Master’s Thesis

Implementation of Java 7 Features in an Extensible Compiler

Jesper Öqvist

Department of Computer Science
Faculty of Engineering LTH
Lund University, 2012

ISSN 1650-2884
LU-CS-EX: 2012-13

Implementation of Java 7 Features in an Extensible Compiler

Jesper Öqvist

August 30, 2012

Abstract

Java 7 introduced new features to the Java Programming Language, e.g., the Try-With-Resources statement, Strings in Switch and Type Inference for Class Instance Expressions.

The new features are small improvements of existing Java features, yet they impact many aspects of the compiler, from scanning and parsing to type analysis and code generation.

This thesis describes the implementation of the Java 7 features using the declarative metacompiler tool *JastAdd*. We show how declarative features of JastAdd, such as nonterminal and reference attributes can be used to implement these features as modules extending *JastAddJ*, an existing Java 6 compiler. The resulting Java 7 compiler is evaluated concerning code size, modularity, compilation time, and memory use during compilation.

Acknowledgements

I wish to thank my supervisor Görel Hedin for guidance during the writing of this thesis. I would also like to thank the following people for valuable feedback or discussions concerning my thesis work: Niklas Fors, Emma Söderberg, and Linus Åkesson. For their opposition of my thesis I thank Snild Dolkow and Jimmy Pettersson. Finally, for making me explain my work in simpler terms, for motivation, and much distraction, thank you Elise.

Contents

1 Introduction **5**

2 Background **6**
2.1 Java 7 Language Features 7
2.2 JastAddJ 7
2.3 JastAdd 8
2.3.1 The Abstract Syntax Tree 9
2.4 Attribute Grammars 11
2.4.1 Synthesized Attributes 12
2.4.2 Inherited Attributes 13
2.4.3 Nonterminal Attributes 15
2.4.4 Rewrites 16
2.5 Generation Architecture 18
2.5.1 Scanner and Parser 19

3 Implementation Process **20**
3.1 Testing 21

4 Implementation of Java 7 Language Features **22**
4.1 Try-With-Resources Statement 23
4.1.1 TWR: Specification 24
4.1.2 TWR: High-Level Design 26
4.1.3 TWR: Implementation Details 30
4.1.4 TWR: Delegated Code Generation 31
4.2 Strings in Switch 34
4.2.1 Strings in Switch: Specification 34
4.2.2 Strings in Switch: High-Level Design 36
4.2.3 Strings in Switch: Implementation Details 36
4.3 Diamond 39
4.3.1 Diamond: Specification 40
4.3.2 Inference of Method Type Arguments 40
4.3.3 Diamond: High-Level Design 41
4.3.4 Diamond: Implementation Details 43
4.4 Improved Numeric Literals 44
4.4.1 Improved Numeric Literals: Specification 44
4.4.2 Improved Numeric Literals: High-Level Design 44

4.4.3 Improved Numeric Literals: Implementation Details . . . 45
4.5 Multi-Catch 46
4.5.1 Multi-Catch: Specification 47
4.5.2 Multi-Catch: Implementation Details 47
4.5.3 Multi-Catch: Alternative Implementation 48
4.6 More Precise Rethrow 48
4.6.1 More Precise Rethrow: Specification 49
4.6.2 More Precise Rethrow: High-Level Design 50
4.6.3 More Precise Rethrow: Implementation Details 50
4.7 Safe Varargs 50
4.7.1 Safe Varargs: Specification 50
4.7.2 Safe Varargs: High-Level Design 51

5 Discussion **51**
5.1 Combining Modules 52
5.2 Problems Encountered 52
5.2.1 Bugs in JastAddJ 53
5.2.2 Problems With Beaver 53

6 Evaluation **54**
6.1 Implementation Size 54
6.2 Benchmark Programs 55
6.3 Measurement Details 55
6.4 Benchmark Results 56

7 Conclusion **58**
7.1 Future Work 59

Appendices **60**

A Obtaining JastAddJ **60**

B Strings-in-Switch Tests **60**

1 Introduction

The Java programming language is continuously evolving. In 2011, sixteen years since JDK 1.0 was released by Sun Microsystems, there have been eight major versions of Java released. Usually several versions of the Java language are supported concurrently and, by extension, the corresponding versions of the reference compiler require maintenance.

The reference compiler for the Java language, *javac*, is updated for each new version of the programming language through iterative modification of the previous version of *javac*. This development method leads to code duplication between the different versions of the compiler, so when a new bug is discovered the fix must be added in each maintained version of the compiler that uses the same features, since the implementation at least started with an identical code base and thus likely suffers from the same bug.

In this thesis, we investigate a different approach to compiler development for a growing language: an extensible base compiler is developed in such a way that it allows new features to be added using self-contained extensions. In theory this approach could eliminate the need to modify the code for the base compiler in order to support new language features, and would limit the code duplication needed to create a new compiler configuration. Additionally, a bug fixed in the base compiler would immediately become fixed in all other configurations without changes to the extensions.

Our approach requires a system of modularization that permits extensions to augment large parts of the existing compiler logic without the need for changes in the base compiler. The system used in this thesis for modular compiler development is named *JastAdd*, and using this system I have developed an extension to the JastAdd Extensible Java Compiler *JastAddJ* [3].

The purpose of my thesis work was both to further explore the extensibility of compilers developed using the JastAdd system, and to add support for the latest version of the Java language, Java 7, to the JastAddJ compiler. I wish to answer the following questions in my thesis, regarding the implementation of Java 7 support as an extension to JastAddJ:

Question 1 To which extent can the features of Java 7 be individually modularized in the JastAddJ implementation?

Question 2 Is the implementation correct?

Question 3 How efficient is the implementation, with respect to

- code size
- compilation time
- memory usage

Section 2 covers the Java 7 language features, the JastAdd system and other tools used to generate the JastAddJ compiler.

Section 3 details the process used to implement the Java 7 features in JastAddJ, and in section 4 the implementation of each feature is examined.

A discussion of the completed implementation can be found in section 5 where I attempt to answer questions one and two above.

In section 6, the code size, compilation time and memory usage of JastAddJ is compared with that of javac in order to answer question three above.

Finally, section 7 wraps up with a conclusion of the lessons learned in this thesis, and the answers to the three main questions.

2 Background

On July 28, 2011 the latest major update to the Java programming language, Java SE 7, was released [11]. Notable earlier versions of the language include J2SE 1.4 (2002), J2SE 5.0 (2004) and Java SE 6 (2006). Informally, these versions are referred to as Java 1.4, 5, 6 and 7.

Each new version of Java has included language changes and changes to the Java class library. The new language features introduced in Java 7 are discussed in subsection 2.1.

Section 2.2 introduces the JastAdd Extensible Java compiler, and in subsection 2.3 we discuss the JastAdd system which was used to build this compiler.

Section 2.4 contains an introduction to attribute grammars. Section 2.5 illustrates how JastAddJ is built using the attribute grammar and inter-type declaration features of JastAdd.

2.1 Java 7 Language Features

The language changes introduced in Java 7 are specified in Java Specification Request 334 [12]. These include new syntax elements as well as semantic changes to pre-existing statements or constructs in the Java language. We have divided the changes into the following distinct features:

- Try-With-Resources
- Strings in Switch
- Diamond
- Improved Numeric Literals
- Multi-Catch
- More Precise Rethrow
- Safe Varargs

The implementation of each of these features is discussed in section 4.

2.2 JastAddJ

A compiler is a computer program which translates source files into executable binary files to be run on a specific machine. In the Java environment the compiler translates Java source files into binary *class files*, which are executed on the Java Virtual Machine (JVM). The machine instructions for the JVM are referred to as *bytecode* [10].

The JastAdd Extensible Java Compiler (JastAddJ) [3] is a Java compiler that was developed for computer language research. JastAddJ originally supported Java 1.4. Support for Java 5 was added as an extension in 2005.

JastAddJ consists of separate front-end and back-end modules. The front-end modules handle parsing and semantic error-checking of Java source files or class files. Each front-end module has a sibling back-end module that handles the bytecode generation.

The six modules of JastAddJ at the time of writing are are listed in figure 1. The Java7Frontend and Java7Backend modules constitute the Java 7 extension to JastAddJ (JJ7).

Java Version	*Module Name*
Java 1.4	Java1.4Frontend Java1.4Backend
Java 5	Java1.5Frontend Java1.5Backend
Java 7	Java7Frontend Java7Backend

Figure 1: The modules of JastAddJ.

2.3 JastAdd

JastAdd is a tool for compiler construction which generates Java source files for a compiler using reference attribute grammars (RAGs) and inter-type declarations contained in aspect files [6]. As we will demonstrate in this thesis, the JastAdd system supports modular compiler construction.

JastAdd compilers analyse and transform programs using the Abstract Syntax Tree (AST) – a representation of the syntactic structure of a source file. The AST is built by a scanner and parser. However, since JastAdd does not generate the scanner or parser they need to be built either by hand or using other tools. Apart from the scanner and parser, all other parts of the compiler can be generated by JastAdd.

JastAdd generates a Java class hierarchy representing the node types of the AST. Attributes, types and methods are woven into the generated Java classes using declarations in aspect files.

Aspects are used to group semantically related attributes and declarations in a way that is not constrained by the Java classes these declarations are woven into. These so-called inter-type declarations are useful in compiler design as all declarations concerning a particular analysis in the compiler, e.g. type analysis, can be grouped into the same aspect file regardless of which Java classes the declarations affect [17].

JastAdd allows a mix of declarative and imperative programming styles — declarative attributes and equations as well as imperative methods and fields are woven into the AST types by JastAdd. The generated Java code will typically not have to be modified as every necessary method, field or class can be added using aspect declarations.

The next section discusses the node types generated by JastAdd, and how these are used to build the AST. Section 2.4 contains an introduction to the attribute grammar features used in JastAdd.

2.3.1 The Abstract Syntax Tree

The AST is a tree structure containing nodes that represent syntactical constructs in a source file. JastAdd builds a Java class hierarchy to represent the nodes of the AST.

Java's polymorphism is used to inherit attributes and methods between node types — similarities between types can be extracted into a common supertype. For example, JastAddJ uses a `Binary` node type to represent all binary expressions. Some operations on binary expressions have been generalised so that only those parts that are not common between all binary expressions have to be specialized in concrete subtypes.

JastAdd generates at least three implicitly defined node types. These default types are seen in figure 2. `ASTNode` is the base node type which all other types inherit from. `List` contains a list of zero or more nodes, and `Opt` is an *optional* node that contains either one or zero nodes.

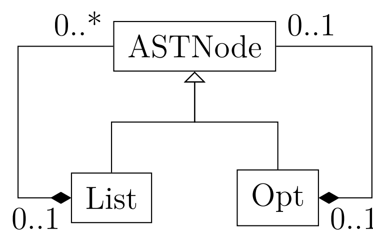


Figure 2: Default node types in the JastAdd-generated AST.

AST Declaration Syntax Custom node types are specified in AST declaration files (with the file name extension `.ast`, cf. figure 9a) as a list of declarations in the following format:

```
Node [: Parent] [::= Children];
```

The parts within brackets are optional; `Node` is the name of the declared node type, `Parent` is the name of the parent type that it extends. If no parent is specified, it extends `ASTNode`. `Children` is a list of child declarations. The declared node type will inherit all children from its parent type and may add additional children.

A child declaration has one of the following forms:

`T` — A child of type `T`. It's name is "T".

`N:T` — A child of type `T`. It's name is "N".

T* — A child of type List, containing zero or more T nodes.

[T] — A child of type Opt, containing zero or one T nodes.

<N> — A token of type String. It's name is "N".

<N:T> — A token of reference type T named "N".

/T/ — A nonterminal attribute of type T named "T".

Syntax Example The following JastAdd declarations build the node types in figure 3:

```
Expr;
Literal : Expr;
IntegerLiteral : Literal ::= <Value:Integer>;
Binary : Expr ::= Left:Expr Right:Expr;
AddExpr : Binary;
SubExpr : Binary;
```

Note that the integer value of an IntegerLiteral is stored as an Integer token. In JastAddJ tokens are used to store names (of variables, classes, methods, parameters etc.), and integer- and string literals.

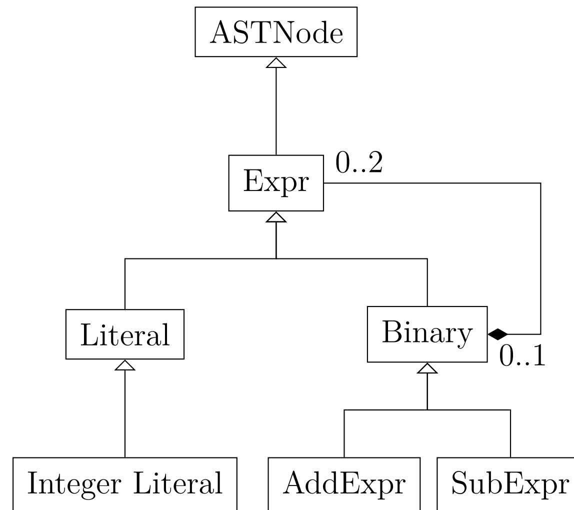


Figure 3: Example AST node type hierarchy (Opt and List nodes omitted).

AST Example Figure 4 displays an AST fragment representing the Java expression $(3 + 6 - 7)$, using the node types declared above. The names of children in the AST fragment are marked on the edges between parent and child nodes, and the name of each node's type is displayed on the node. Nonterminal attributes (not shown in this figure) are marked using a dashed line connecting the nonterminal attribute to the rest of the AST. The grammar and node types used here are a simplified version of those used in JastAddJ.

