointers for all methods of objects of this class. One instance of this struct exist in run-time.

C code file. The organization of the generated C code files is sketched below as:

- Include necessary header files
- Declare the static object model structs for each class/interface; class, class static, object layout, object static layout, vtable, interface table (if applicable).
- Declare function prototypes for all constructors and methods. This is needed since declare/use order of these is free in Java.
- All function (methods and constructors) implementations.
- The Java classes init function. Pushes layouts on the GC root stack, fill in virtual method tables, and initialize static attributes.

The process of compiling a Java program to an executable machine code image is sketched in Fig. 9.
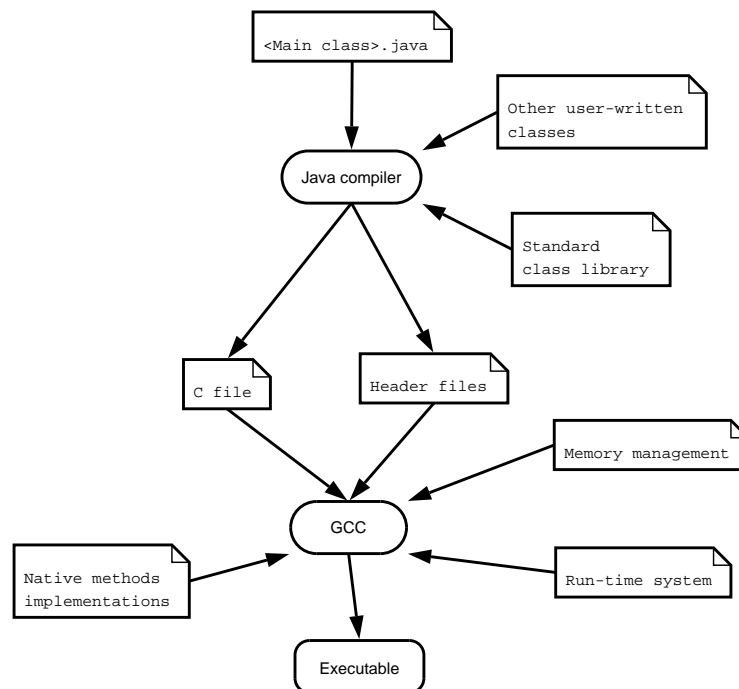

Fig. 9. Flowchart of compilation process.

6 Evaluation

The use of a modern RAG-based compiler construction toolkit, JastAdd, lead to a rather compact—yet modular and easy to read—compiler specification. Our

Java compiler, as of today, comprises only about 10000 lines of source code including abstract grammar, concrete grammar, and all aspects needed for semantic analysis, simplifications, optimizations, and code generation needed for generating real-time capable C code. The sizes of the modules of our compiler are listed in Table 2.

	Lines of code
Front-End	
Abstract Grammar	181
Concrete Grammar	1044
Semantic Analysis	
Name- and Type Analysis	1458
Transformations and Optimizations	
Simplifications	901
Dead Code Optimization	154
Code Generation	
Code generation	5745

Table 2. Source code sizes for the different stages of our compiler.

The current version of the compiler front-end (parser and static semantic analysis) is fully compatible with the current Java standard, version 1.4. Code generation still lacks support for some features of the Java language, most notably inner- and anonymous classes, but the implementation of these features is quite straight-forward and will not add more than, at most, some hundred lines of aspect code to the compiler.

Preliminary Benchmarks. Our Java compiler is still very much in development and very little effort has been spent on compiler speed, but to get an idea on how much slower it is compared to available Java compilers, some very preliminary benchmarks have been performed.

The test platform was an ordinary PII 300MHz workstation with 128 MB of RAM. The operating system was Debian GNU/linux, kernel version 2.4.19, and the Java environment is the Sun J2SE version 1.4.1. As reference Java compilers we used javac version 1.4.1 and gcj version 3.3.3.

Two applications were used to benchmark our compiler against the references. *HelloWorld* is a very small one-class application, basically just instantiating itself and printing the words "Hello World" on the terminal. The *RobotController* is a much larger application consisting of about 25 classes, implementing one part of a network-enabled controller for an ABB industrial robot. For some reason, possibly due to the use of native methods, it was not possible to compile the robot controller application using gcj.

