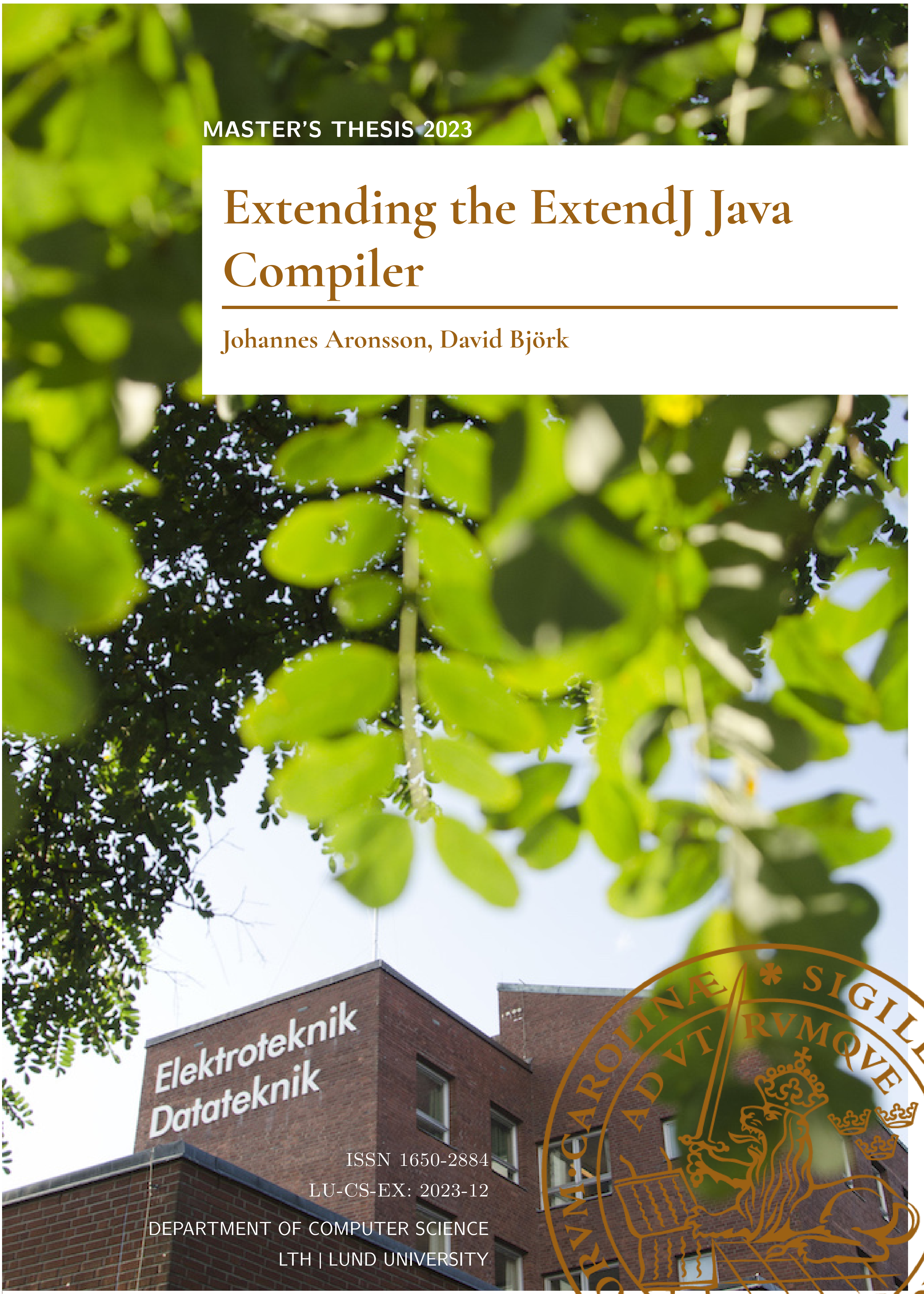
MASTER'S THESIS 2023

Extending the ExtendJ Java Compiler

Johannes Aronsson, David Björk



ISSN 1650-2884
LU-CS-EX: 2023-12

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-12

Extending the ExtendJ Java Compiler

Utökning av ExtendJ Java kompilatorn

Johannes Aronsson, David Björk

Extending the ExtendJ Java Compiler

Johannes Aronsson
jo5152ar-s@student.lu.se

David Björk
da1705bj-s@student.lu.se

June 5, 2023

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Idriss Riouak, idriss.riouak@cs.lth.se

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

ExtendJ is a Java compiler supporting Java versions from 4 to 8, and it is built using the JastAdd metacompiler. ExtendJ is designed to enable modular extensions. This thesis aims to examine ExtendJ's extendibility and performance by attempting to add support for Java versions 9, 10, and 11. Many features were introduced in these versions, including local type inference with the `var` identifier. The implemented features were then evaluated by compiling real-world projects to test the implementation, and measure compilation time as well as memory usage. Finding and compiling relevant projects proved difficult, and almost only projects using Java 8 features and earlier where compiled. The performance of ExtendJ versions 8 to 11 was compared with the corresponding OpenJDK compilers by measuring compilation time and memory consumption. The compilation time of ExtendJ was found to be within a factor of 3, while the memory consumption was within a factor of 6. We also found that ExtendJ is modularly extensible to a high degree.

Keywords: Java, JastAdd, Compiler, Reference Attribute Grammars

Acknowledgments

We would like to thank our supervisor, Idriss Riouak, for his continuous help throughout the entire thesis work with weekly meetings and for his additional help when we needed it.

Contents

Contribution Statement

The table below indicates the responsibilities each author had in writing this thesis:

Author	*Writing*	*Implementation*	*Compiling Projects*	*Performance Evaluation*
J. Aronsson	◐	◐	●	○
D. Björk	◐	◐	○	●

The dark portion of the circle represents the amount of work and responsibilities assigned to each author for each individual step:

◔ Author was a minor contributor to the work.

◐ Author was a contributor to the work.

◕ Author led and did a majority of the work.

● Author led and did almost all of the work.

Chapter 1
Introduction

Compilers are an essential tool in the software development process. A compiler translates code from a source programming language into another target language. They make it possible to write code in a high-level language that is then compiled into a lower-level language or executable. The high-level language can be designed to be easy and fast to write, understand, and debug making development efficient. The target language can instead be designed for execution speed, memory usage and processor architecture-specific performance. Designing a language to be compiled can therefore lead to high-level abstractions and good performance.

The Java programming language is a high-level language compiled into a lower-level language called Java bytecode. Java code can be compiled using the official reference Java compiler included in OpenJDK, or using other open-source or commercial alternatives, e.g., the Eclipse compiler. ExtendJ [1], formerly JastAddJ, is yet another Java compiler and was built to support the research of compilers. ExtendJ is based on *Reference Attribute Grammars* (RAGs) that are extensible by their nature. RAGs specify the behavior of a programming language by declaring attributes for nodes in a tree, e.g., the *Abstract Syntax Tree* (AST) created by a parser. The implementation of RAGs that ExtendJ uses is the JastAdd [2] metacompiler. The JastAdd metacompiler is a system for compiler construction and code analysis tools using RAGs. A main feature of both the system and the ExtendJ compiler is enabling modular extensions [1, 2].

Several extensions have been created for the ExtendJ compiler, e.g., the two static program analysis tools, IntraCFG [3] and a non-null checker [4]. Another example is the JFeature [5] tool that uses ExtendJ to identify Java features for different projects.

ExtendJ currently supports Java versions 4-8, where Java 4 is the base version, and the newer versions have been added as modules to the compiler [6, 7]. To further test the extendibility of ExtendJ and JastAdd and to allow for future extensions to be built for ExtendJ, this thesis aimed to add support for more recent Java versions. Extending ExtendJ would improve existing tools, e.g., updating JFeature to be able to analyze more recent projects using the new Java language features. The first long-term support (LTS) version of Java was Java 8, later followed by Java 11. This thesis aimed to add support for Java 9, 10, and

11 features so that ExtendJ can eventually be fully compliant with the Java 11 LTS release. In addition to improving ExtendJ, this thesis aimed to evaluate it in two main ways. The first one was to assess to what extent it is possible to add extensions in a modular fashion since that is an essential feature of the compiler and the JastAdd system. The second was to measure the extended versions' performance and precision limitations compared to the OpenJDK-based Java compilers. These goals are presented below as two research questions we sought to answer in this thesis.