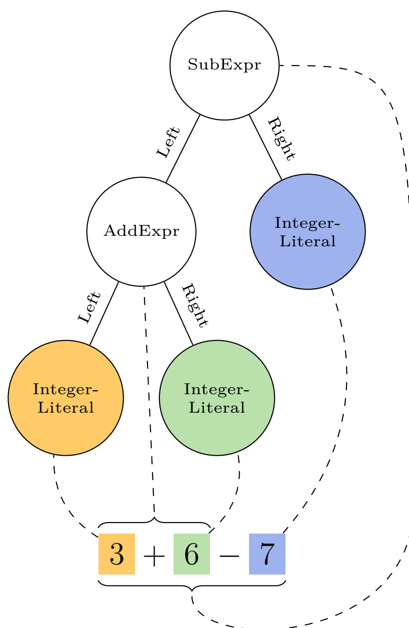


Figure 4: An AST fragment representing the Java expression $(3 + 6 - 7)$.

2.4 Attribute Grammars

Attribute grammars (AGs) provide a nice declarative way of performing semantic computations in an AST [9]. The two basic kinds of attributes supported by JastAdd are *synthesized* and *inherited* attributes, discussed in subsubsection 2.4.1 and 2.4.2 respectively. Synthesized attributes propagate information upwards in the AST, while inherited attributes propagate information downward.

JastAdd supports an extension of AGs called reference attribute grammars (RAGs) which allows attributes to be references to other AST nodes [6].

JastAdd generates attribute evaluator methods for each attribute and checks, e.g., that there is an equation for each attribute in every possible AST. Since the programmer who implements the attribute can not and should not

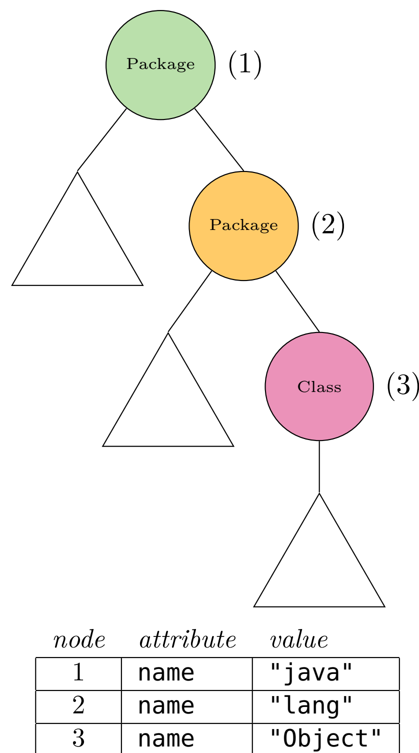
have to know the order of attribute evaluation, it is important that they are side-effect free. Working with the assumption that the attributes are side-effect free, JastAdd may decide to cache attribute values in order to avoid redundant computations.

Besides synthesized and inherited attributes, JastAdd supports *non-terminal attributes* (NTA) [16]. Nonterminal attributes are discussed in subsubsection 2.4.3.

2.4.1 Synthesized Attributes

Figure 5 illustrates the computed values of the synthesized attribute `name` in an AST fragment. This attribute computes the name of a `Class` or `Package` node. The value of the attribute is computed in the same node that the attribute is declared in, and the attribute may be accessed from the node itself or a parent node — thus propagating the attribute value upwards in the tree. Let's assume the `Package` and `Class` node types each have a `Name` token that contains the parsed name of the node, the `name` attribute can then be implemented with the following JastAdd code:

```
syn String Package.name() = getName();
syn String Class.name() = getName();
```



node	*attribute*	*value*
1	`name`	`"java"`
2	`name`	`"lang"`
3	`name`	`"Object"`

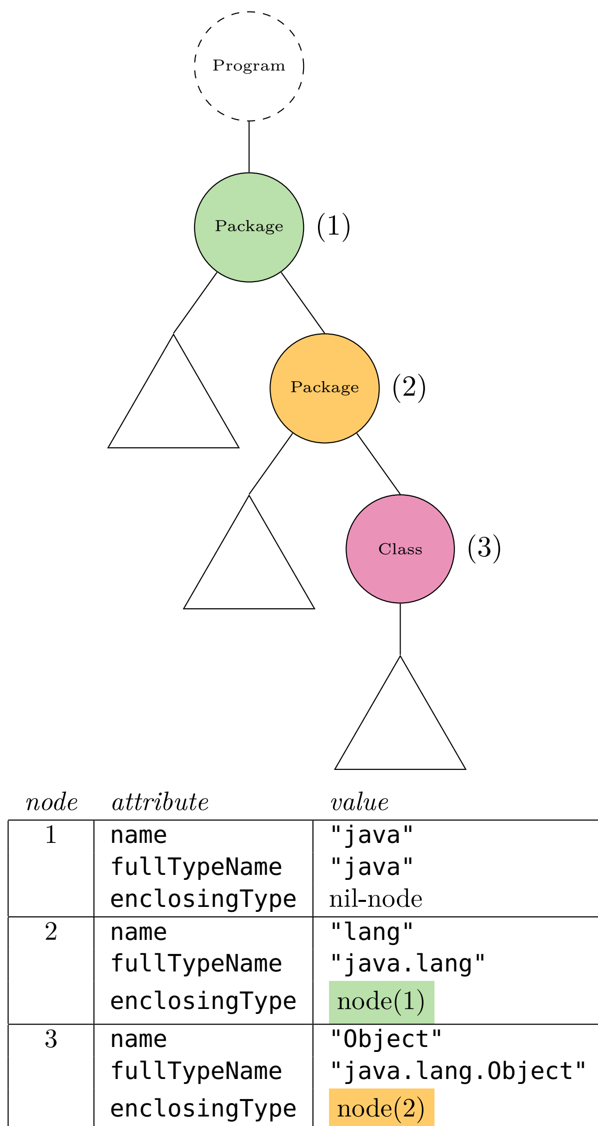
Figure 5: A synthesized attribute that computes node names. The value of the attribute for each numbered node is given in the table under the AST fragment.

2.4.2 Inherited Attributes

If we wish to compute the full type name of a class it is not enough to simply know the name of the class. We need to know the names of all enclosing classes and packages, if any. However, as we have seen in the previous example, synthesized attributes only propagate information upwards in the AST. Now we wish to access information (enclosing class/package name) in AST ancestor nodes. To accomplish this we can use inherited attributes to pass the enclosing type or package name downward through the AST.

The difference between inherited and synthesized attributes is the context in which the attribute is computed. An inherited attribute is computed in an ancestor node. The ancestor that computes the attribute is the first one that has a declared equation for the attribute. JastAdd traverses the AST upwards until it finds an equation for the attribute, then evaluates that equation. JastAdd also ensures that there is always at least one equation for each inherited attribute in an ancestor node type, for every possible AST.

Figure 6 illustrates the use of inherited attributes in the same mock AST fragment as in figure 5, with one difference: a `Program` node has been added, since this node is required to compute the inherited `enclosingType` attribute for the top-level `Package` node. Again, the computed value of the attributes is given for each numbered node in the table beneath the AST fragment. The `name` attribute has no dependencies, however the `fullTypeName` attribute depends on the `name` and `enclosingType` attributes. The `enclosingType` attribute depends only on the first ancestor which has an equation for `enclosingType`.



node	*attribute*	*value*
1	`name`	`"java"`
	`fullTypeName`	`"java"`
	`enclosingType`	nil-node
2	`name`	`"lang"`
	`fullTypeName`	`"java.lang"`
	`enclosingType`	node(1)
3	`name`	`"Object"`
	`fullTypeName`	`"java.lang.Object"`
	`enclosingType`	node(2)

Figure 6: Inherited attributes to compute full type names.

The equations for the new attributes `fullTypeName` and `enclosingType` are given in pseudo-code below:

```
inh ASTNode Package.enclosingType();
inh ASTNode Class.enclosingType();
eq Program.getChild(int index).enclosingType() = nilNode();
eq Package.getChild(int index).enclosingType() = this;
eq Class.getChild(int index).enclosingType() = this;

syn String Package.fullTypeName() {
    if (enclosingType() != nilNode())
        return enclosingType().name()+name();
    else
        return name();
}
syn String Class.fullTypeName() {
    if (enclosingType() != nilNode())
        return enclosingType().name()+name();
    else
        return name();
}
```

2.4.3 Nonterminal Attributes

Nonterminal Attributes (NTAs), also called higher-order attributes [16], are attributes whose values are nonterminals — i.e. AST nodes. An NTA is quite different from a regular synthesized or inherited attribute in that the "value" of the attribute is only computed once, regardless of JastAdd's caching preference. The produced node is then rooted at the node that computed the nonterminal attribute.

NTAs are useful to synthesize constructions in the program that were not literally present in the parsed source code. For example, during type analysis, JastAddJ requires a `TypeDecl` node to represent array types. There is no matching declaration to be parsed, since array types are implicitly declared in Java. Instead, an NTA is used to create a placeholder array type declaration.

To illustrate NTAs in AST diagrams we will use a dashed line to connect the NTA node to its parent in the AST. The NTA name is displayed just like other node names, on the line connecting the node to its parent, but is written in italics. Figure 7 below illustrates the *Constant* NTA, added to the `SubExpr` node type:

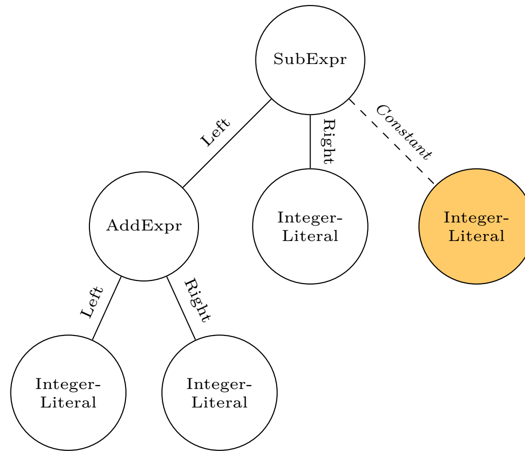


Figure 7: Nonterminal attribute

2.4.4 Rewrites

The AST can be transformed automatically by JastAdd through a mechanism called rewrites. Rewrites are a kind of term rewriting system [8] which allow nodes (terms) in the AST to be transformed under given conditions.

In JastAdd, rewrites are specified using so-called *rewrite rules*. Each rewrite rule may be unconditional, or conditional. Conditional rewrites use a boolean expression to determine if the node should be rewritten. A rewrite creates a fresh AST node, either from scratch, or using copied descendants from the original node. The following figures will use a dashed arrow to indicate which nodes have been copied from the original node. Since the new node replaces the rewritten node, the originals of the copied descendants are discarded. The copying of descendants is in effect equivalent to moving them to the new AST fragment.

Rewrites are triggered by the regular child access operations — the programmer does not manually trigger the rewrite. The rewrites are applied transparently — the child access operation will return the transformed child node.

Rewrites are useful for a number of common tasks; e.g. constant propagation or transforming equivalent expression forms into a uniform structure. In general, a rewrite should not affect the semantics of the transformed AST fragments.

Example: Constant Propagation The Java expression $3+5+6$ is a constant expression. It can safely be replaced by the constant value 14. A rewrite rule for this transformation might look like this:

```
rewrite Expr {
    when (isConstant()) {
        return new Constant(constantValue());
    }
}
```

Example: Multiple Declaration Flattening Java allows multiple declarations — declaring several variables of the same type in one declaration statement:

```
int index, i, j;
```

Let's say a compiler represents any variable declaration with a `MultiDecl` node, even if it is a single declaration. The `MultiDecl` node contains a list of declared variable names. This can be flattened to a list of `SingleDecl` nodes, which more tightly couples the declared { variable, field, parameter } name with its type. Figures 8a and 8b illustrate the flattening of two `MultiDecl` nodes.

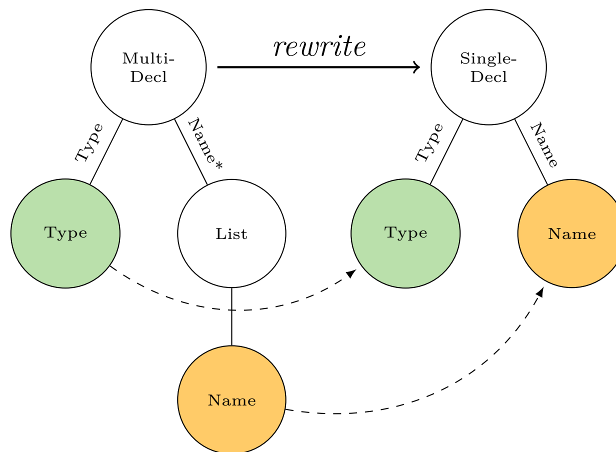


Figure 8a: Flattening of declaration statements: single declaration case.

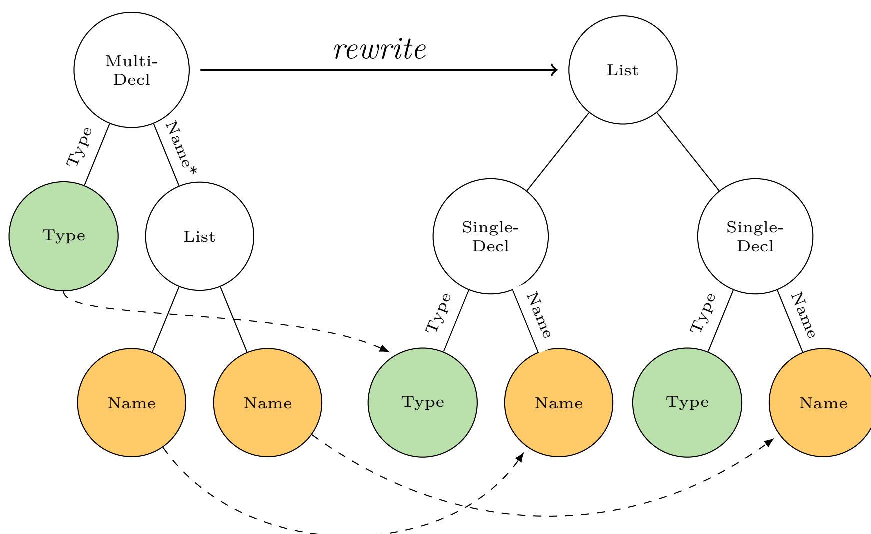


Figure 8b: Flattening of declaration statements: multiple declaration case.