As can be seen in Table 3, our Java compiler is substantially slower than the other tested compilers. One main reason is the two-pass nature of our compiler (see Fig. 9), the time needed for gcc to compile the generated C file exceeds 90 s

	Our compiler	gcj	javac
HelloWorld			
Memory usage (MB)	14	<5	21
Time (s)	26	0.65	3
RobotController			
Memory usage (MB)	34	-	30
Time (s)	160	-	9

Table 3. Java compiler measurements

itself. Another reason for the large difference in compilation times is simply that compiler performance has been, and still is, of low priority in the compiler development process. Nevertheless, separate compilation of Java classes, and using more modern computers, would surely decrease compilation times significantly.

Observations. The modularization of a computer achievable using JastAdd benefits compiler development in a number of ways. Some examples include:

Instrumentation The compiler can be instrumented with code for debugging, for example an aspect to dump information in AST nodes.

Measurements Code can be added for measuring, for example the effect of various optimizations.

Experiments A compiler developer can try experimental code, which is easy to remove later. For example, a new optimization can be written as a JastAdd aspect and tested. If it turns out to be useful, the aspect stays, otherwise it goes.

7 Future Work

The Java to C compiler and associated run-time system framework is, as of current status, capable of handling most of the Java language, generating semantically correct C code. Apart from the fact that neither the compiler, nor the runtime system and class library, are 100% complete, with regard to the Java specification and the Java Development Kit (JDK), there are many interesting problems to look into.

Optimizations. Generating code that will function properly in all possible executions will result in conservative code, with sometimes unnecessary overhead degrading application performance[3]. We are therefore looking at several ways of enhancing general performance, without sacrificing real-time performance.

There are a number of OO optimization techniques which could be used to increase general performance of an application. To this class of optimizations belong such well-known techniques, see for example [16, 17, 15], as method call de-virtualization and class in-lining.

---

[3] e.g., The wanted sampling rate of a high priority regulator thread can not be accomplished due to Garbage Collect(ion|or) (GC) overhead

Code Analysis. Persson [20], has published work on using the JastAdd tool to implement worst-case memory usage and Worst-Case Execution Time (WCET) analysis on Java applications. His work should be continued and implemented in our Java compiler, not only as an aid to the programmer, but the analysis results should be possible to use in some optimizations.

Hybrid Execution Environment In some situations a hybrid execution model, mixing code executed in a Java Virtual Machine (JVM) with natively compiled code, can be preferred to choosing one of the execution models. Since the IVM [21] uses the same object model as our Java compiler, it should be possible to integrate these execution environments.

8   Conclusions

For the construction of a Java compiler, academic state-of-the-art tools based on RAGs and AOP techniques were used. The compiler was constructed in a modular fashion, with a number of aspects for the JastAdd tool, comprising the normal phases of a compiler; static semantic analysis, optimizations, and code generation.

Having implemented a compiler for a complete modern OO programming language, using the JastAdd tool, we have drawn the following conclusions:

- The OO fashion of the generated AST and the use of semantic equations renders a very compact, yet apprehensible, compiler implementation.
- Code analysis, refactorings, and optimizations, can be conveniently described as aspects performing computations and transformations on an OO AST.
- Although substantially slower to compile Java applications than other Java compilers (javac and gcj), it is still fast enough to build embedded software using standard workstations.

To our knowledge, this is the first compiler for a complete OO programming language built using RAG tools.

9   Acknowledgements

References

1. Hedin, G., Magnusson, E.: The JastAdd system - an aspect-oriented compiler construction system. SCP - Science of Computer Programming **1** (2002) 37–58 Elsevier.