RQ1: How modularly extensible is the ExtendJ Java compiler?
RQ2: What are the limitations in performance and precision compared to the javac compiler?

The structure of this report is as follows. Chapter 2 presents the theoretical foundations for compiler construction, the JastAdd system and the Java language. Chapter 3 introduces the changes required to add support for the Java 9, Java 10, and Java 11 features. Then, the evaluation of the compiler is presented in Chapter 4. Finally, we discuss the results in Chapter 5 and answer the research questions in Chapter 6.

Chapter 2

Background

A compiler can significantly affect the compiled code's performance and development efficiency through faster build times. This chapter begins by outlining the fundamental steps that many compilers employ to compile a program in Section 2.1. Then, the ExtendJ compiler and the JastAdd system are described in Sections 2.2 and 2.3. In Section 2.4, we introduce the Java programming language, and the new language features introduced in Java 9, 10, and 11. Finally, we will describe how the performance and precision of a Java program can be analyzed in Section 2.5.

2.1 Compiler Architecture and Functionality

A compiler can perform many tasks and compilers for different languages can significantly differ. However, a compiler usually consists of a front-end and a back-end. The purpose of the front-end is usually scanning and parsing the source code as well as performing syntactic and semantic analysis. The back-end then generates the target code for the program. This section provides a high-level overview of the key concepts involved in the construction and phases of a compiler, in the context of ExtendJ.

2.1.1 Scanning and Parsing

The first step ExtendJ performs is *scanning*. The scanning phase uses JFlex [8] to generate a scanner from the developer's scanner specification. The specification consists of regular expressions describing what tokens to create when scanning through a source file.

The created stream of tokens is then parsed into an Abstract Syntax Tree that follows a JastAdd abstract grammar [2]. This is done using the Beaver parser generator [9] to transform the parser specification into an LR parser with a look-ahead of one (LALR(1)) [10]. An LALR(1) parser has limited knowledge about the context and performs *shift* or *reduce* actions on the stream of tokens it parses according to the production rules. The specification defines production rules for language constructs such as statements and expressions. There is a conflict if these rules are ambiguous, and two or more rules can match a set of tokens simultaneously. *Reduce-reduce* conflicts occur when two rules match fully and either can be applied. *Shift-reduce* conflicts occur when one rule can be applied to perform a *reduce* action and another to perform a *shift* action. These conflicts must be avoided to have a well-functioning parser. If there are no parser rules that match a given set of tokens or if the source files contain strings that do not match any scanner rule, it is a syntax error.

2.1.2 Abstract Syntax Trees

The Abstract Syntax Tree (AST) is a tree created by the parser containing nodes representing language constructs in a source file. The AST representation allows analysis of a program to be written efficiently, e.g., in JastAdd, by declaring attributes for nodes. These attributes are described further in Section 2.3. Figure 2.1 shows an example of a part of an AST corresponding to the assignment statement `int y = 2;` and the arithmetic expression `1 + y * 3`. The Figure demonstrates how the AST nodes are related to tokens and groups of tokens in a program and how nodes can be connected through reference attributes, explained further in 2.3.1. In this example, a `VarAccess` node keeps track of where the variable is declared through the reference attribute `decl()`.

In JastAdd, the AST nodes are represented in Java classes that are defined in the abstract grammar files. These define the primary class hierarchy and which children AST nodes have. There are also several additional Java classes to represent lists and optional children. All AST nodes inherit from `ASTNode` and additional classes, e.g., `Expr` and `Stmt`, are introduced to make attributes reusable. All expressions inherit from `Expr` and similarly for other structures. In this thesis, the color notation of the `ASTNode` node means it is a Java object that is a node in the AST.

2.1.3 Semantic Analysis

Upon completion of syntactic analysis, the subsequent phase is semantic analysis of the parsed program. Here semantic errors such as type errors, variables missing declarations, and more are detected. This analysis is typically done on the AST created by the parser. Semantic analysis in ExtendJ is done mainly with JastAdd attributes.

2.1.4 Code Generation

The last step of the compilation is to generate the target code that can then be run. The code generation step is the main part of the back-end of a compiler, and it is common to have multiple back-ends so that target code for different target systems can be generated. For ExtendJ, the code generation is done by directly transforming the AST nodes into so-called Java bytecode [11] that can then be run on the *Java Virtual Machine* (JVM). The bytecode is stored in class files that can then be run directly, if they contain a main method, or are called by other class files. A benefit of Java is that only one back-end is needed with the JVM enabling the generated code to be run on different operative systems or computer architectures.

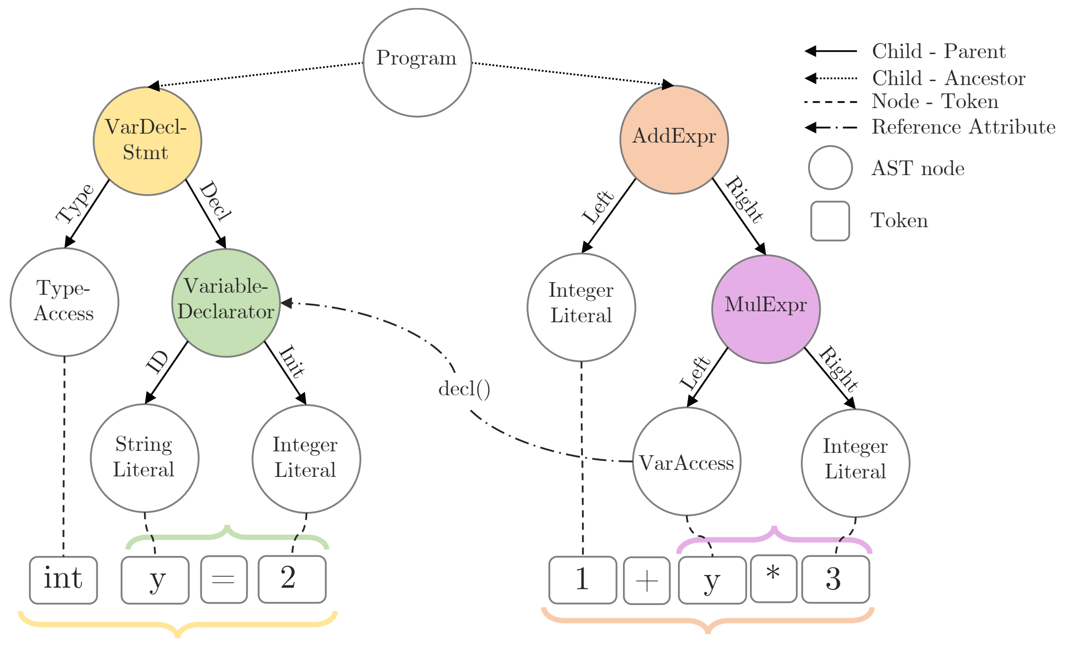


Figure 2.1: A simplified example of an AST showing how nodes relate to the source code and how the use of a variable is connected to the declaration of the variable through a reference attribute.

2.2 ExtendJ

ExtendJ is a Java compiler mainly built using the JastAdd metacompiler system [1] described in Section 2.3.2. The purpose of ExtendJ is to function as a regular Java compiler while having a modular design to make it possible to add extensions to the language. Functioning as a Java compiler means it creates class files from Java source code that can then be run by a regular Java run-time environment [1]. The main components of ExtendJ are the Java version modules consisting of a back-end and a front-end. These consist of *aspects* [12] that enables the modular definition of attributes for the AST nodes. Using aspects, new attributes or methods can be declared and existing functionality extended or overwritten.

The parser in ExtendJ is built using the Beaver [9] parser generator, and the scanner using the JFlex [8] scanner generator. These do not enable a modular design, but a preprocessor in JastAdd allows definitions to be split into different files. This enables the compiler to be separated into modules corresponding to a Java version. Using an appropriate build script, the modules can be combined, compiled, and packaged into a *jar* file to create a Java compiler for any supported Java version.

The latest significant extension of ExtendJ was with the *Extending JastAddJ to Java 8* [7] paper in 2014. ExtendJ currently supports almost all Java 8 features, and contains some bugs and minor issues, mostly related to type inference. Therefore, it can compile most, but not all, Java 8 projects[1].