2.5 Generation Architecture

Figure 9a illustrates the build pipeline of JastAddJ's AST Java class package. Most of the compiler logic is embedded in the Java classes representing the node types. JastAdd builds the node types from the AST definitions in `.ast` files from each module, then weaves attributes, rewrites, fields, and methods into the generated classes.

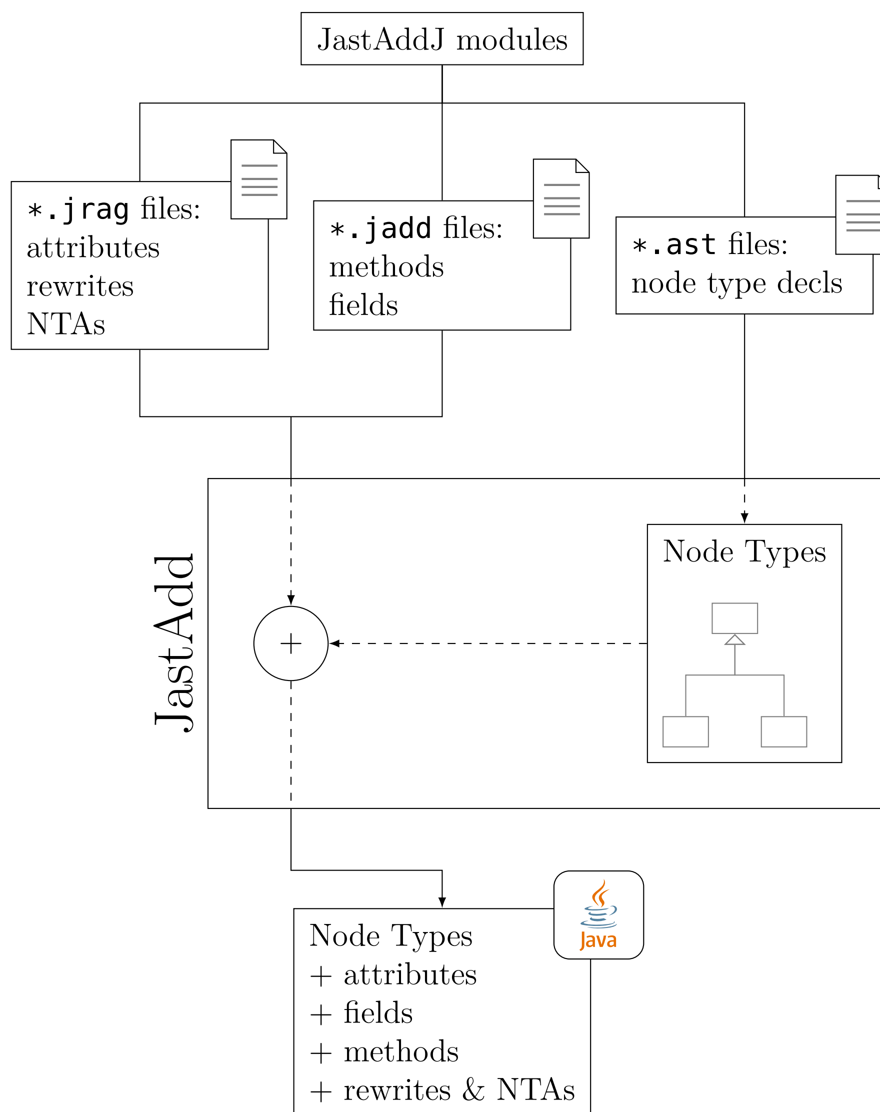


Figure 9a: The build pipeline of the JastAddJ AST package.

2.5.1 Scanner and Parser

Building a compiler with JastAdd requires some external tools to parse the Java source files and build the AST. For JastAddJ, the parser is generated by the Beaver parser generator [2], and the scanner is generated by JFlex [7].

The *JastAddParser* program is used to pre-process the parser specifications. JastAddParser combines a set of parser grammar productions into one parser specification that can be used with the Beaver parser generator. This allows the JastAddJ parser specifications to be somewhat modular — new

productions can be added to the parser grammar by JastAddJ extensions in a way that would otherwise not have been possible.

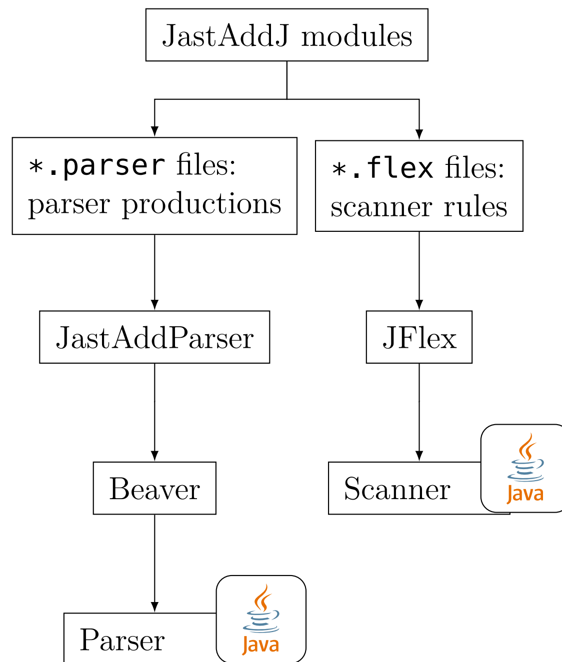


Figure 9b: The build pipeline of JastAddJ's scanner and parser.

3 Implementation Process

We had the following goals for the implementation of JJ7:

- The new features should be easy to extend
- The implementation should be reasonably simple to understand for new developers
- The implementation should be well documented
- The implementation should be complete
- The implementation should be correct

The implementation process used test cases to test the completeness and correctness of JJ7. The implementation of the Java 7 language features was specification-driven; initial test cases were derived directly from the specification, and the implementation was modularized with one module per feature.

Most of the initial test cases were compilation tests, which tested that the compiler accepted the general syntax variations of the new language features. Later, some execution tests and more advanced compilation tests were added to test that the compiled programs executed correctly and to test possible implementation edge cases. Section 3.1 covers the test cases used in greater detail.

During the development, the following steps were taken for each feature:

Testing Tests cases were created based on the preliminary documentation of the Java 7 features (see subsection 2.1).

Implementation A first implementation of the feature was written to satisfy the test cases.

Refactoring Refactoring opportunities were used to refine the implementation. Tests for bugs discovered during the implementation were added to the test suite.

Documentation Documentation comments were written for the new attributes and methods.

For some features, new bugs or missing features were discovered during the documentation step, which lead to the addition of new tests, reimplementation, refactorings and new documentation.

3.1 Testing

Extensive testing is a valuable tool in software development in general, and compiler design is no exception. A high quality test suite helps ensure the correctness of the compiler, and allows developers to refactor the compiler or implement new features with increased confidence that the features tested by the test suite have not broken.

Ideally, any errors introduced during development should be caught by the test suite. Unfortunately, compilers are complex software systems and it is often not possible in practice to create a test suite that would catch all possible compiler errors.

Due to the perceived importance of testing for the development of JJ7, tests were the first concern addressed during the implementation. Unfortunately, it was not feasible to obtain a license to use the official test suite for Java compliance, known as the JCK, owned by Oracle.

Instead of using the JCK test suite, it was decided that new tests would be written for JJ7. Two different frameworks were used for these tests; *Jacks*

and *JUnit*. Jacks is a framework *and* a test suite that was originally developed by IBM and was the regression test suite of the Jikes project [13]. Since then, Jacks has become part of the Open-Source Mauve project [14]. JUnit is a well known unit testing framework for Java written by Kent Beck [4].

These two test frameworks complement each other — Jacks is used to test that compilation passes or fails, while JUnit tests the execution of compiled programs.

A custom test suite was developed as part of this thesis work. It contains 128 compilation tests and 36 execution tests. Of the 128 compilation tests, 6 are regression tests for bugs that were fixed in the Java 1.4 or 5 versions of JastAddJ during the development of JJ7.

All tests were run with the Java 7 reference compiler to confirm that the tests were correct.

4 Implementation of Java 7 Language Features

The implementation of each feature listed in subsection 2.1 is discussed one at a time in the following sections, focusing on three aspects:

Specification A summary of the syntax, semantics, and possible corner cases of the feature.

High-Level Design An overview of the approach used to implement the feature.

Implementation Details Details of how the feature was implemented.

Figure 10 lists the parts of JastAddJ that were affected by each feature:

	Feature	*Scanner*	*Parser*	*Front-end*	*Back-end*
4.1	Try-With-Resources		✓	✓	✓
4.2	Strings in Switch			✓	✓
4.3	Diamond		✓	✓	
4.4	Improved Numeric Literals	✓	✓	✓	
4.5	Multi-catch		✓	✓	✓
4.6	More Precise Rethrow			✓	
4.7	Safe Varargs			✓	

Figure 10: Scope of each feature

4.1 Try-With-Resources Statement

The `try` statement has been an essential part of the Java language since its inception. Any *checked* exception[1] which may be thrown by some statement *must* be caught and handled by a surrounding `try` statement with a matching `catch` clause, or be declared to be thrown by the enclosing method.

A `try` statement can also be used as a simple control flow harness — if the optional `finally` block is present, then any control-flow path that passes through the corresponding `try`-statement *must* also pass through the `finally` block.

A common use case for the `try` statement is the handling of exceptions thrown by an operation on a resource. Consider the following Java 6 example:

```
Socket socket = null;
try {
    socket = new Socket("elise", 1080);
    OutputStream out = socket.getOutputStream();
    out.write("Life can be hilariously cruel.".getBytes());
} catch (IOException e) {
    // report the error and perform recovery
} finally {
    if (socket != null) {
        try {
            socket.close();
        } catch (IOException e1) {
        }
    }
}
```

The *resource* here is the `socket` variable, which holds a reference to a socket that we connect to a server. We want to send a message to this server, but both socket creation and the sending of the message can potentially fail and throw an exception — if e.g. the server is unreachable the call to the constructor will throw an `IOException` and if the server suddenly disconnects after the connection has been established then the call to `write` will throw an `IOException`.

If an exception is thrown after the socket has been connected we wish to disconnect it, in order to not leave a dead socket hanging around. The

[1] Java has both *checked* and *unchecked* exceptions (`RuntimeException`, `Error` and their subclasses); the unchecked exceptions need not be caught by the programmer but will interrupt the thread if uncaught.

`finally` block meets this need of closing the socket, since we want to close the socket regardless of whether there was an exception or not. However, notice that we must check that the socket was actually created (not `null`) before we call `close`.

The `try`-with-resources (TWR) statement from Java 7 is a new variant of the `try` statement that simplifies the above use case. Using a TWR statement, the previous example could look like this:

```
try ( Socket socket = new Socket("elise", 1080) ) {
    OutputStream out = socket.getOutputStream();
    out.write("Life can be hilariously cruel.".getBytes());
} catch (IOException e) {
    // report the error and perform recovery
}
```

The TWR statement takes care of closing the resource (`socket`), as if the `finally` block was still there.

This new form of the `try` statement allows the declaration of one or more resources at the start of the `try` statement to be used as local variables within the `try` block and which are automatically closed when execution reaches the end of the `try` block.

4.1.1 TWR: Specification

The TWR statement starts with a list of resource declarations, within parentheses, before the body of the `try` block. Each resource must be a subtype of `AutoCloseable`, a new interface introduced in the Java 7 class library which declares the `close` method used for auto-closing the resources.

It is possible to declare several resources in the declaration part of the TWR. The resource declarations must be semicolon-separated, and an optional trailing semicolon is permitted:

```
try ( AutoCloseable r1 = newResource();
    AutoCloseable r2 = newResource(); ) {
}
```

If a checked exception can be raised by any of the resource initialization expressions, or by the `close` method of an initialized resource, it must be handled in the enclosing method or initializer, or be declared as being thrown.

A TWR statement may have a list of `catch` clauses to handle exceptions raised within the body of the TWR statement, by any of the resource initializations, or by the automatic closing of any of the associated resources:

```
try ( AutoCloseable resource = mayThrowE1(); ) {
    throw new E2();
} catch ( E1 e ) {
    // handle exception thrown by
    // resource initialization
} catch ( E2 e ) {
    // handle exception thrown from
    // the body of the TWR block
} catch ( E3 e ) {
    // handle exception thrown by
    // automatic closing of the resource
}
```

There may also be a `finally` block at the very end of the TWR statement. Its semantics are the same as if used in an ordinary `try` statement — the `finally` block is always executed, regardless of which exceptions are thrown and/or caught.

Any exception thrown by the body of a TWR statement is called the *primary* exception of that TWR. When a primary exception is thrown the automatic closing of resources can cause a new exception, called the *suppressed* exception, to be thrown. The suppressed exception is added to the primary's list of suppressed exceptions[2].

[2]The suppressed exception list was added to the class `Throwable` in the Java 7 class library

4.1.2 TWR: High-Level Design

The TWR statement can be described through a desugaring process that translates any TWR statement to a set of nested `try` statements using intermediate *basic* TWR statements. A basic TWR statement is a TWR statement with only one resource declaration and no `catch` or `finally` clauses. The basic TWR does not handle resource initialization exceptions.