---

[4] http://www.lucas.lth.se

2. Nichols, B., Buttlar, D., Proulx Farrell, J.: Pthreads Programming. 1st edn. O'Reilly (1996) ISBN: 1-56592-115-1.
3. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
4. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. Lecture Notes in Computer Science **2072** (2001) 327–355 *http://eclipse.org/aspectj/.*
5. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2** (1968) 127–145 Published by Springer-Verlag New York Inc.
6. Hedin, G.: Reference attributed grammars. Presented at WAGA99 (1999)
7. Nilsson, A., Ekman, T., Nilsson, K.: Real java for real time – gain and pain. In: Proceedings of CASES-2002, ACM, ACM Press (2002) 304–311 To be presented.
8. Ive, A., Blomdell, A., Ekman, T., Henriksson, R., Nilsson, A., Nilsson, K., Gestegård-Robertz, S.: Garbage collector interface. In: Proceedings of NWPER 2002. (2002)
9. Ekman, T., Hedin, G.: Rewritable Reference Attributed Grammars, Olso (2004) Presented at the 18th European Conference on Object-Oriented Computing (ECOOP) 2004.
10. : (Java-cc parser generator) Metamata Inc. *http://www.metamata.com.*
11. Hudson, S.E., Flannery, F., Anaian, C.S., Wang, D., Appel, A.W.: Cup parser generator for java. *http://www.cs.princeton.edu/ appel/modern/java/CUP/* (1999)
12. Menjíbar, F.: Portable Java compilation for Real-Time Systems. Master's thesis, Dep. of Computer Science Lund University (2003)
13. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. 1st edn. The Java Series. Addison-Wesley (1996)
14. Nilsson, A.: **PhD -> Licentiate** Compiling Java for Real-Time Systems. PhD thesis, Dep. of Computer Science, Lund Institute of Technology, Lund University (2004)
15. Tip, F., Sweeney, P.F., Laffra, C.: Extracting library-based java applications. Communications of the ACM **46** (2003) 35–40
16. Arnold, M., Hind, M., Ryder, B.G.: An empirical study of selective optimization. In: Proceedings of the International Workshop on Languages and Compilers for Parallel Computing. (LCPC '00). (2000)
17. Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B., Tarditi, D.: Marmot: An optimizing compiler for java. Technical report, Microsoft Research, 1 Microsoft Way Redmond, WA 98052 (1999)
18. Henriksson, D., Cervin, A., Åkesson, J., Årzén, K.E.: On Dynamic Real-Time Scheduling of Model Predictive Controllers. In: Proceedings of the 41st IEEE Conference on Decision and Control, Las Vegas, Nevada (2002)
19. Nilsson, K., Blomdell, A., Laurin, O.: Open Embedded Control. Real-Time Systems **14** (1998) 325–343
20. Persson, P.: Predicting time and memory demands of object-oriented programs. Licentiate thesis, Department of Computer Science, Lund Institute of Technology (2000)
21. Ive, A.: Towards an embedded real-time java virtual machine. Licentiate thesis, Department of Computer Science, Lund Institute of Technology (2003)

A Simplification Transforms

JastAdd aspects needed for transforming name usage in Java from arbitrarily complex to the simplest form.

```
class Simplify {
  void Stmt.simplify () {
    if (stmt.needsRewrite ()) {
      setStmt(stmt.rewrite(),stmtIndex);
    }
  }
  syn boolean Stmt.needsRewrite = false;
  syn boolean Expr.needsRewrite = needsRewrite (0);
  ExprStmt.needsRewrite = getExpr().needsRewrite ();
  boolean AssignExpr.needsRewrite {
    return getSource.needsRewrite (0) ||
                          getDest.needsRewrite (0);
  }
  boolean Access.needsRewrite (int level) {
    return nbrOfDeref () > level;
  }
  syn int Expr.nbrOfDeref = 0;
  VarAccess.nbrOfDeref = 1+getEnv().nbrOfDeref ();

  void AssignExpr.rewrite(List l) {
    int sLevel=0,dLevel=1;
    VariableDeclaration varDecl = createTempVar(type());
    l.add(varDecl);
    Expr source = getSource().rewrite(l,sLevel);
    Expr dest = getDest().rewrite(l,dLevel);
    l.add(new ExprStmt(new AssignSimpleExpr(
                        accessVar(varDecl),dest)));
    l.add(new ExprStmt(getSpecialAssignExpr(
                        accessVar(varDecl),source)));
    l.add(new ExprStmt(new AssignSimpleExpr(
                        dest,accessVar(varDecl))));
  }

  Expr Access.rewrite(List l, int level) {
    if (nbrOfDeref () > level) {
      Expr e = getEnv().rewrite(l,0);
      if (level == 0) {
        VariableDeclaration varDecl = createTempVar(type());
        setEnv(e);
        l.add(varDecl);
        l.add(new ExprStmt(
              new AssignSimpleExpr(accessVar(varDecl),this)));
        return accessVar(varDecl);
      } else { setEnv(e); }
    }
```

```
        return this;
    }
}
```