[1]The compliance issues for ExtendJ are described further on the ExtendJ web page: `https://extendj.org/compliance.html`

```
syn Type Expr.type();
eq MinusExpr.type() = getOperand().type();
eq IntegerLiteral.type() = typeInt();
```

Listing 2.1: A synthesized attribute `type()` propagating type information upwards in the AST.

```
inh boolean BreakStmt.insideLoop();
eq Program.getChild().insideLoop() = false;
eq ForStmt.getStmt().insideLoop() = true;
eq WhileStmt.getStmt().insideLoop() = true;
```

Listing 2.2: An inherited attribute `insideLoop()` propagating information downwards in the AST.

2.3 Attribute Grammars

Attribute Grammars [13] (AGs) are a formalism for specifying the syntax and semantics of a programming language. This is done by adding *attributes* to the parsed tree of a string such as the AST parsed from a source file. These attributes are specified in a declarative way, which means that the rules to compute attributes are specified, but not the order in which they will be applied [14]. This enables new ways to solve problems by dividing them into attributes and declaring them individually [15].

We can distinguish two types of attributes: *synthesized* and *inherited*. Synthesized attributes are a way to propagate information upwards in the AST. They consist of a declaration and one or more equations for the node type or sub-types of it. The example in Listing 2.1 shows how type information for expressions can be calculated and propagated upwards. The `Expr` nodes need type information, and for this, we declare the `type()` attribute for `Expr`. To compute the type, we declare appropriate equations for sub-types of `Expr`, e.g., for `MinusExpr` and `IntegerLiteral`. The `type()` attribute can then be used to check type compatibility by comparing the expression type with the requirements of the context it is in.

To propagate information downwards in the AST, inherited attributes are needed. This allows nodes to find information about their surrounding context by accessing attributes in ancestor nodes. The attribute is computed by traversing the tree upwards until the first ancestor node that has a definition for the attribute is found. Listing 2.2 shows a simplified example from EXTENDJ where we compute if a `BreakStmt` is inside a loop. The attribute is declared in the `BreakStmt` but can only be computed by ancestor nodes since context is needed. If the `BreakStmt` is inside a loop, it will either be the full body or a child of the body of a `ForStmt` or `WhileStmt`. JASTADD will traverse the AST upwards until it finds a node that is the statement body of a `ForStmt` or `WhileStmt`, or until the root `Program` node is found.

```
syn nta TypeDecl Program.typeNull() {
    NullType classDecl = new NullType();
    return classDecl;
}
```

Listing 2.3: A declaration of a simple NTA `typeNull()` corresponding to the built in null-type.

2.3.1 Reference Attribute Grammars

Reference Attribute Grammars [16] (RAGs) were introduced as an extension of attribute grammars for object-oriented languages. AGs are limited when dealing with properties that are far away in the AST from where they are needed [16]. With RAGs, attributes can now be references to other tree nodes, allowing such properties to be computed efficiently. An example of this can be found in Listing 2.1, where the types themselves are AST nodes in the tree, either as built-in type nodes or class declaration nodes.

Reference attributes that become part of the AST when being evaluated are called *Non-Terminal Attributes* (NTA). The new NTA node becomes a child to the node where the attribute is defined. This means the value of the attribute is a new node that has the node defining it as its parent. The simplified example in Listing 2.3 demonstrates how an NTA is used in ExtendJ to create a null-type node. This is needed for all basic types since there are no classes defining them, but they still need to be represented by an AST node.

There are many systems that implement the RAGs formalism, e.g., Kiama [17], Silver [18], and JastAdd [2]. In this thesis we will focus on the JastAdd metacompiler.

2.3.2 The JastAdd Metacompiler

JastAdd is a system designed for constructing compilers and related tools in a modular way [2]. JastAdd is composed of a language and a compiler. JastAdd not only supports RAGs, but also additional attributes such as NTAs [19] and *circular* attributes [20, 21]. The circular attributes can depend on themselves, and have their value calculated by finding a fixed point through iteration.

All attributes, equations, fields, and methods for the AST nodes in JastAdd are part of an aspect. The JastAdd system reads the aspects and inserts the contents into the target AST classes. This allows additions to be made to the classes without needing to modify the class directly, making them easier to extend. Additionally, this allows the definition of classes to be divided into different aspects enabling a modular design.

Refining attributes

To further allow for modular extensions JastAdd facilitates a way to make changes to existing attributes and methods using the keyword *refine*. This allows new modules to change the behavior of the existing code without needing to modify it directly. An example of this

```
aspect Modifiers{
    syn boolean MethodDecl.isAbstract() =
       getModifiers().isAbstract() ||
       hostType().isInterfaceDecl();
 }

aspect Java8Modifiers{
    refine Modifiers
    eq MethodDecl.isAbstract() =
        getModifiers().isAbstract() ||
           (hostType().isInterfaceDecl() &&
           !isStatic() && !isDefault());
}
```

Listing 2.4: An original attribute and its refined counterpart.

can be seen in Listing 2.4, where `static` and `default` interface methods were introduced in Java 8. In the example, we can see how the original attribute `isAbstract()` for method declaration nodes, `MethodDecl`, in the aspect `Modifiers` is refined in the Java 8 aspect `Java8Modifiers`. Now the `isAbstract()` attribute can account for the new types of interface methods without requiring any changes to where the attribute is used. This allows new modules to redefine the implementation of older attributes meaning the older modules can be extended without being modified directly.

2.4 The Java Language

The Java programming language has been popular for many years and is used extensively. The language is constantly evolving, with new versions being released regularly. These versions come with new additions and changes to the language to keep it modern and relevant. This section will describe changes made to the language in Java 9, Java 10, and Java 11.

This thesis uses the Oracle Java SE binaries as a reference Java compiler and run-time environment. The official reference implementation of Java is the OpenJDK JVM [22], but production-ready binaries are available through Oracle or other vendors. For the remainder of this thesis, when referencing the javac compiler, we mean these compilers unless otherwise specified.

2.4.1 Java 9

Java 9 introduces six changes to the Java language, four of which are minor, one more substantial, and one requiring extensive changes to the OpenJDK JVM. The six changes are as follows:

- final and effectively final variables can be used as resources in the try-with-resources statement,
- the `@SafeVarargs` modifier can be used with private instance methods,
- the diamond operator can be used with anonymous classes if the argument type of the inferred type is denotable,
- remove underscore from the set of legal identifier names,
- interfaces can have private methods, and,
- the module system.

```java
try (Scanner scanner =
new Scanner(new File("in.txt"));
    PrintWriter writer =
    new PrintWriter(new File("out.txt"))) {
    /* use resources */
}
```

Listing 2.5: In Java 8 the resources need to be created in the try-with-resources statement.

```java
final Scanner scanner =
new Scanner(new File("in.txt"));
PrintWriter writer =
new PrintWriter(new File("out.txt"));
try (scanner; writer) { /* use resources */ }
```

Listing 2.6: Java 9 allows more compact try-with-resources statements. The variable `scanner` is final, and `writer` is effectively final.

Of the six changes, the module system is the largest and was part of the larger *Jigsaw Project* [23], which also introduced extensive changes to the internal structure of the JVM.

At its core, a module is a set of related packages that have been grouped together. Packages within modules may be classed as 'exported', meaning that their types may be accessed from outside the module. If packages are not exported, only other packages within the module may reach them [24, p. 175]. Adding support for this in ExtendJ would mean the module information files would need to be parsed and the contents incorporated into the name analysis. We decided it was not realistic to include support for modules in the scope of this thesis.

Of the five remaining features, the largest one is the update to the try-with-resources statement [24, p. 470-475]. The update means resources in the resource list can be declared outside of the try-statement if they are final or *effectively final*. This, in turn, can lead to more concise code. A variable is effectively final if its value is not changed after initialization. For objects, this means that the *reference* to the object is not changed, while the state of the object itself may be altered.

An example using try-with-resources is shown in Listing 2.5, and how it can be re-written using the changes introduced in Java 9 is shown in Listing 2.6.

```
ArrayList<String> list = new ArrayList<String>();
Stream<String> stream = list.stream();
```

Listing 2.7: In Java 9, all local variables must have an explicit type.

```
// infers type: ArrayList<String>
var list = new ArrayList<String>();
// infers type: Stream<String>
var stream = list.stream();
```

Listing 2.8: In Java 10, local variable types can be inferred using `var`.