The desugaring process consists of three steps:

1. Eliminate `catch` and `finally` clauses, if any are present, by enclosing the TWR statement in a surrounding `try`- statement and moving the `catch` and/or `finally` clauses to it. See figure 11.

2. Unroll the TWR statement into a nested set of basic TWR statements. This process is illustrated in figure 12.

3. Transform each basic TWR into a regular `try` statement according to the template shown in figure 13.

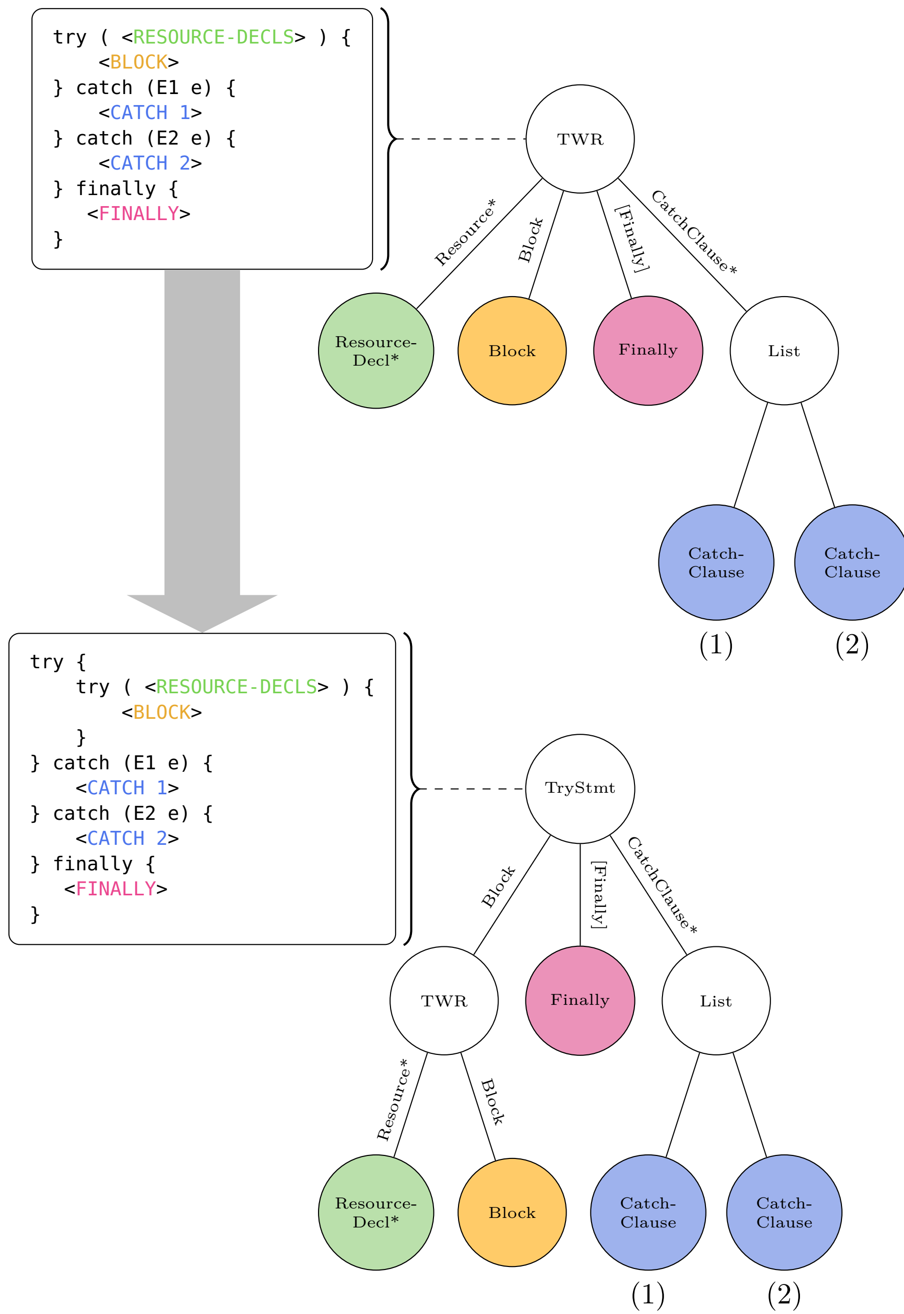


Figure 11: The `catch` and `finally` clauses are moved out to an enclosing `try` statement.

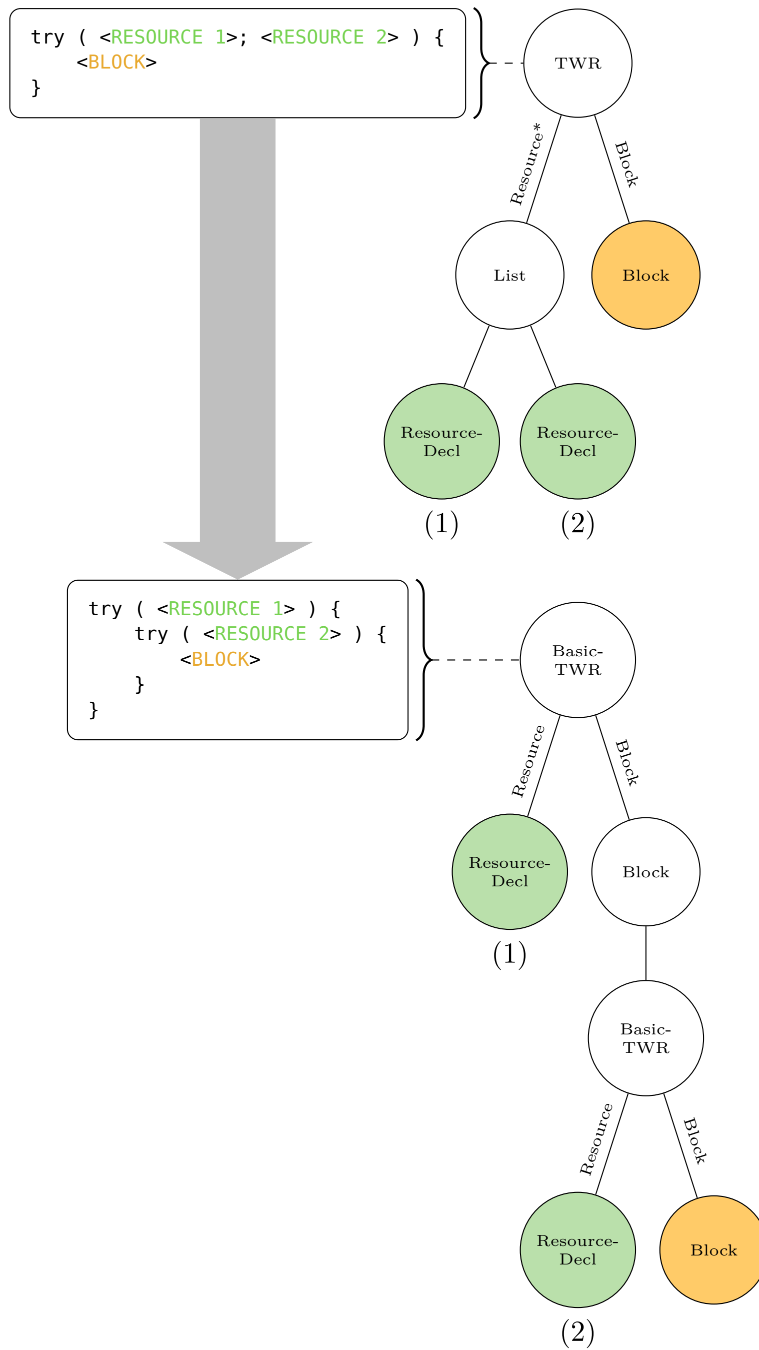
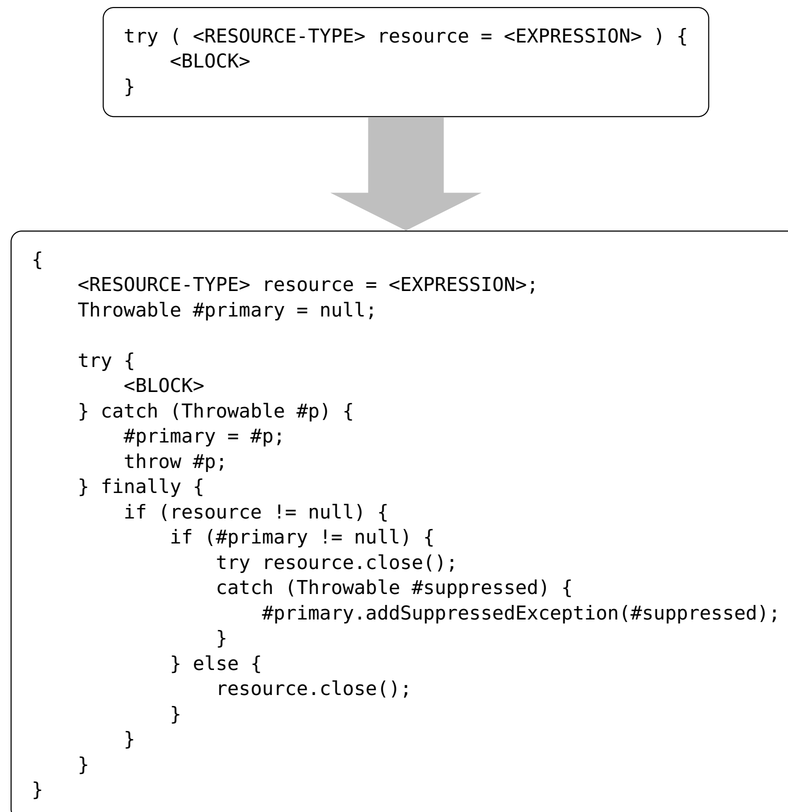


Figure 12: A TWR statement without `catch` or `finally` clauses is unrolled to form a nested set of basic TWR statements.



```
try ( <RESOURCE-TYPE> resource = <EXPRESSION> ) {
    <BLOCK>
}
```

```
{
    <RESOURCE-TYPE> resource = <EXPRESSION>;
    Throwable #primary = null;

    try {
        <BLOCK>
    } catch (Throwable #p) {
        #primary = #p;
        throw #p;
    } finally {
        if (resource != null) {
            if (#primary != null) {
                try resource.close();
                catch (Throwable #suppressed) {
                    #primary.addSuppressedException(#suppressed);
                }
            } else {
                resource.close();
            }
        }
    }
}
```

Figure 13: The transformation of a basic TWR statement into a regular `try` statement, illustrated in pseudo code.

The implementation of TWR affected both the front-end and back-end of JastAddJ, as new parser productions, code generation and static semantic error handling was needed.

A list of the needed changes follows:

- New parser productions in order to parse the TWR syntax.
- New AST classes to represent TWR and basic TWR statements.
- Type checking for resource declarations to ensure that the resources are subtypes of `AutoCloseable`.
- Exception and reachability error checking.
- Code generation for the basic TWR statement: automatic closing of resource and exception handling.

4.1.3 TWR: Implementation Details

A couple of new productions were added to the parser grammar to accept the different forms of TWR, and the following node types were added to the AST definitions:

`TryWithResources` — try-with-resources statement.

`ResourceDeclaration` — new type of declaration node to handle the resource declarations of a TWR statement.

`ResourceModifiers` — specialization of the ordinary `Modifiers` node type which is implicitly `final`.

`BasicTWR` — basic TWR statement.

Much of the exception checking from the Java 5 `TryStmt` node type could be reused for `TryWithResources`. The exception checking analysis in JastAddJ checks whether an exception is properly caught or declared to be thrown. Exception checking also ties in to the reachability analysis as the reachability of a `catch` clause depends on whether there is actually an appropriate exception thrown within its corresponding `try` block that is not caught by a previous `catch` clause.

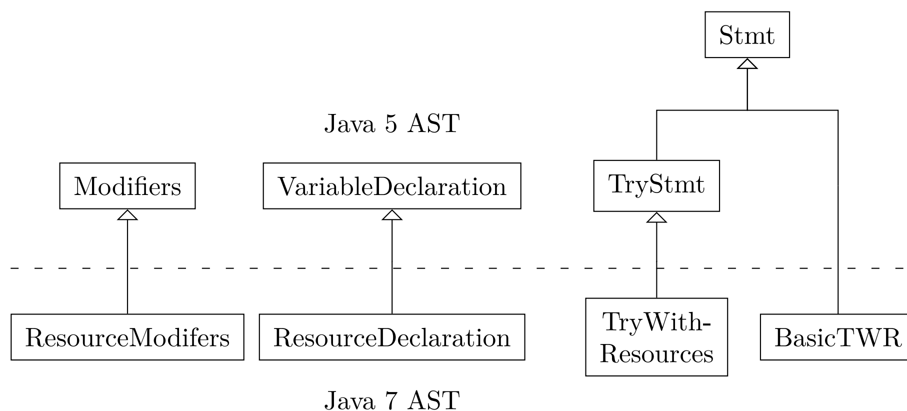


Figure 14: New node types to represent the TWR variation of the `try` statement.

Several existing attributes were extended for the new `TryWithResources` node type to handle front-end static semantic errors etc. These are not covered here, as the changes were mostly trivial.

4.1.4 TWR: Delegated Code Generation

As described above, the full TWR statement can be expressed as a series of nested `try` and basic TWR statements. This fact can be very conveniently exploited using rewrites. However, a rewrite is applied to the AST whenever an AST node with a rewrite rule is accessed[3], and so the static semantic error checking implemented for the TWR statement would have to be implemented for the basic TWR statement instead. This would have a couple of drawbacks, e.g., TWR statements could not be unparsed in the same form as they occur in the source file, and error messages generated for errors such as name collision with a resource name or resources not being subtypes of `AutoCloseable` would be much less descriptive due to the lack of high-level information about the original TWR statement.

In JJ7, in stead of using a rewrite to make the transformation of the TWR statement, a nonterminal attribute (subsubsection 2.4.3) is used. An NTA was added to the `TryWithResources` node, named *Transformed*, which computes a transformed version of the statement. Figure 15 shows this new NTA as a child of a `TryWithResources` node.

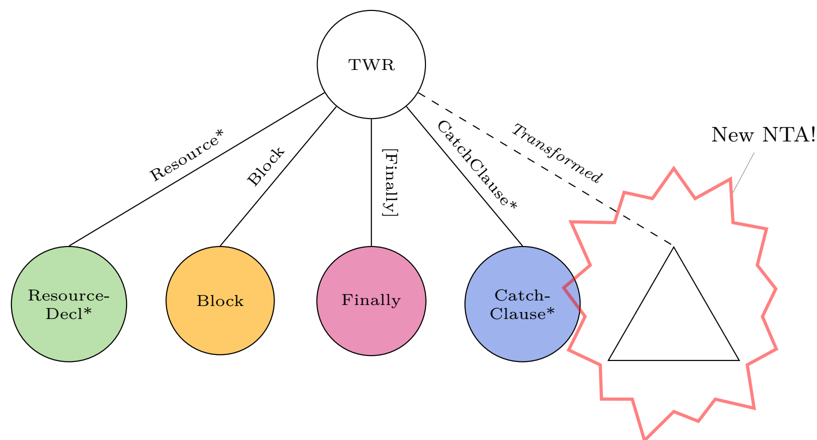


Figure 15: The *Transformed* NTA computes the transformed version of a TWR statement.

During the code generation phase of compilation `TryWithResources` nodes compute the *Transformed* NTA and delegate the code generation task to that generated node. The generated NTA is either a basic TWR node or a `try` statement node, depending on the structure of the original TWR statement. All that remains is to implement the code generation of basic TWR

[3]Some magic can be done using a rewrite condition that is triggered only during code generation, but such an approach gets very messy very quickly.

statements, a much simpler task compared to generating bytecode directly for the TWR statement.

Besides reducing the number of lines of code in the TWR implementation, the delegated code generation approach can save some time during compilation. Since the NTA is only generated when it is first accessed, it will not be computed before the code generation step. If there are any semantic errors in the program being compiled, the NTA will never be computed since compilation halts before code generation.

Another benefit of this approach is the absence of rewrites, which can take considerable time to compute for nodes that are not close to the bottom of the AST. Higher-level node types such as `TryWithResources` tend to be higher up in the AST.

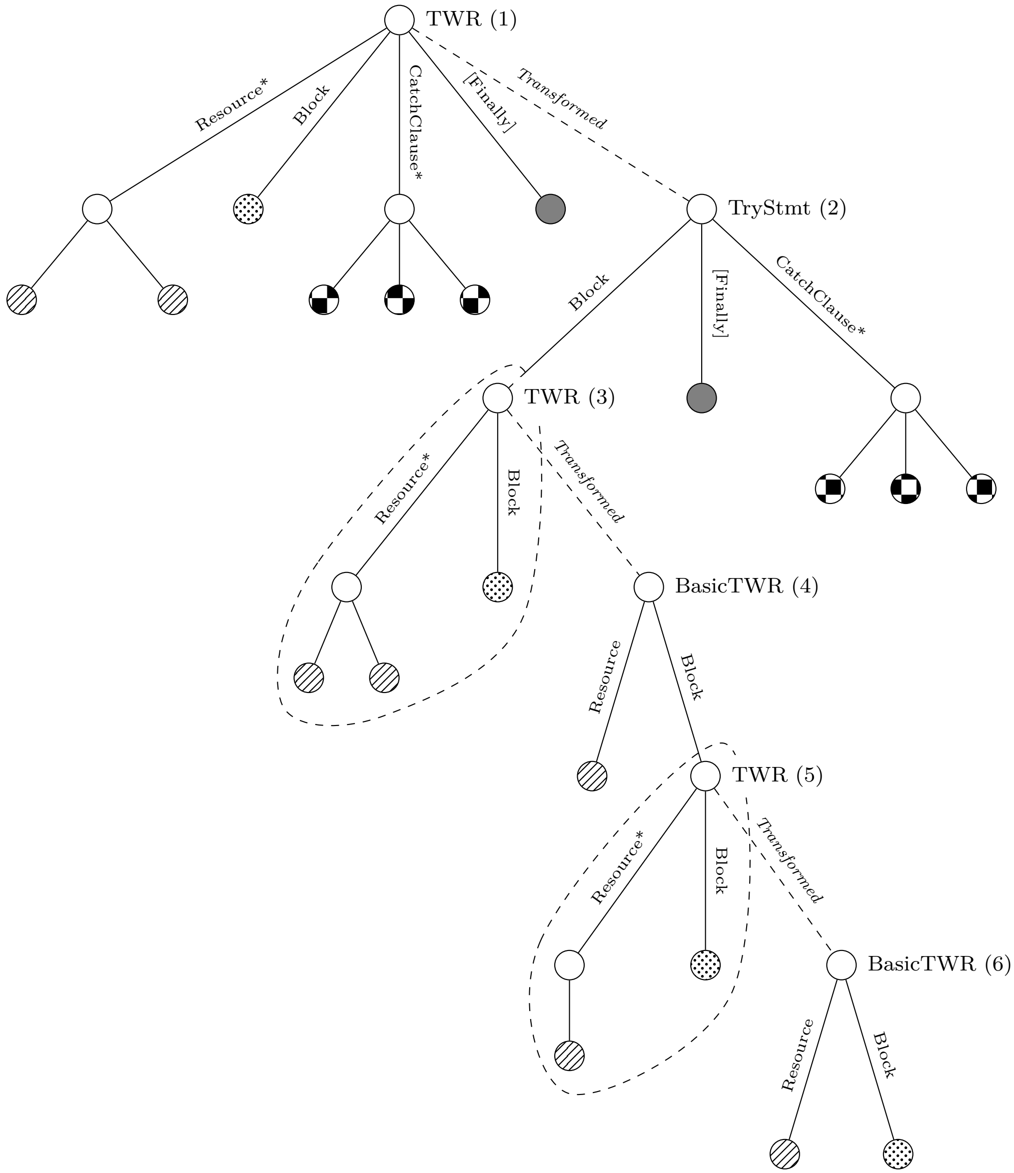


Figure 16: The *Transformed* NTA.

Figure 16 illustrates how the *Transformed* NTA is generated for a TWR statement. The desugaring process described above is used to generate nested `try` and basic TWR statements:

1. We start with TWR (1), this statement has a few `catch` clauses and a `finally` block. We apply step one of the desugaring process to get the *Transformed* NTA: a `try` statement (2) with an inner TWR (3).

2. TWR (3) has no `catch` or `finally`, so its *Transformed* NTA is a basic TWR (4). The first resource declaration ▨ from TWR (3) is copied and used as the resource declaration of the generated basic TWR (4), while the rest of the resources (in this case there is only one left) are copied to a new nested TWR (5).

3. TWR (5) has only one resource declaration and hence can be directly transformed into basic TWR (6).

Note that the original TWR block ▩ has been copied down to the inner basic TWR. Bytecode is generated only for the innermost copy of the original `try` code block. No code is actually generated for the enclosing TWR nodes.

4.2 Strings in Switch

The `switch` statement directs control flow to the first `case` label that matches the `switch` expression. Consider the method `s1` below. It will return -1 if it is called with $i \in \{1, 9\}$, 0 if $i = 13$, or 1 otherwise:

```
int s1(int i) {
    switch (i) {
        case 1:
            return -1;
        case 9:
            return -1;
        case 13:
            return 0;
    }
    return 1;
}
```

Before Java 5, only integer expressions were allowed in `switch` statements. In Java 5, `enum` types were added to the language and also allowed in the `switch` statement. In Java 7, `String`s were added to the set of allowed types in the `switch` statement.

4.2.1 Strings in Switch: Specification

The semantics of a `switch` statement with a `String` expression are exactly the same as for any other type of `switch` statement.

The `switch` statement starts with an expression within parentheses, followed by a block containing a number of `case` labels. Each `case` label must

have a constant expression of the same type as the `switch` expression. The `switch` statement will transfer control to the `case` label within the `switch` block that has an expression which is equal to the `switch` expression.

If no such label is present, and a `default` label exists within the `switch` block, control flow continues at the `default` label, otherwise the entire `switch` block is skipped.

For example, the following method `s2` will return -1, since the `String` `"saw"` does not equal any of the `case` labels — the target of the `switch` will be the `default` label:

```
int s2() {
    switch ("saw") {
        case "tBx":
            return 1;
        case "tBw":
            return 0;
        default:
            return -1;
    }
}
```

The `switch` statement has inherited a few idiosyncrasies from the C programming language. One of these is the so-called *fall-through* feature: a `switch` statement transfers control to one of its `case` labels, but if there is no `break` or any other statement to transfers control out of the `switch` block before the next `case` label, then control flow will continue beyond that label without leaving the `switch` block.

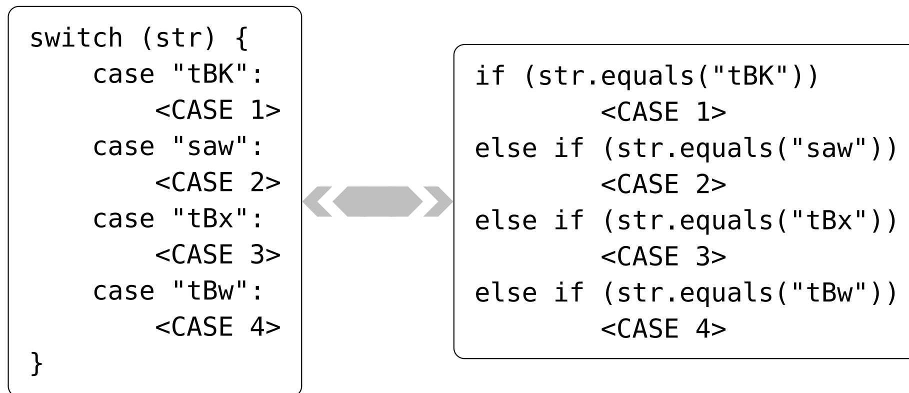
Fall-through is a widely used feature of the `switch` statement. Obviously it should work as usual, even if the `switch` expression is a `String`.

The method `s3` below is exactly equivalent to the method `s1` above. Note the use of fall-through between the first and second `case` labels:

```
int s3(int i) {
    switch (i) {
        case 1:
        case 9:
            return -1;
        case 13:
            return 0;
    }
    return 1;
}
```

4.2.2 Strings in Switch: High-Level Design

Intuitively, a `switch` statement is equivalent to a series of chained `if`-statements:



However, this does not allow fall-through. Additionally, a `switch` statement can achieve better performance than a chain of `if`-statements by reducing the number of branches and comparisons in the generated Java bytecode.

The Java Virtual Machine provides two bytecode instructions to implement `switch` statements; `lookupswitch` [10, p. 525] and `tableswitch` [10, p. 560]. The `tableswitch` instruction requires more memory than `lookupswitch` but may in some cases be faster.

4.2.3 Strings in Switch: Implementation Details

The parser did not have to be modified to accept `String`s in the `switch` statement or `case` labels. Any expression, even `String` literals, was already accepted by the parser.

The existing Java 5 implementation in JastAddJ of the `switch` statement had a type checking method that ensured only expressions of `enum` or integer types (excluding `long`) were used as `switch` expression. The `case` labels were only type checked against the `switch` expression, so once the `switch` expression was accepted by the type checking method the `case` labels were also correctly type-checked.

The only other front-end change for this feature besides type checking was duplicate checking of `case` labels (duplicate `case` labels are not allowed). This was implemented by refining an existing attribute.

The back-end implementation required more work than the front-end. An early goal was to implement the strings in switch feature using `switch` statements in order to preserve fall-through without having to add extra-

neous branch instructions. However, a single `switch` statement was not enough. Somehow the `String` labels had to be enumerated, and the `switch` expression had to be transformed to an index into that enumeration.

The approach used to solve this problem was to generate two `switch` instructions. The first one selects an index that is used in the second. The code generation follows these steps:

1. Enumerate the original `switch` labels. The `default` label is number zero and the other `case` labels are numbered $1, 2, \cdots$ in order of their appearance in the `switch` block.

2. Evaluate the `switch` expression and store it in the temporary variable `#str`.

3. Initialize the temporary index variable `#target` to zero.

4. Switch on the hash code of `#str` to determine the correct value for the index variable.[4]

5. Generate the original `switch` as an ordinary integer `switch` statement, with `#target` as the `switch` expression and replacing the `case` labels for their enumerated indices from step 1.

[4]The `String.hashCode` method returns the same value across different platforms, so there is no risk in storing the hash codes in generated bytecode.