2.4.2 Java 10

Java 10 introduces type inference for local variables with the `var` identifier. This decreases the amount of boilerplate code needed when declaring variables [25]. Using `var` can help make the code more readable and faster to write if used in the correct situations. The Java 10 specification states the following regarding `var`: *"var is not a keyword, but rather an identifier with special meaning as the type of a local variable declaration."* [26, p. 24]. This means the identifier is *context sensitive* and the rules for its used depends on how it is used. It can be used as normal for e.g., variable and method identifiers, but not as the identifier for classes or interfaces. The identifier also comes with several limitations on its use in declarations. It is a compile-time error if:

- `var` is used to declare more than one variable. E.g., `var x = 1, y = 2;`,
- `var` is used to declare a variable with bracket pairs. E.g., `var x[] = y;`,
- a variable declaration using `var` lacks an initializer. E.g., `var x;`
- a variable declaration using `var` has an array initializer. E.g., `var x = {1, 2};`, or,

- a variable declaration using `var` has an initializer containing a reference to itself. E.g., `var x = (x = 1);`.

An example of how `var` can be used is illustrated in Listings 2.7 and 2.8.

An interesting property of the `var` identifier, is that it allows us to capture certain non-denotable types as the type of our variable. Non-denotable types are types that cannot be written with the language syntax. In Java 10, these include intersection types, capture types, anonymous class types, and the null-type. Of these, intersection types and anonymous class types can be inferred as is, while the null-type is rejected. Capture types are treated specially, which is described in the following section. The effect of this is that there are programs that can be expressed with the help of the `var` identifier that cannot be expressed without it. A survey on the OpenJDK code base found that 1% of all variable declarations that have an initializer would contain a capture type if they were changed to a declaration using `var` [25].

Type Projections

Capture types may contain *synthetic type variables*, which are type variables introduced by the compiler during capture conversion or inference variable resolution [26]. The type of a variable declared with `var` may not contain these type variables according to the Java 10 specification [26, p. 76-78], and they are instead replaced by applying an *upward type projection* on the type. This projection is also described in the Java 10 specification. The type projection is always applied to the type of the initializer of a declaration using `var`. On types that are not or do not contain synthetic type variables, upward type projection acts as the identity function, while for others it finds a suitable replacement type. For parameterized types such as `Map<K, V>`, it replaces each type with its respective upward projection, and for arrays, it performs an upward projection on the base type.

```
lines.forEach( (line) ->
    {System.out.println(line);} );
```

Listing 2.9: In Java 10, the type of lambda parameters is either explicitly typed or there is no type declared.

```
lines.forEach( (var line) ->
    {System.out.println(line);} );
```

Listing 2.10: In Java 11, lambda parameters can be declared with the identifier var.

2.4.3 Java 11

In Java 11, the decision was made to allow the `var` identifier to also be used for implicitly typed lambda expressions. Type inference for lambda expressions is not new, but this change creates better uniformity in the language and enables the use of type annotations in a concise way [27]. Listing 2.9 demonstrates how the type of lambda parameters could be inferred in Java 10, and Listing 2.10 shows the additional way the types can be inferred using `var` in Java 11.

2.5 Evaluating Java Programs

Evaluating software can generally be divided into two main categories, correctness and performance. Since ExtendJ is complied into Java code and executed as a Java program, we present the background needed to evaluate Java programs. In Section 2.5.1 we describe how performance of a Java program can be measured and analyzed, and in Section 2.5.2 how a program's correctness can be evaluated.

2.5.1 Performance Evaluation

The performance of Java programs is complex to analyze through benchmarking since many factors can affect the results. One factor that makes Java stand out from many other languages is the Just-In-Time (JIT) compilation that optimizes the program during run-time. This means an initial run in a JVM may have significantly different performance than subsequent runs. Once a program has been executed a certain number of times within a JVM, its performance will stabilize due to the JIT compiler's limited optimization capacity, reaching a state called steady-state. Analyzing both steady-state and start-up performance is relevant since different use cases will be dependent more on one or the other. For a developer compiling a project the start-up performance will matter most since the project is typically only compiled once at a time. For an interactive code analysis tool that is constantly compiling code when changes are made, it is more likely the steady-state performance that matters more because the subsequent compilations can be performed in the same JVM.

This thesis uses the approach described in the paper *Statistically Rigorous Java Performance Evaluation* by A. Georges, D. Buytaert and L. Eeckhout [28]. The paper describes how to handle the non-determinism of Java performance, to avoid drawing incorrect conclusions from an evaluation.

When measuring start-up performance, Georges et al. recommend the following procedure. First, measure the performance of multiple VM invocations, where each invocation runs a single benchmark iteration. Once these measurements are obtained, take the mean, and from that value calculate the confidence interval for the desired confidence level. Note that this procedure assumes all measurements are independent of each other. For this reason, excluding the first VM invocation is good practice, as it may make system changes that persist past the first invocation [28].

Measuring steady-state performance is more complicated than measuring start-up performance, as there is a need to determine when a steady-state is reached. How many iterations it takes to reach such a steady-state can vary greatly depending on the application that is evaluated [28]. The general way to measure steady-state performance is to measure the performance of multiple iterations of a benchmark over multiple VM invocations. To determine when steady-state performance is reached, the *coefficient of variation* (CoV) of the last k iterations is used. Once the CoV falls below a predetermined value, the mean for that VM invocation using the measurements of the last k iterations is computed. These means are then used to calculate an overall mean, which is, in turn, used to compute the confidence interval for the desired confidence level, just as with start-up performance [28].

A confidence interval is an interval for which there is a given likelihood that the actual mean of the population is within the interval. As such, intervals will be larger for higher degrees of confidence, i.e., a 95% confidence interval will be larger than a 90% confidence interval. The way these confidence intervals are calculated depends on the number of samples. If the number of measurements is small ($n < 30$), the *Student's t-distribution* can be used. If the number of measurements is large ($n >= 30$), *Gaussian distribution* can be used instead [28].

Analysis of Variance (ANOVA) is a way to compare multiple alternatives, where one variable is altered between each alternative. The idea behind ANOVA is to compare the variation within an alternative to the variation between alternatives. ANOVA assumes that the variation within each alternative is due to random effects (errors) in the measurements. If the variation between alternatives is greater than the variation within them, then one can conclude that there is a statistically significant difference between them. In practice, this means that three values are computed: the sum-of-squares due to the difference between alternatives (SSA), the sum-of-squares due to errors between measurements (SSE), and the sum-of-squares total (SST), which is the sum of the SSA and SSE. A simple way to quantify whether the variation within or between alternatives is greater is to compare the fractions SSE/SST versus SSA/SST [28].

2.5.2 Precision of a Program

Evaluating program precision is a crucial process in software development that involves assessing the correctness, completeness, and quality of a computer program. It ensures that the software meets the specified requirements and performs its intended functions as expected, while also identifying and correcting any errors or defects that may affect its performance or reliability. By measuring a program's precision, developers can increase confidence in its correctness, reduce the risk of bugs and system failures, and improve the overall user experience. There are many ways to achieve this such as:

- manual inspection,
- program analysis,
- automated tests, and,
- execution on real world input.

Each method has its own benefits and drawbacks that need to be considered. Manual inspection is simple and allows a skilled developer to notice bugs early, but it is also a time-consuming process, and as such does not scale well with larger projects. Program analysis, on the other hand, allows the developer to make firmer statements regarding the completeness and correctness of their software. While this is the case, finding and adapting appropriate program analysis methods, while also ensuring that the theory behind the analysis is sound, is no simple matter. Automated testing and execution on real world input, in contrast to program analysis, are far more practical methods. One of the most common types of automated testing is unit testing, where each test tests a specific feature. Unit tests are simple to write, and can allow developers to cover a large part of the codebase. However, a limitation of unit testing is that it can only show that the program has the expected behavior for specific inputs. As such, executing the program on real world input can be a complementary step, as it may help in discovering hard to foresee edge cases.

It is important to note that no program testing method can guarantee that all bugs, or indeed any bugs, will be found. As Dijkstra stated in 1970 – "Program testing can be used to show the presence of bugs, but never to show their absence" [29].

Chapter 3
Implementation

This chapter describes the implementation of the Java 9, 10, and 11 features. The issues faced during implementation are also described to help answer the research questions.

3.1 Java 9 Implementation

Implementing the Java 9 features could be done modularly, except the addition of the enhanced try-with-resources (TWR) statement, that required changes to the previous implementation. The minor language changes introduced in Java 9 were simple to implement and, as such, will not be covered in detail. Focus will instead lie on the changes made to the TWR statement.

3.1.1 Try-with-resources

The extension to the TWR statement could not be done by adding a module. Adding a new production to the parser in the Java 9 module would create conflicts, but it was necessary to accommodate the use of resources in addition to declarations. This is due to the parser's limited knowledge, making it impossible for a new rule not to cause conflicts with the existing parser. A new parser rule would have to allow an arbitrary number of resource declarations and resource uses, but it cannot track whether at least one resource use was parsed. This means that this new rule would match a TWR statement only containing resource declarations. The old rule would also match this, creating a reduce-reduce conflict. Extending the current production was not possible either since the existing rules were not compatible with the extension.

There were two main ways of solving this limitation. The first solution was to rewrite the Java 7 implementation to make it more extendable and add a Java 9 module that extends it. The downside is that changing the Java 7 implementation might cause bugs or issues that were previously not there. The second solution was to exclude the Java 7 implementation from the build scripts and add a new implementation for TWR in Java 9. Doing this would cause significant code duplication since the new solution would still be based on the original one. We chose the first option since this makes the entire implementation easier to understand and more extendable in the future.

We tried two ways to change the Java 7 implementation to be more extendable. The first one was to wrap `ResourceDeclaration` in a statement node, `ResourceDeclStmt`. This node could later be extended to contain other forms of resources that are not declarations. This was fast and relatively easy to implement, but would have two negative effects on the compiler. One is memory usage, where we now have one more node for each resource with the only purpose of wrapping a resource node. The second is that the implementation added one attribute to all statement nodes, meaning further memory usage and unnecessary attributes for all other statement nodes.

The second method introduced an abstract node type, i.e., `Resource`, which inherits the type `Stmt`. `ResourceDeclaration` was then changed to be a subtype of `Resource`. Instead of inheriting from the type `VariableDeclarator`, `ResourceDeclaration` now contains one, refactoring the previous inheritance into a composition. This refractoring can be seen in Figure 3.1. With this new structure, it was easy to introduce a new subtype to `Resource` – `ResourceUse`. However, this required changing many attributes in the Java 7 module, where attributes that were previously part of `ResourceDeclaration` directly now are part of the `VariableDeclarator` contained within it.

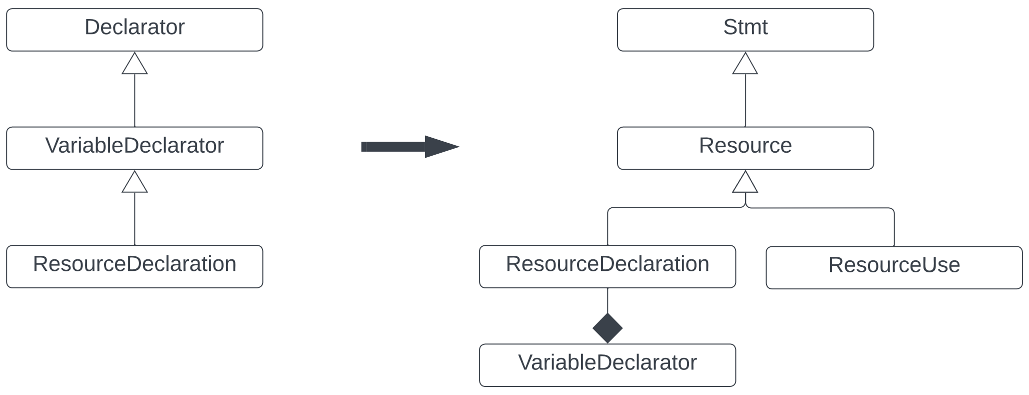


Figure 3.1: An illustration of the restructuring done in the Java 7 module, as well as how `ResourceUse` was added in Java 9.

A benefit of this approach was that it was more natural to construct parser rules. All that was needed was changing the type of the list of resources to be of the type `Resource` rather than the type `ResourceDeclaration` in the Java 7 module. Additional rules could then be added in the Java 9 module for parsing the new type of resource. Another benefit is potentially decreasing the extra memory needed. In the first solution, we created an additional node for each resource, but with this solution, there is one node created for each `ResourceDeclaration` and no additional nodes created for each `ResourceUse`. These two benefits caused us to select this way of changing the Java 7 implementation.

After implementing the changes to the Java 7 TWR module and adding the front-end for the Java 9 solution, the next step was deciding how to implement the bytecode generation. The Java 9 addition to TWR meant that the resource list could now contain uses of variables in addition to declarations of variables. What variables can be used as resources are limited to local or field variables that are final or effectively final. Two main ways of implementing this addition to the bytecode generation were explored. The first was following the way the Java 9 specification describes it by replacing the use of a variable in a resource list with a declaration of a temporary variable of the same type [24, p. 472], see Figure 3.2. This temporary variable requires a unique identifier and a transformation of the AST to insert its declaration in place of the original variable. All uses of the variable in the TWR statement would then need to be replaced by the new temporary variable. This would require several additions to ExtendJ, since there is no current support for creating unique identifiers for temporary variables and for replacing uses of a variable with the new temporary variable. The second was to ignore the Java 9 specification, keep the original variable, use it in the TWR block and then close it in a generated `finally` clause. This would only require minimal additions to the bytecode generation and would still follow the behavior defined by the Java 9 specification and was therefore chosen.

> If a basic try-with-resource statement is of the form:
>
> *try (VariableAccess ...) Block*
>
> then the resource is first converted to a local variable declaration by the following translation:
>
> *try (T #r = VariableAccess ...) { Block }*

Figure 3.2: Excerpt from the Java 9 specification [24, p. 472] describing how use of variables in a try-with-resources statement are transformed into a declaration of a temporary resource.

3.1.2 SafeVarargs

Allowing the `SafeVarargs` annotation to be used on private instance methods was possible to do in a modular way. The only modification needed was an extension to the error reporting system which was straightforward to implement. This was done with a `refine` of the `safeVarargsProblems()` attribute, adding a check to allow the use on private method declarations.

3.1.3 Private Interface Methods

The Java 9 addition of private interface methods could be implemented modularly by refining how the method declaration, i.e., `MethodDecl`, modifiers are determined and how the error reporting for the modifiers is done. A small addition to `ClassDecl` was also necessary to correctly determine if a class has unimplemented methods.

```
if(name().equals("_")){
    problems.add(error("As of release 9, '_' is a
       keyword, and may not be used as ..."));
}
```

Listing 3.1: The addition that was made to error reporting for all declarations, so that underscore may not be used as an identifier.

In addition to these changes, a minor change in bytecode generation had to be implemented. Since private methods were previously not allowed in interfaces, there was no code for handling private methods in interfaces when generating bytecode.

3.1.4 Remove Underscore as an Identifier

Preventing underscore from being used as an identifier was fast to implement and it could be done in a modular way. Existing name error checking for different types of identifiers could be refined to check for underscore identifiers and report them as errors. The way this was implemented was by adding the check in Listing 3.1 to the error reporting corresponding to declarations.

3.1.5 Diamond Operator

Java 9 allows the diamond operator to be used with anonymous classes if the inferred type is denotable. This introduced the need to check denotability in EXTENDJ. Denotable types are types one could find in Java code- i.e., built-in types, classes, and interfaces. Non-denotable types are all other types, such as `null` or anonymous classes.

The implementation of this addition is not complete, and when using the diamond operator in this context on an interface, EXTENDJ will crash. Using the diamond operator with interfaces causes the type of the anonymous class to be inferred incorrectly, and we did not manage to find the cause for this. The implemented changes could be made modularly without modifying the existing modules.

3.2 Java 10 Implementation

The only change in the Java language from version 9 to 10 was the addition of local type inference with the `var` type identifier. Implementing this language feature was in large parts straightforward, since the type inference system already in place could be reused.