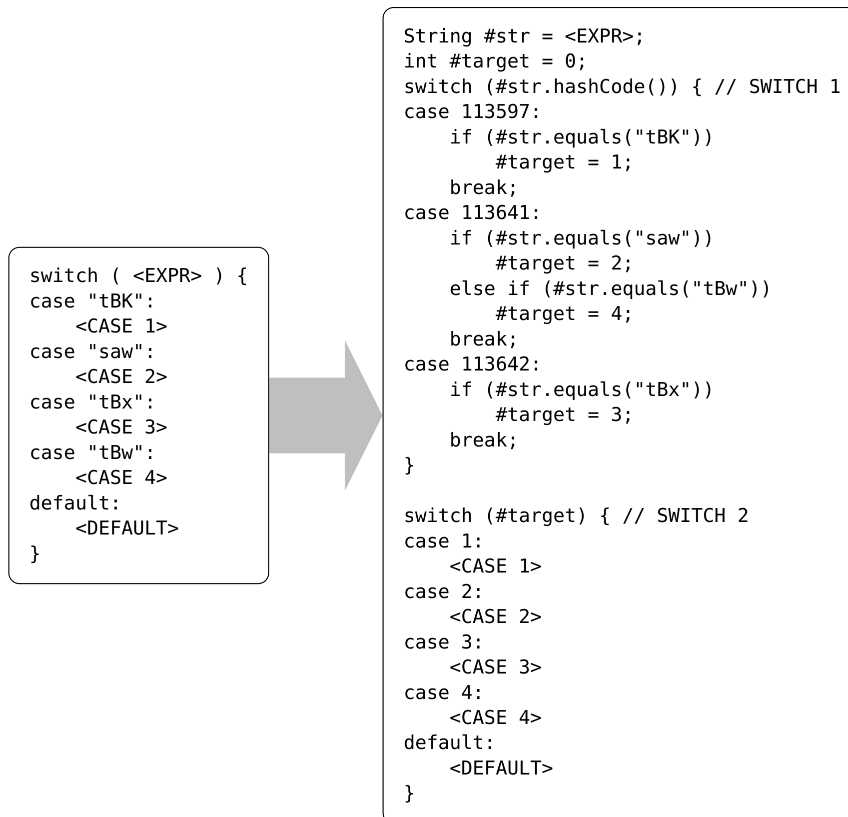


Figure 17: Code generation for a `switch` statement with a `String` expression.

The code generation for the Strings in Switch (SiS) feature was implemented by refining the code generation method of the `switch` statement from the Java 5 version of JastAddJ. The new method checks if the `switch` expression is a `String`, and if so it will enter the SiS specific code generation procedure, otherwise the Java 5 code generation method is invoked.

The SiS specific code generation is not implemented using delegated code generation as with the TWR statement. This is because the SiS code generation requires unnamed temporaries which were not available in the JastAddJ AST already, and adding these would require more work than simply handwriting the new code generation procedure. Adding an unnamed temporary node type and refactoring the implementation of this feature is possible future work.

Figure 17 shows how Java bytecode is generated for a particular `switch` statement. The pseudo code in the figure represents the generated bytecode. In this figure, anything that has a name starting with `#` is an unnamed temporary variable.

The first `switch` branches based on the hash code of the `String` expres-

sion. It is possible for two non-equal `String`s to have the same hash code, so at least one `if` statement is needed after every `case` label to check that the `switch` expression is equal to the constant expression of the `case` label. Note that the two `String`s `"saw"` and `"tBw"` in fact have the same hash code — an extra `if` statement is needed to distinguish between those two possible `String` values.

We can see that fall-through is preserved in the generated code, since the order of the statements from the original `switch` statement is preserved in the second generated `switch` statement — it is merely the `case` labels that have changed, and they are equivalent to the corresponding labels of the original statement.

As previously mentioned, `switch` statements in Java may be implemented using two different bytecode instructions; `lookupswitch` and `tableswitch`. The `tableswitch` instruction is the most efficient in terms of speed, while `lookupswitch` is more space efficient.

The test suite developed during the implementation of JJ7 tests both the `tableswitch` and `lookupswitch` code generation for strings in switch. For information about how this is tested refer to appendix B.

4.3 Diamond

Generics were introduced in Java 5 to aid the programmer by providing improved static type analysis. However, generics in Java 5 were in many cases needlessly verbose. For example, consider the following Java statement:

```
Map<String, List<Integer>> map =
    new HashMap<String, List<Integer>>();
```

In the above example, a new instance of the generic class `HashMap` is created; it maps `String`s to lists of `Integer`s. The programmer must specify the same type arguments twice — once in the variable type and once in the class instance creation expression.

In Java 7 this problem was alleviated with the *Improved Type Inference for Generic Instance Creation* feature, henceforth known as *diamond*. Now, under certain conditions, the type arguments can be omitted:

```
Map<String, List<Integer>> map = new HashMap<>();
```

The name of this feature stems from the appearance of the empty type parameter list `<>` and its resemblance to a parallelogram. The empty type argument list is also called the *diamond operator*.

The diamond feature allows type arguments to be omitted as above in an assignment context, or even when the instance expression is an argument of a method invocation, e.g.:

```
void someMethod(Map<String, List<Integer>> map);
...
someMethod(new HashMap<>());
```

4.3.1 Diamond: Specification

When instantiating a generic class, the programmer may provide an empty type argument list `<>`, called the *diamond operator*. The compiler will then infer the type arguments of the instantiated class.

Type inference for generic instance creation piggybacks on the inference of method type arguments that has been part of the Java language since Java 5. The algorithm for inference of method type arguments is complex, so only an short and informal overview is given below. Refer to the Java Language Specification for the exact algorithm [5, p. 466].

4.3.2 Inference of Method Type Arguments

The type arguments of a generic method invocation are inferred first through the types of the actual arguments using method invocation conversion. Then, if any of the type arguments are still unresolved the return type is used to infer those unresolved type arguments (if possible) using assignment conversion.

Consider the following example:

```
<T> List<T> newList() {
    return new LinkedList<T>();
}

void m() {
    List<Integer> list = newList();
}
```

The `newList` method is a generic method that creates an instance of the generic class `List`. The method `m` has a call to `newList`. Since the method invocation occurs on the right hand side of an assignment, the result is subject to assignment conversion. The assignment conversion implies that

the return type `List<T>` must be assignable to the type `List<Integer->`, and so the type parameter `T` is inferred through the return type to be `Integer`.

Now consider a second invocation of the same method declared above:

```
Integer value = newList().get(0);
```

The result of the method invocation is no longer subject to assignment conversion, so the type argument remains unresolved and `Object` is used as the inferred type argument. The above invocation will not compile, since an `Object` reference can not be directly assigned to a `Integer` reference.

If the `newList` method is changed slightly so that it takes an initial element to put in the list, then type inference will use the argument to calculate the type of `T` without having to use the return type:

```
<T> List<T> newList(T initial) {
    List<T> list = new LinkedList<T>();
    list.add(initial);
    return list;
}

void m() {
    Integer value = newList(3).get(0);
}
```

4.3.3 Diamond: High-Level Design

Class instance expressions are treated as a method invocations. This allows the method type inference to be used to determine the type of the instance expression.

If a generic class C with k constructors is instantiated using the diamond operator, then for each constructor c_i where $i \in 1, 2, \cdots, k$ in C a placeholder method m_i is created. Each placeholder method m_i is parameterized with $n + m$ type parameters, where n is the number of type parameters of C and m is the number of type parameters of c_i ($m \geq 0$). If $P_1, P_2, \cdots, P_n$ are the first n type parameters of m_i, the return type of m_i is $C < P_1, P_2, \cdots, P_n >$.

To determine the type arguments of C the class instance expression is treated as a method invocation, with the placeholder methods as the candidate methods. The regular rules of method invocation are used to select the most specific placeholder method m_j to use, depending on the arguments of

the class instance expression. Type inference is used to determine the type arguments $A_1, A_2, \cdots, A_{n+m}$ of m_j, which in turn gives the inferred type $C < A_1, A_2, \cdots, A_n >$ of the class instance creation expression!

Example 1

```
class E1<T> {
    T pass(T t) { return t; }

    static {
         E1<Integer> instance = new E1<>();
         assert 13 == instance.pass(new Integer(13)).intValue();
    }
}
```

Since the class instance expression is subject to assignment conversion we can omit the type arguments — the compiler will as expected compute the type argument `Integer`.

Example 2

```
class E2<T> {
    T pass(T t) { return t; }

    static {
         assert 13 == new E2<>().pass(new Integer(13)).intValue();
    }
}
```

A trained programmer can probably deduce that the type argument *should* be `Integer`, since the method `pass` is called with an `Integer` argument. However, the call to the constructor has no arguments, and is not subject to assignment conversion, so the type argument can not be resolved. The compiler will use `Object` as type argument instead (method type inference uses `Object` for unresolved type arguments).

After the type of the class instance expression has been determined, method resolution is performed but there is no method named `intValue` in the type `Object`, so the compiler halts with an error.

Example 3

```
class E3<T> {
    T i;
    E3(T i) { this.i = i; }

    static {
        assert 13 == new E3<>(new Integer(13)).i.intValue();
    }
}
```

This example is similar to the previous example. Again, a reasonably skilled programmer can deduce that the type argument should be `Integer` since the type of the argument passed to the constructor is `Integer`.

In this case the Java compiler will also infer the type argument `Integer`, because the actual argument of the constructor constrains the type argument.

4.3.4 Diamond: Implementation Details

Type inference for generic instance creation is entirely managed in the front-end. The implementation only required modifications to the parser, to accept the diamond operator, and to the type analysis of class instance expressions using the diamond operator.

The new node type `DiamondAccess` was added to represent the use of the diamond operator in a `TypeAccess`. New type analysis was implemented for the `DiamondAccess` using the placeholder method approach described above.

The placeholder methods could be implemented using the existing `MethodDecl` node type, but in order to exempt the placeholder methods from semantic error checking the new node type `PlaceholderMethodDecl` was created, dedicated to placeholder methods. The `PlaceholderMethodDecl` implementation was trivial — all of the semantic error checking methods were changed so that they do nothing for this special node type.

The placeholder methods were inserted into the AST in as children of `GenericClassDecl` using a nonterminal attribute. Since nonterminal attributes are only computed when accessed, this approach implies little overhead for supporting the diamond feature — if the diamond operator is not used then no placeholder methods will be generated.

4.4 Improved Numeric Literals

The following numeric literal kinds are available in Java 7:

kind	*octal*	*decimal*	*hexadecimal*	*binary*
Integer literal	✓	✓	✓	since Java 7
Long literal	✓	✓	✓	since Java 7
Floating point literal		✓	since Java 5	
Double literal		✓	since Java 5	

Java 7 allows the programmer to use a binary representation for integer and long literals, and allows underscores in all numeric literals. For example, the following numeric literals are valid in Java 7:

```
0b00101     // binary integer literal
1.3_4       // underscore in decimal double literal
0xb0a7_10ad // underscore in hexadecimal integer literal
100__000    // multiple underscores in integer literal
```

Underscores were added as a meaningless separator character intended to make certain literals more readable to programmers. They are perhaps most useful in binary literals which may become very long — up to 64 digits (not counting leading zeroes)!

4.4.1 Improved Numeric Literals: Specification

A binary literal starts with the characters `0b` and continues with a string of zeroes and ones. The same numeric range limits apply to binary literals as for their hexadecimal counterparts.

Underscores have no meaning in the numeric representation of a numeric literal Any number of underscores may be placed between two digits of a numeric literal, or between the leading zero of an octal literal and the first digit. For example, the following literal is not valid since there is an underscore between a digit and a non-digit: `0xc01d_p001` (`p` is the exponent character for hexadecimal floating point literals).

4.4.2 Improved Numeric Literals: High-Level Design

Underscores in numeric literals cause a new problem in the scanning of Java code: distinguishing between legal and illegal uses of the underscore character during the scanning phase requires lookahead. The existing scanner in

JastAddJ did not use lookahead.

There were two ways to solve this problem; either the scanner could be extended to handle underscores directly by filtering out valid underscores and raising errors for literals with misplaced underscores, or an intermediate node could hold the scanned integer literal, including underscores, to be parsed at a later stage of compilation.

These two approaches both have advantages and disadvantages;

Approach 1 Rewriting the scanner:

- No need to modify the AST definitions.
- A more complex scanner specification using custom scanner routines to handle the lookahead problem.
- Since underscores are filtered out before parsing, numeric literals could not be unparsed in the same format as the source literal.

Approach 2 Adding an intermediate numeric literal node type:

- Simplified scanner rules for numeric literals.
- The possibility to better handle the reporting of syntax errors in integer literals, instead of halting during scanning with a less descriptive error message.
- The AST must be refactored to add the numeric literal node type.

Although the first approach would require less work and refactoring, it was decided that due to the improved error feedback the second approach was slightly favorable.

4.4.3 Improved Numeric Literals: Implementation Details

Two new node types were added to the AST; `NumericLiteral` and `IllegalLiteral`.

`NumericLiteral` is the intermediate representation of a parsed numeric literal. A rewrite rule was added for this node type which rewrites it either to an `IllegalLiteral`, if there are any syntax errors, or one of the specific numeric literal node types `IntegerLiteral`, `LongLiteral`, `FloatingPointLiteral` or `DoubleLiteral`.

In JastAddJ the scanner rules for numeric literals were complex in order to allow all legal syntax variations yet not accept any kind of malformed literal.

During the implementation of the *Improved Numeric Literals* feature of Java 7 the scanner rules for different kinds of numeric literals were made less strict and thus simplified. Most of the possible syntax errors are checked in the numeric literal parsing method instead, so in JJ7 most numeric literal syntax errors are reported with more detailed error messages than previously.

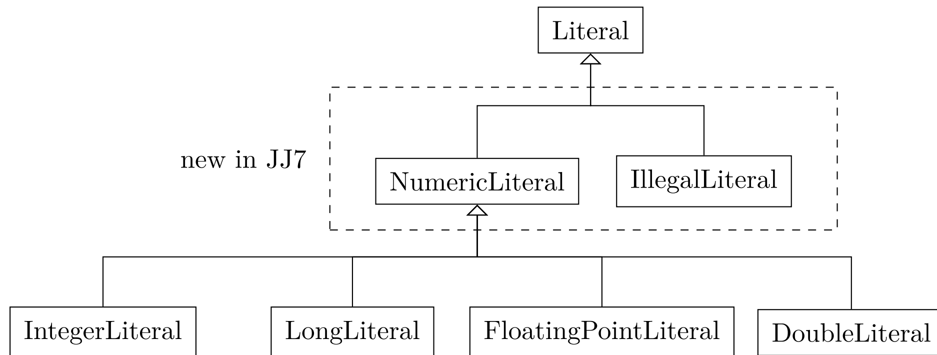


Figure 18: The new `Numeric-Literal` and `Illegal-Literal` node types.