3.2.1 Var Type Identifier

Since the usage of `var` is context sensitive, and only has special meaning as a type identifier, it could not be added as a keyword. As such, `var` was implemented by adding an attribute isVar() to the VarDeclStmt node and related nodes that returns **true** when the name of the declared type is "var". The attribute VarDeclStmt.type() was then refined to return the inferred type of the variable if it was declared using var, and the declared type if it was not. To find the inferred type another attribute was added to the VarDeclStmt node. This attribute was also added to EnhancedForStmt, i.e., **for each**-loops, since the loop variable can now also be declared using var. The value of the attribute will be the component type of an array being iterated over or the iterableElementType() of the collection being iterated over. This also needed a small addition to the code generation for EnhancedForStmt to use the inferred type when needed.

The var identifier has several restrictions that regular type identifiers do not have. For example, there are initializers that require an explicit target type, and are not allowed to be used in a declaration using var. Among these initializers are array literals as well as lambda expressions. If they are used in this context, then an error should be produced in accordance with the Java specification [26, p. 433 – 434]. Another restriction mentioned in Section 2.4.2 is that a variable declared with the var identifier cannot be referenced in its own initializer. To detect these occurrences, an attribute varOccurs() was added to all expression types that are valid initializers for the var identifier. This attribute returns **true** if the variable occurs in the initializer, and **false** otherwise. This check was then implemented for all Expr nodes where a variable use may occur, to identify when the variable being declared is used. Since many Expr nodes may contain other expressions, this check is done recursively on those Expr nodes.

As stated in Section 2.4.2, capture types should be projected to a supertype using upward type projection. An attempt was made to implement type projection by closely following the description found in the Java 10 language specification [26, p. 76-78]. However, this implementation attempt had to be abandoned. The most significant issue encountered was in verifying the correctness of the implementation. Writing tests where the inferred type of an expression contained relevant capture types proved difficult. To test type projections, capture types containing synthetic type variables are needed and without this, it became impossible to test the implementation. An extensive search for existing tests was carried out, but no tests were found. As such, the Java 10 implementation is not feature-complete.

3.3 Java 11 Implementation

The only new language addition in Java 11 was enabling the use of the var identifier in lambda expressions.

3.3.1 Var in Lambda Expressions

Implementing the Java 11 feature of allowing the parameters to lambda expressions to be declared with var could be done without modifying previous modules. What was needed was refining the type checking and type error reporting for lambda expressions as well as adding attributes to support this. The main part of this was giving the lambda parameters the type that can be inferred, instead of the declared type as parameters with a regular type identifier does. This was done with a new inherited attribute inferredType() for the lambda parameters, which is then used to set the type if the parameter is declared using var. The declaration

```
inh lazy TypeDecl
   ParameterDeclaration.inferredType();

refine TypeAnalysis
eq ParameterDeclaration.type() {
     if(getTypeAccess() != null &&
        getTypeAccess().isVar()){
         return inferredType();
     }
     return getTypeAccess().type();
}
```

Listing 3.2: How the correct type of a lambda parameter, i.e., `ParameterDeclaration`, is set in Java 11 using the new attribute `inferredType()`.

of the attribute and how it is used is shown in Listing 3.2. The value of `inferredType()` was calculated the same way as lambda parameters missing a type declaration in Java 8.

Chapter 4

Evaluation

To address the research questions, the extended ExtendJ compiler needed to be evaluated. To answer **RQ1** we examine the additions and if they could be made modularly in Section 4.1. In Section 4.2, we describe the problems encountered when compiling real-world projects. Finally, in Sections 4.3 and 4.4, we evaluate the precision and performance of ExtendJ to answer **RQ2**.

4.1 Extendibility

A majority, but not all the additions for Java 9, 10, and 11 could be done modularly. However, the change to try-with-resources could not be implemented without modifying the Java 7 implementation. This meant modifying seven different files and aspects relating to the parser, semantic analysis, and code generation. Two additional smaller changes needed to be made to the Java 8 implementation in the reading of bytecode and an aspect name to make it possible to extend it to Java 9.

4.1.1 Lines of code

A way of estimating the programming effort when extending a program is by counting the Source Lines Of Code (SLOC) of a program. For this, we used the SLOCCount program created by David A. Wheeler [30], which excludes comments and empty lines. The resulting SLOC counts for the ExtendJ modules 8-11 are shown in Table 4.1, along with the SLOC counts for the Java files generated for the whole compiler, including all previous modules. The reason for the small increase in generated code is because much of the the added modules consist of refining existing attributes.

Module	SLOC per Module	SLOC Compiled ExtendJ	Increase in Generated Code
Java 8	5419	142 003	19 507
Java 9	341	142 126	123
Java 10	400	142 757	631
Java 11	168	142 871	114

Table 4.1: Source Lines Of Code (SLOC) counts for ExtendJ 8-11 modules, SLOC counts for the Java code generated when compiling ExtendJ for the same module level, and the increase in the SLOC compared to the previous version of ExtendJ.

4.2 Compiling Real-World Projects

To evaluate the precision and performance of ExtendJ, we needed to compile real-world projects. We could not find any corpus with Java 9, 10, or 11 projects, so open-source projects were used instead. GitHub was mainly used for this, where many Java projects are available. The first step in the selection process was discarding projects requiring Java versions above Java 11 to compile. Then, we attempted to find two sets of projects, one set where the now implemented Java features are used and one which only requires Java 8 to build to enable comparisons between the ExtendJ versions 8-11 and Java versions 8-11. For some projects requiring Java 12 or above, we attempted to find older versions only needing version 11 or earlier.

Compiling Java projects is usually done using build systems such as Gradle, Maven or Ant using commands specified by the projects. Once a relevant project had been identified, it was downloaded and compiled with a compatible javac version to ensure all dependencies were available. This was not always successful as sometimes dependencies could not be found, or something would go wrong during compilation. One reason for this could be that not all projects specify what Java version is needed, and we used an incompatible version.

We employed three main approaches to ultimately compile projects with ExtendJ. However, we did not necessarily utilize all three methods at once; rather, we first attempted the initial approach, and only if it was unsuccessful, did we proceed to the next option. The first approach was to modify the build scripts to compile the project with ExtendJ instead of javac. The second was to find the classpath without using a build tool and compile the project manually, and the third was to substitute the javac executable with a script that runs ExtendJ. The resulting projects that could be compiled are shown in Table 4.2 with the corresponding version, SLOC count, description, and compilation method.

These projects together with the ExtendJ and javac executables used are available for download at `https://zenodo.org/record/7924893`. The scripts used to measure performance are also included together with instructions on how to run them.

4.2.1 Modifying Build Scripts

Most projects use build scripts for compilation and other tasks, and modifying them would be an efficient method to be able to compile many projects. For the Gradle build scripts we

```
task compileWithExtendJ(type: JavaExec) {
  classpath = files('extendj.jar')
  main = 'org.extendj.JavaCompiler'
  def files = sourceSets.main.java.srcDirs.collect {
     fileTree(it).matching { include '**/*.java'
     }.files }.flatten().collect { it.absolutePath }
  def runtimeClasspath =
     sourceSets.main.runtimeClasspath.asPath
  args = [
    '-nowarn',
    '-classpath',
    runtimeClasspath
  ] + files
}

compileWithExtendJ { dependsOn compileJava }
```

Listing 4.1: The Gradle task used to compile projects using ExtendJ.

were able to create the task shown in Listing 4.1. The task would run the ExtendJ compiler with the same classpath and input files used when compiling the project with javac. This was used to compile the ExtendJ and Disruptor projects with ExtendJ, but not all Gradle scripts have the same structure and could not be compiled in this way. We attempted to write similar tasks for Ant and Maven build scripts, but were unsuccessful. The main issue was determining the classpath and the set of source files.

4.2.2 Manual Compilation

To compile the projects without a build script we needed to construct the classpath and the set of Java files to compile. One way we tried doing this was by finding the classpath for an entry point of a project (e.g., class containing a main method). This means all files needed to run that entry point are compiled, and its main method can be run. A downside of this is that there might be multiple entry points in a project, and they might each depend on only a subset of the full project. An example of this is Java libraries that might have no main entry point and instead contain several modules. The reason this method was tried was that the classpath for one Java file could be found using the static analysis tool IntraJ that is based on IntraCFG [3]. Part of IntraJ's output when analyzing a Java source file is the classpath needed to compile it, and with this we were able to compile entry points for Jython and a few other projects. Because of the limitations of this approach, we prioritized compiling projects through the build scripts. This was the only method we were able to compile Jython with, however, it had to be excluded when measuring performance. This was because only the entry point was recompiled when measuring steady-state performance, as opposed to the whole project.

A modified version of this approach involves constructing the classpath for the whole project and compiling all relevant Java files together with the needed dependencies. Constructing the classpath for an entire project is complex, and we could not find any sources describing this process. However, there exists a small corpus of Java projects needing at most Java 8 to compile, as well as defined classpaths and dependencies. This allowed us to compile Fop, Antlr, Gson and Mockito, but was not helpful in finding any Java 9, 10, and 11 projects. Despite this, these projects were useful for comparing the performance of different ExtendJ versions with the corresponding javac versions.

4.2.3 Replacing the javac Executable

The final approach we tried to compile projects with ExtendJ was to replace the javac executable with a script that forwards the arguments to ExtendJ. The idea was that all build scripts could be redirected to use ExtendJ instead of javac by replacing the actual executable. Replacing javac with a shell script that calls ExtendJ was straight-forward, but getting the build scripts to use it was not. The default behavior of the Ant, Gradle and Maven scripts we tried was to run the Java compiler from within a JVM. The Java compiler is called through the standard library instead of the javac executable and changing the executable has no effect on the compilation. Calling the compiler in this way is how steady-state performance can be measured, since repeatedly calling the executable creates a new JVM instance each time and no steady-state can be reached. This issue was partially overcome by again modifying the build scripts to run the javac executable in a new JVM. We were able to do this to the Gradle build scripts in combination with adding parsing of the arguments in the executable to make them compatible with ExtendJ.

However, for Ant and Maven build scripts, we were not able to make any similar changes successfully, meaning this method was not the general method we hoped. Since the projects we could compile with this method were the same as the ones where we modified only the build script, we used that method for the evaluation.

Project	**Version**	**kSLOC**	**Description**	**Compilation Method**
Disruptor	4.0.0	18	Messaging library	Modified build script
Mockito	4.5.1	20	Mock objects for Java	Pre-built classpath
Gson	2.9.0	23	JSON parser	Pre-built classpath
Antlr	2.7.2	35	Parser generator	Pre-built classpath
Jython	2.2	88	Java Python implementation	Manual compilation
Fop	0.95	106	XML to PDF library	Pre-built classpath
ExtendJ	11	143	Java compiler	Modified build script

Table 4.2: Projects that were successfully compiled with ExtendJ and how they were compiled.

4.3 Precision of Implementation

The precision of the implementation was evaluated in two main ways, regression tests and compilation of real-world projects. Using regression tests was a clear choice since ExtendJ already had an extensive and up-to-date regression test suite that could be extended. Regression tests can show that the behavior is as expected for a specific data set curated by the developers while the real-world projects can highlight issues the developers could not foresee and write tests for.

4.3.1 Regression Tests

When writing regression tests, two important aspects are the run-time of the test-suite and the code coverage of the tests. Currently, there is no automated system running regression tests for ExtendJ, meaning tests needs to be run manually and the number of tests needs to be kept manageable. The number of tests for ExtendJ 8 is currently around 1800, and for the 9, 10, and 11 modules, we have added 48, 44 and 16 tests, respectively. These tests were helpful during initial development and for discovering flaws, but they cannot guarantee the implementation is correct. Compilers are complex programs with an infinite set of inputs making it impossible to efficiently measure precision of a compiler with only regression tests. However, since there were issues finding relevant real-world projects for this, extra care was put into ensuring the regression test suite was as comprehensive as possible. This was done by attempting to reach full code coverage of the new features and identifying potential edge cases.

4.3.2 Precision on Real-World Projects

Compiling large real-world projects using Java 9, 10, and 11 features with the corresponding ExtendJ versions would ensure the implementation is correct for practical use. To find what language features are used in the projects we were able to compile, we used the JFeature analysis tool [5]. It is built using ExtendJ to parse Java programs and works by using JastAdd `collection` attributes to gather information about the AST. Due to the modular design of JFeature and RAGs it was simple to add extra collection attributes for the new language features. The results of JFeature can be seen in Table 4.3, where all projects and their number of Java 8-11 features are shown.

4.4 Performance

To compare the extended version of ExtendJ with the javac compiler we measured the compilation time and memory usage of the projects we could compile. Compilation time is important during development since a slow compiler can be both frustrating and make development take more time. Memory usage is also an important metric, since ExtendJ needs to be able to compile large projects without requiring special hardware.

Compilation time and memory usage are the two performance metrics we were able to compare for ExtendJ and OpenJDK based Java compilers. All the following measurements

Project	**Java 8**							**Java 9**				**Java 10**		**Java 11**
	ConstructorReference	ConstructorReferenceAccess	IntersectionCastExpr	Lambda	MethodReference	MethodReferenceAccess	DefaultMethod	DiamondAccess	ResourceUse	Private interface method	Safe Varargs on private	Var	For each var	Lambda var
Disruptor	17	0	0	118	18	0	5	1	0	0	0	0	0	0
Mockito	3	0	0	37	13	0	10	0	0	0	0	0	0	0
Gson	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Antlr	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Jython	0	0	0	2	0	0	0	0	0	0	0	0	0	0
Fop	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ExtendJ	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.3: Analysis of compiled projects using JFeature to identify which Java language features introduced in Java 8-11 are used.

were performed on a benchmark computer running Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-70-generic x86_64) running on an Intel i7-11700K with 8 cores and fixed 3.5 GHz clock frequency. The computer has 128 GiB DDR4-3200 RAM with a 1 TiB M.2 harddisk. When measuring performance, the procedures described in Section 2.5.1 were used.

We could not run any javac 10 version on the benchmark computer which means we could not collect any performance data for javac 10. Also, note that there is no data for compiling Disruptor using javac 8 and ExtendJ 8 since the project requires javac 9 or ExtendJ 9 to compile.

4.4.1 Compilation Time

To evaluate the compilation time, we used the method described by A. Georges, D. Buytaert and L. Eeckhout [28], which we have outlined in Section 2.5.1. In this evaluation, we compiled each project 16 times in a fresh JVM, and for each JVM, we compiled up to 30 times or when a coefficient of variation (CoV) threshold of 0.05 for the last 15 runs was reached. The javac compilers used in the comparison are the included compilers from Oracle JDK 1.8.0.351, 9.0.4, 10.0.2 and 11.0.17. Because of the large spread in project size the results are divided up into two graphs. Figure 4.1 contains the compilation time for all projects excluding the large ExtendJ and Fop projects and Figure 4.2 contains the results of ExtendJ, Fop and Antlr, the largest project in the first graph, to put the results from the large projects into perspective.

The ANOVA method described in 2.5.1 was used to make it possible to determine if there is a statistically significant difference in compilation time for the ExtendJ compilers. The results are shown in Table 4.4. We also calculated the 95% confidence intervals of the data sets, and all lie within 7% of the mean compilation times.

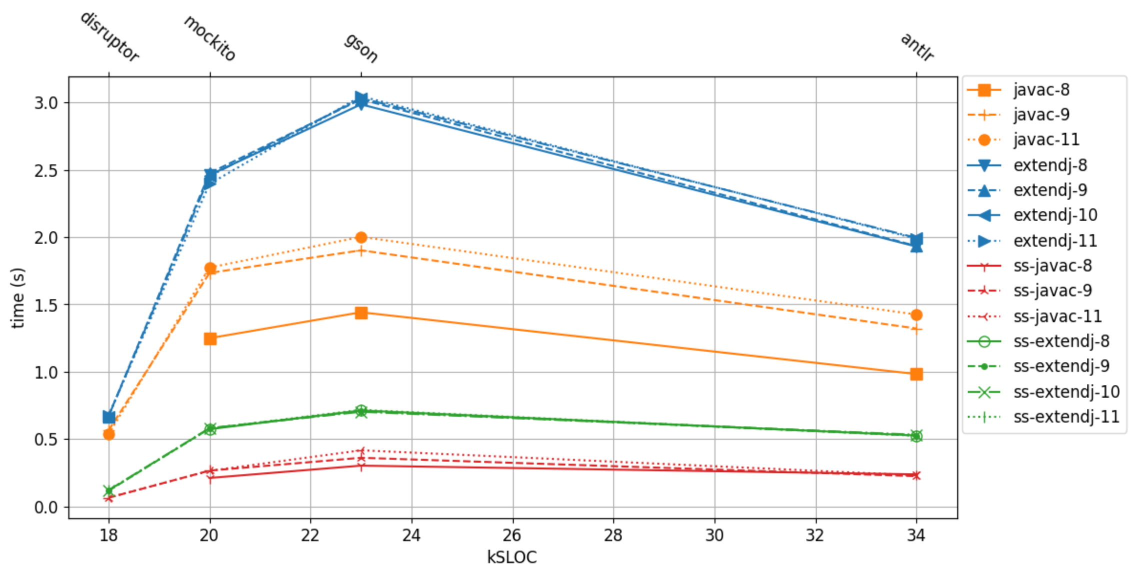


Figure 4.1: Compilation times for all projects, excluding ExtendJ and Fop, when using different compilers and compiler versions. The prefix *ss* stands for steady-state.