4.5 Multi-Catch

The `catch` clause of a `try` statement could, up to Java 6, catch a single exception type. Even if several exception handlers did the same thing, each exception had to be caught individually:

```
try {
    // this may throw exceptions Problem1 or Problem2:
    doSomething();
} catch (Problem1 e) {
    handleProblem();
} catch (Problem2 e) {
    handleProblem();
}
```

Using the multi-catch feature of Java 7, the above example can be simplified to:

```
try {
    doSomething();
} catch (Problem1 | Problem2 e) {
    handleProblem();
}
```

4.5.1 Multi-Catch: Specification

The `catch` clause may use a union type as the type of its declared exception parameter. A union type is a list of types, that are not in a subtype (or supertype) relation to each other, separated by the `|` character.

A `catch` clause used as described above is called a multi-catch clause and its exception parameter is implicitly `final`. The catchable exception types of a multi-catch clause are the types specified in the union type, but the effective type of the exception parameter is computed by an algorithm known as *lub* in the Java Language Specification. The type of an exception parameter that has a declared union type of $T_1|T_2|\cdots|T_n$ is $lub(T_1, T_2, \cdots, T_n)$ [5, p. 402].

4.5.2 Multi-Catch: Implementation Details

Since the union type was only used in the multi-catch clause, it was not useful to create a new AST node type to represent union types — at least not a type that could be used in the already existing `CatchClause` node type. Instead, the `CatchParameterDeclaration` node type was added to represent exception parameters in multi-catch clauses. Due to fundamental incompatibilities between node types `ParameterDeclaration` (used in `CatchClause`) and `CatchParameterDeclaration`, there was no simple way to make the latter a subclass of the former. Since `CatchClause` could not represent a multi-catch, the `MultiCatch` node type was added for that purpose.

While `CatchParameterDeclaration` and `ParameterDeclaration` had nearly nothing in common, `CatchClause` and `MultiCatch` did have very much in common. Most attributes and methods would be exactly the same for both these node types yet `MultiCatch` could not inherit from `CatchClause` because they had different exception parameter children.

In order to not create a lot of duplicated code, and to simplify the implementation of the multi-catch feature, `CatchClause` was refactored so that it became an abstract superclass to the new `BasicCatch` node type which took the old role of `CatchClause`. With this refactoring applied to JastAddJ it was simple to implement the `MultiCatch` node type as a subclass of `CatchClause`. All computations that were identical between `MultiCatch` and

`BasicCatch` were placed in `CatchClause` — only a few attributes needed to be implemented for `MultiCatch`.

The *lub* algorithm was already implemented in the Java 5 extension of JastAddJ, as this algorithm is part of the type inference for method invocations that was added in Java 5. Using the *lub* algorithm in the type analysis of the `MultiCatch` node type was quite simple.

4.5.3 Multi-Catch: Alternative Implementation

The way multi-catch was implemented splits the `CatchClause` node type into `BasicCatch` and `MultiCatch`. However, it could have been enough to just use a more general `CatchParameterDeclaration` in the original `CatchClause`, but that would in effect require implementing the multi-catch feature, yet without it being used, in the core version of JastAddJ.

The chosen implementation only refactored the core JastAddJ AST definitions, while the extension adds all the required new attributes and node types.

4.6 More Precise Rethrow

A rethrown exception would in Java 6 and earlier versions of the language require exceptions of the same type as the exception parameter to be handled by an enclosing `try`-statement or declared to be thrown by the enclosing method. For example, in the following code snippet it is obvious that only a `FileNotFoundException` can be thrown by the method `m`, yet the more general exception type `IOException` must be declared to be thrown since that is the declared type of the exception parameter `e`:

```
// Java 6 & earlier:
void m() throws IOException {
    try {
        throw new FileNotFoundException();
    } catch (IOException e) {
        throw e;// rethrow of IOException
    }
}
```

In Java 7, type analysis of rethrown exception parameters is refined, so the `throws` declaration of `m` can be made more specific:

```
// Java 7:
void m() throws FileNotFoundException {
    try {
        throw new FileNotFoundException();
    } catch (IOException e) {
        throw e;// rethrow of FileNotFoundException
    }
}
```

4.6.1 More Precise Rethrow: Specification

A `final` or effectively `final` exception parameter is subject to more precise exception analysis in the context of a `throw` statement. Only the exception types that are catchable by the corresponding `catch` clause may be rethrown.

Example In method `foo` below there is an unreachable `catch` clause which in Java 6 or earlier would have been considered reachable by the compiler:

```
class E1 extends Exception {}
class E2 extends Exception {}
public void foo() {
    try {
        throw new E1();
    } catch (Exception e) {
        try {
            throw e;
        } catch (E1 e1) {
        } catch (E2 e2) {
            // unreachable
        }
    }
}
```

A Java 7 compiler will find that since the exception parameter `e` is effectively `final` (it is never assigned to), and the corresponding `catch` clause can only catch exceptions of type `E1`, the `throw e;` statement throws an exception of type `E1` and thus the second `catch` clause of the inner `try` statement is unreachable!

4.6.2 More Precise Rethrow: High-Level Design

The computed type of a `throw` statement needed to be altered for `throw` statements that rethrow an effectively `final` exception parameter. However, the effective type of the exception parameter must remain unchanged during all other type-dependent computations such as method lookup etc.

In the above example, the type of `e` is still `Exception` yet only exceptions of type `E1` can be thrown by the `throw e;` statement.

This dichotomy was resolved by adding a new attribute to handle the thrown exception types of a `throw` statement separately.

4.6.3 More Precise Rethrow: Implementation Details

The `throwTypes` and `effectivelyFinal` attributes were the main attributes added in the implementation of the more precise rethrow feature.

The `effectivelyFinal` attribute was added to the `ParameterDeclaration` node type which represents exception parameters of uni-catch clauses. An exception parameter is effectively `final` either if it is declared `final` or if it is not assigned to in the scope of its `catch` block. The exception parameter of a multi-catch clause is always effectively `final`.

Relevant exception checking attributes and methods were modified to use the `throwTypes` attribute which was added to the `Expr` node type. For `ParameterDeclaration` and `CatchParameterDeclaration` this attribute returns the list of (re-)throwable types.

4.7 Safe Varargs

Unchecked warnings were added in Java 5. They are generated to help the programmer prevent heap pollution. [15] Since Java 5, calling a method with a variable arity parameter of non-reifiable type would generate an unchecked warning at the method invocation. These unchecked warnings could be suppressed using the `@SuppressWarnings` annotation at the method invocation.

In Java 7 there is an additional unchecked warning at the method declaration, but the programmer can use the `@SafeVarargs` annotation to inform the compiler that the declaration is safe and can not cause heap pollution, thus silencing the warnings both at the declaration and all call sites.

4.7.1 Safe Varargs: Specification

The declaration of a method with a non-reifiable parameter which also has variable arity will cause the compiler to report an unchecked warning. This warning can be suppressed with the `@SafeVarargs` annotation, which also

suppresses the related unchecked warnings raised by all invocations of the method:

```
@SafeVarargs
public static <T> void foo(T... a) { }
```

The `@SafeVarargs` annotation may not be used on anything that is not a method declaration with a generic variable arity parameter.

4.7.2 Safe Varargs: High-Level Design

The safe varargs feature was simple to implement in JastAddJ. There were attributes already available to determine if a method had variable arity parameters, and whether the method invocation was affected by the `@SuppressWarnings("unchecked")` annotation. The only things that needed to be added were attributes to compute if an argument was reifiable, and whether or not the method declaration used the `@SafeVarargs` annotation.

Even though attributes were available in JastAddJ to handle the `@SuppressWarnings` annotation, no unchecked warnings were actually generated. So in order to fulfill the entire specification of the safe varargs feature, the unchecked warnings that were missing in JastAddJ were also implemented.

5 Discussion

The implementation of JJ7 was mostly straightforward: for most features it was simple to find the relevant attributes and figure out what needed to be modified or added. Other features required significantly more work. I spent most time on the Diamond feature. This feature utilises the already implemented type inference for generic methods in JastAddJ. It was difficult to understand how that type inference worked — there are very many attributes concerning type analysis, and their dependencies are complex. The use of non-terminal attributes makes debugging difficult due to the way they can dynamically alter the AST at any time during execution.

Even though there are definitely parts of JastAddJ that are far from trivial to understand, I suspect that it is a simpler task for someone who is already familiar with the JastAdd system to implement a new feature in JastAddJ than it would be to implement the same feature in a javac-like system.

I believe that using RAGs in most cases makes a compiler easier to understand without concious effort on the developers' part. However there are

some problems such as the type analysis in JastAddJ that require great care in order to make the attribute design as simple as possible. A simple design reduces the cost of maintaining or extending the code.

The greatest benefits of using JastAdd is the concise syntax and extensibility of JastAdd code. Changing attributes and other inter-type declarations can be done very easily by adding an additional aspect file. New AST definitions can also be added modularly with JastAdd, but changing existing AST definitions may break compatibility with previous extensions.

An obvious drawback for JastAdd developers is the lack of mature developer tools such as a powerful IDE and debugger. It can be difficult to navigate JastAdd source code, particularly when looking for the declaration or implementation of specific attributes or inter-type declarations. Debugging can be done using a regular Java debugger, but requires stepping through the generated attribute evaluation code in order to find relevant attribute code.

5.1 Combining Modules

During the implementation I strived for high modularization, even trying to modularize individual features in JJ7. In some cases it was possible to develop individual features independently of others in separate modules, however there were cases where there was some interdependence between features. For example, the refactoring that was used for the `CatchClause` node type during the implementation of the multi-catch feature affected the implementation of the try-with-resources feature. The multi-catch feature could have been implemented without this refactoring but then the overall quality of the implementation would have suffered — several duplicated methods would have been added.

There are certainly cases of interdependent features in the implementation of Java 5 in JastAddJ. For example, the auto-boxing feature causes an edge case in the implementation of the enhanced for loop which needs to check explicitly for auto-boxed types and unbox them. Again, those features could probably be separated with some smart refactoring to the core JastAddJ compiler.

5.2 Problems Encountered

During the implementation of the Java 7 language features, some problems in both the tools used to build JastAddJ and in JastAddJ itself were discovered.

These problems are documented in the following sections.

5.2.1 Bugs in JastAddJ

Various bugs were identified in JastAddJ and fixed during the development of the Java 7 extension. Although several of these bugs were not critical for JJ7, fixing them was taken as an opportunity to learn more about JastAddJ.

1. Exception handling was not checked for `ClassInstanceExpr`: JastAddJ did not check that exceptions thrown by a class instance expression were caught.

2. A code generation bug in the enhanced `for` loop caused JastAddJ to output defect bytecode if the enhanced `for` loop iterated over an `Iterable` object.

3. Unchecked conversion warnings were not raised by JastAddJ. Some of these warnings were added together with the unchecked warnings required by the safe varargs feature.

4. There was an error that caused certain `throw` statements to be incorrectly type checked. JastAddJ would for example not accept any type of catch clause for the following throw statement:

```
throw (System.currentTimeMillis() % 2 == 0) ?
    new E1() : new E2();
```

5. The `@Override` annotation's semantics was changed in Java 6. JastAddJ needed to be updated to reflect this change [1].

5.2.2 Problems With Beaver

The *Beaver* parser generator is used to generate JastAddJ's parser (see subsection 2.5). It has an option to optimize the generated parser's parsing table. This removes states in the parsing table that correspond to multiple productions sharing the same lookahead.

Mysterious parse errors started appearing during the implementation of try-with-resources, when the test cases were compiled. These parsing problems ceased when the optimization option in Beaver was turned off.

Another error in Beaver was discovered at a later time, when parsing some Java programs would cause the generated parser to reliably crash. After updating Beaver to version 9.7, those crashes also disappeared.

6 Evaluation

In this section the results of benchmarking JastAddJ against OpenJDK's javac are presented.

In subsection 6.1 the size of the code base for JJ7 is compared to that of javac.

In subsection 6.2 the programs used to measure compile time and memory usage are presented, and in subsection 6.3 the details of the environment and method used to obtain these measurements are listed.

Section 6.4 contains the benchmark results. The results of the benchmark help to answer the question about the efficiency of the implementation of JJ7.

6.1 Implementation Size

The Source Lines of Code (SLOC) metric is used to compare implementation sizes. SLOC counts were obtained using the *SLOCCount* program by David A. Wheeler and represent physical lines of source code, excluding comment lines and empty lines.

The generated Java code of JastAddJ is about 78 thousand lines, but the JastAdd source files used to generate JastAddJ are only 25 thousand SLOC, and of those the JJ7 extension is only 2.2 thousand lines.

The SLOC counts of JastAddJ (non-generated code) and javac are listed in figure 19. Listed in this table is also the percent increase in number of lines of code since the previous version of each compiler. Note that there is no Java 6 version of JastAddJ since the Java 5 version also supports Java 6. Also note that the increase for JastAddJ is lower than that for javac, between the current and previous versions.

The fact that the Java 7 implementation in JastAddJ required a lower percent increase in code size than javac indicates that fewer things had to be added in JastAddJ to support the Java 7 features. This supports our claim that JastAddJ is simpler to extend than javac-like compilers.

Compiler	*SLOC*	*increase*
OpenJDK 6 b24	47397	–
OpenJDK 7 b146	55130	16.3%
JastAddJ 1.5 R20120306	23097	–
JastAddJ 7 R20120306	25271	9.4%

Figure 19: Source Lines of Code (SLOC) counts for various versions of JastAddJ and javac.

6.2 Benchmark Programs

The following benchmark programs were compiled by JastAddJ and javac in order to compare the compile time and memory usage of the two compilers:

Name	*Version*	*SLOC*	*Description*
antlr	2.7.2	34615	Aspect programming system
clojure	1.3.0-RC0	35831	Compiler for the clojure language
jastaddj	R20111208	87342	JastAddJ
javac	jdk7-b146	55130	OpenJDK 7 javac
jdepend	2.9.1	2460	Java package dependency analyzer
jsilver	1.0.1-SNAP	30164	HTML template system
jython	2.2alpha1	76368	A Python environment in Java
lucene	3.0.1	45337	Text search engine

Figure 20: Benchmark programs

The source code for the benchmark programs was written in Java version 6 or earlier. None of them used Java 7 features, although the javac benchmark program does depend on the Java 7 class library. At the time the benchmark programs were selected we did not find any other suitable Java 7 programs.

6.3 Measurement Details

The compilation time and memory usage measurements were obtained by running the benchmarks in a "hot" client-mode JVM. The compilers were programatically invoked for a warm-up routine of five invocations with just-in-time (JIT) compilation enabled, then an additional fifteen invocations with JIT disabled. After the warm-up runs the compiler was invoked fifty times, while measuring the total compile time and memory usage for each invocation. Memory usage was estimated by taking the current heap size before each invocation and subtracting that number from the current heap size after the invocation. The arithmetic mean of the fifty measurements are displayed in the result tables in the following section.

The machine used to run the benchmarks was an Intel® Core™2 CPU running at 1.83GHz with 2048 KiB cache and 2 GiB primary memory. The test machine was running Ubuntu 12.04.1. The JVM used was an OpenJDK 7 (update 3) JVM with the maximum heap size set to 2 GiB.

The additional command line arguments for the JVM used to run JastAddJ and javac were:

```
-Xms128m -Xmx2048m -XX:ReservedCodeCacheSize=256m
```

JastAddJ 7 (build R20120306) and OpenJDK 7 (update 3) javac were used to compile the benchmark programs. The default options for both compilers were used, and the compiled programs were not executed.

6.4 Benchmark Results

Figure 21 lists the average compile time and memory usage for each benchmark program and compiler combination. The compile time results are also plotted in figure 22. Figure 23 plots the JastAddJ compile time as a percentage of the corresponding compile time using javac for the same benchmark program.

Benchmark	javac		jastaddj	
	s	MiB	s (% of javac)	MiB
antlr	1.32	50	2.62 (199%)	61
clojure	3.53	61	3.50 (99%)	110
jastaddj	4.15	102	8.14 (196%)	214
javac	2.43	137	5.43 (224%)	144
jdepend	0.14	47	0.58 (401%)	35
jsilver	1.05	84	2.85 (270%)	71
jython	3.48	91	6.84 (197%)	149
lucene	1.76	95	4.98 (282%)	123

Figure 21: Benchmark result summary

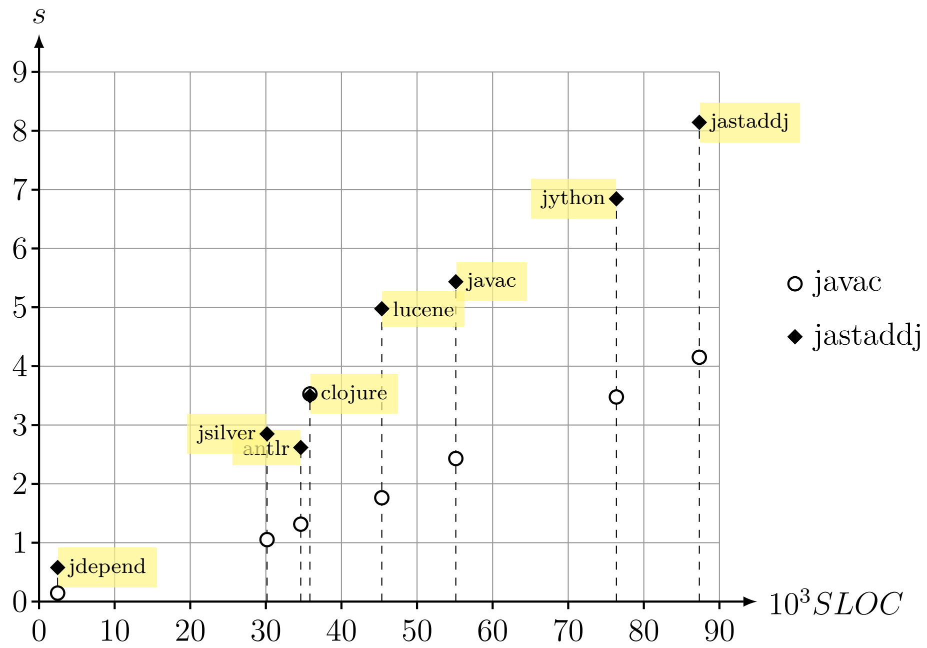


Figure 22: Average compile time.

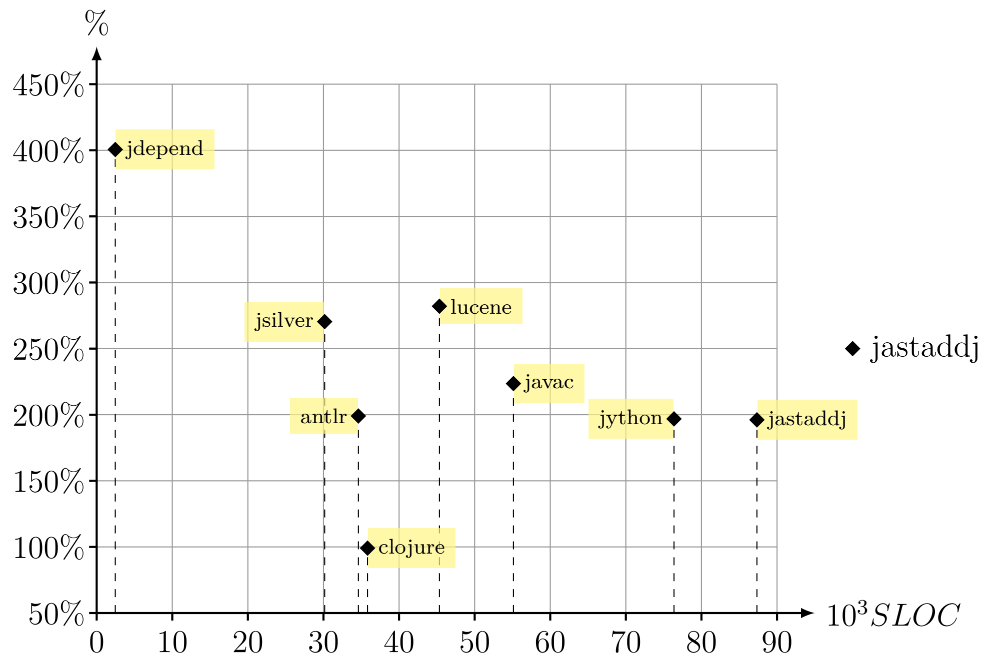


Figure 23: Compile time measured in % of javac compile time.

As can be seen in figure 23, the compile time using JastAddJ is between 100% and 300% that of the javac compile time for all benchmarks except jdepend. There is no clear relationship between the slowdown and the number of source lines of code. For JastAddJ it appears that the compile time is in a linear relationship to the SLOC count. The javac compile time is also nearly linear, if we ignore clojure.

It is surprising that clojure takes so long time to compile with javac. This could perhaps be attributed to some coding pattern used extensively in clojure which javac does not compile efficiently. JastAddJ uses the same time to compile clojure, but with just less than twice the memory usage.

Previous tests with the Java 1.4 version of JastAddJ and JDK indicated that JastAddJ was just less than three times slower than javac. The data gathered in this thesis suggests that the performance gap in compilation time between JastAddJ and javac has not changed significantly. [3] No special optimizations were used in the implementation of JJ7, on the other hand nearly all of the computations and nodes used for a Java 7 feature will only be used if the feature is actually used in the compiled program. We ran the same benchmarks on the Java 1.5 version of JastAddJ and saw no significant difference in the compile time and memory usage compared to the Java 7 version.

7 Conclusion

In this thesis we have presented the implementation of JJ7, an extension to JastAddJ adding support for the new Java 7 language features listed in subsection 2.1. The design decisions and approaches used to implement each feature were presented in section 4. My reflections on the implementation work and modularity using JastAdd are listed in section 5. Finally, in section 6 I compared JastAddJ to javac, measuring implementation size, compile time and memory usage.

The first main question that was asked in the introduction to this thesis is how modular the implementation would be. As mentioned in section 5, the features are not fully modular — i.e. separate features can not be selected independently of each other. However, perfect modularity would require predesigned support for the particular features of Java 7. Despite this JJ7 did not require many refactorings to JastAddJ and with those changes implemented it functions as a separate module.

The question of correctness can not fully be answered. There are currently no known bugs in the implementation of the Java 7 features, but as new bugs are very hard to anticipate it is not possible to say that the implementation

is correct. Judging only by the limited unit tests discussed in subsection 3.1, the implementation is also complete. Oracle's test suite to validate Java compilers was not used and so we can make no claims as to the conformance of JastAddJ with the Java 7 specification.

Finally, we have the question of efficiency. As seen in section 6, the JastAddJ compiler is approximately 100% to 300% slower than javac for large programs. JastAddJ was just below a factor three slower than javac when the Java 1.4 versions of each compiler were compared. It is not surprising that the difference in compile time stayed roughly the same. The result shows none the less that JastAddJ has not become over-encumbered by the new features.

7.1 Future Work

The following are potential areas for future work, based on JJ7:

- The upcoming Java version, Java 8 could be implemented in JastAddJ, using JJ7.
- The evaluation of JastAddJ in this thesis is very basic. A more in-depth evaluation using more Java compilers to compare JastAddJ against, and more benchmark programs, would be very valuable. In particular it would be interesting to see what the rather large variations in compile time, between different test programs, are caused by. The results of further studying the performance of JastAddJ could be used to optimize JastAddJ for decreased compile time or memory usage.
- There is a refactoring opportunity in JJ7: the strings in switch feature could be implemented using a nonterminal attribute. This would require an unnamed temporary node type to be added to the AST (cf. 4.2).
- There exist many extensions to JastAddJ, built using the Java 5 version, that could be updated to utilise the Java 7 features and core JastAddJ refactorings implemented in JJ7. Such extensions include dataflow analysis, a java-to-C cross compiler, and many other research projects and applications.
- Obtaining a license for Oracle's compatibility test suites for Java would be very valuable in order to test the completeness and correctness of JJ7 with regards to the Java 7 specification.

Appendices

A Obtaining JastAddJ

JastAddJ is available via anonymous SVN at `http://svn.cs.lth.se/svn/jastadd-oxford/projects/branches/JastAddJ-stable/`.

The compiler can be built using either the Apache Ant script `build.xml` in the root directory of JastAddJ, or in any of the subdirectories. Each subdirectory corresponds to one module of JastAddJ (subsection 2.2).

B Strings-in-Switch Tests

The two bytecode instructions `lookupswitch` and `tableswitch` can be used to implement the `switch` statement. JastAddJ generates the `switch` instruction that produces the smallest bytecode for any given `switch` statement. The size of `lookupswitch` is $4 + n * 8$ bytes, where n is the number of `case` labels. The size of `tableswitch` is $8 + (h - l + 1) * 4$ bytes, where h is the value of the highest `case` label and l is the value of the lowest.

The JJ7 test suite tests the code generation for both these instructions. In order to test both, the test cases must be cleverly constructed so that it is known which instruction will be generated. This can be done by looking at the hash codes of the strings used in the `case` labels.

For example the following `switch` statement, when compiled with JastAddJ, will produce a `lookupswitch` since there are few `case` labels, but the difference in their values is large ($h > 10^8$, $l = 97$).

```
switch (value) {
    case "a":
        break;
    case "b":
        break;
    case "c":
        break;
    case "smurf":
        break;
}
```

The next statement will produce a `tableswitch` (size = 24 bytes) rather than a lookupswitch (size = 36 bytes):

```
switch (value) {
    case "a":
        break;
    case "b":
        break;
    case "c":
        break;
    case "d":
        break;
}
```

References

[1] Peter Ahé. Override snafu. http://blogs.oracle.com/ahe/entry/override_snafu, November 2011.

[2] Beaver. Homepage. http://beaver.sourceforge.net/, September 2011.

[3] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[4] E. Gamma and K. Beck. Junit. *http://www.junit.org*, 2011.

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java^{TM} Language Specification: Java SE 7 Edition*. Oracle, February 2012.

[6] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[7] JFlex. Homepage. http://jflex.de/, September 2011.

[8] J.W. Klop et al. Term rewriting systems. *Handbook of logic in computer science*, 2:1–116, 1992.

[9] D.E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[10] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java^{TM} Virtual Machine Specification: Java SE 7 Edition*. Oracle, February 2012.

[11] Oracle. Jdk 7 release notes. http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-429209.html, July 2011.

[12] Java Community Process. Java specification request 334. http://www.jcp.org/jsr/detail?id=334, June 2011.

[13] The Jikes Project. Homepage. http://jikes.sourceforge.net/, June 2011.

[14] The Mauve Project. Homepage. http://sources.redhat.com/mauve/, June 2011.

[15] The Java^{TM} Tutorials. Using non-reifiable parameters with varargs methods. http://download.oracle.com/javase/tutorial/java/generics/non-reifiable-varargs-type.html, August 2011.

[16] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 131–145. ACM, 1989.

[17] Xiaoqing Wu, Barrett R. Bryant, Jeff Gray, Suman Roychoudhury, and Marjan Mernik. Separation of concerns in compiler development using aspect-orientation. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1585–1590. ACM, 2006.