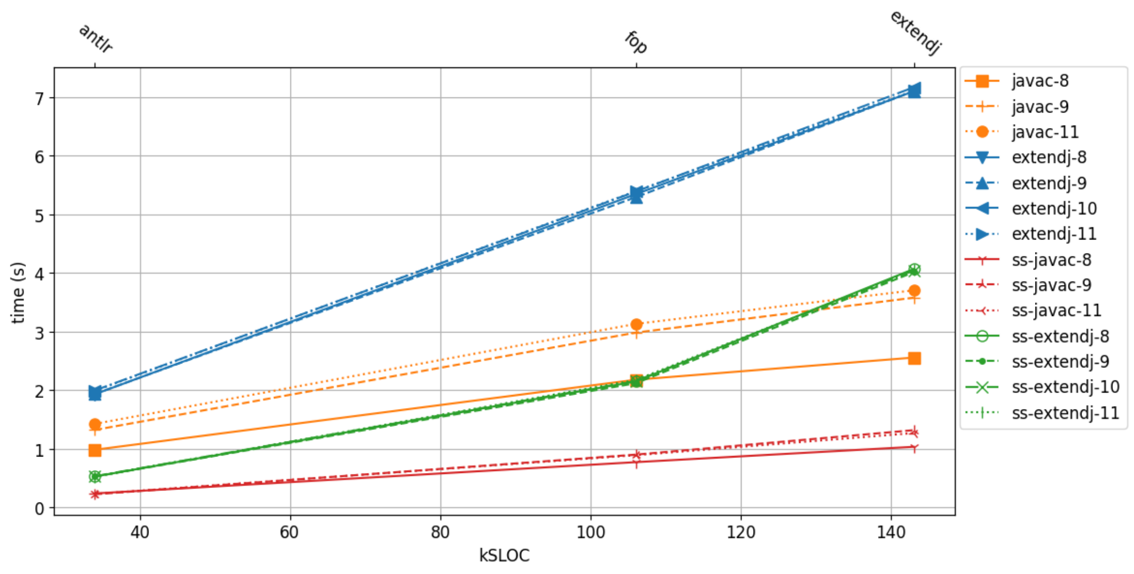


Figure 4.2: Compilation times for the three largest projects when using different compilers and compiler versions. The prefix *ss* stands for steady-state.

4.4.2 Memory Usage

To determine the ExtendJ memory consumption, the memory in use after compiling each project was measured 30 times, in a fresh JVM each time. Memory in use was calculated using the code in Listing 4.2 that takes the total memory the JVM currently has access to minus the

Project	Time – Start-Up	Time – Steady-State	Memory Usage
Disruptor	0.980	0.877	0.987
Mockito	0.848	0.984	0.981
Gson	0.952	0.912	0.988
Antlr	0.858	0.954	0.965
Fop	0.908	0.822	0.974
ExtendJ	0.974	0.958	0.964

Table 4.4: ANOVA results (SSE/SST) for the ExtendJ compilers' compilation time and memory usage. A value over 0.5 means there is no statistically significant difference between the data sets.

```
long inUse = Runtime.getRuntime().totalMemory() -
    Runtime.getRuntime().freeMemory();
```

Listing 4.2: How memory in use was calculated.

amount of memory that is not currently allocated.

The results of this can be seen in Figure 4.3 where the mean memory usage for the projects when compiled with the different compilers can be seen. To be able to discuss the increase in memory usage between javac and ExtendJ we identified the proportionally largest increase for the projects. This was for Gson when comparing javac 9 and ExtendJ 10 and the increase was with a factor of 5.56.

To test if there is a statistically significant difference of the memory usage between the different ExtendJ versions we again used the ANOVA method. The results of this are shown in Table 4.4.

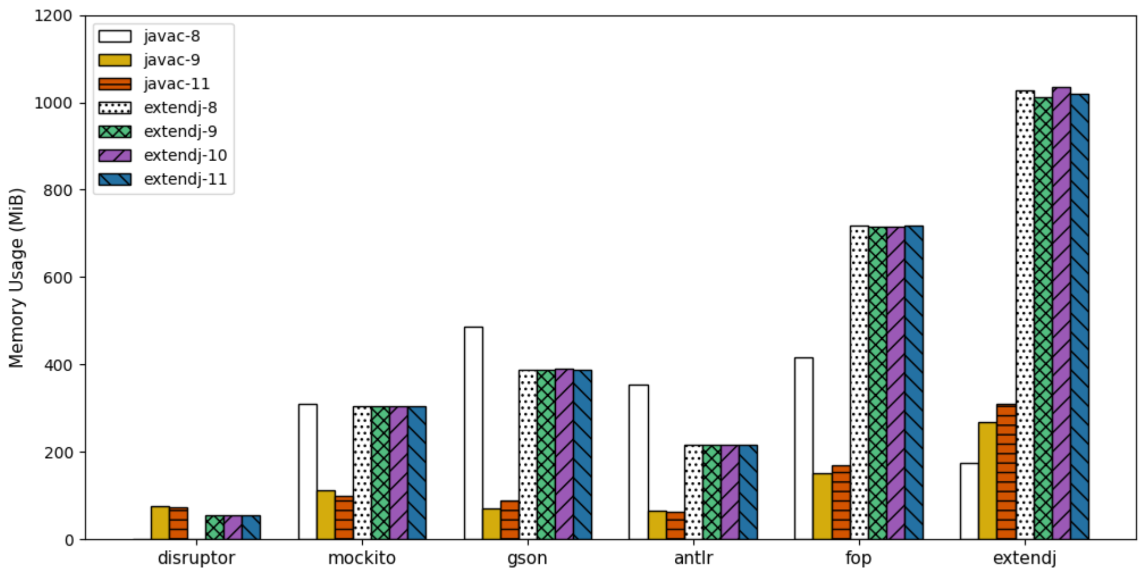


Figure 4.3: Memory in use after compiling the projects, ordered left to right by increasing SLOC count.

Chapter 5
Discussion

In this thesis, we have identified and implemented most of the Java 9, 10, and 11 features in ExtendJ and attempted to evaluate the implementation. During this, we realized the complexity of evaluating a Java compiler on real-world projects, and explored possible paths to accomplish this. In this chapter, we first discuss the implementation aspects in Section 5.1. Then, we discuss the difficulties and insights gained from the attempts to compile real-world projects in Section 5.2. In Section 5.3, we discuss the performance of ExtendJ and if it affects the usability of the compiler. Finally, we discuss future work on ExtendJ in Section 5.4.

5.1 Implementation

During the implementation, we aimed to evaluate how modularly extensible ExtendJ is. As described in Section 4.1, most new features could be added modularly and in only 909 lines of source code divided between the Java 9, 10, and 11 modules. This is a clear indication that the design of ExtendJ is modular and allows for new modules to be added efficiently. The RAGs enable the extensions to be made this modularly and in so few SLOC. The attributes make it possible to identify what data is available from a node and use that to efficiently define new attributes. The main feature that could not be added modularly was the extension to the try-with-resources statement. Java features have generally been implemented with a high level of abstraction in ExtendJ to allow for extensions, but this was not the case for the implementation of try-with-resources. With better foresight the design could have been made extendable from the beginning.

Another important metric is that two graduate-level developers, with the help of a supervisor, were able to understand, modify and extend the compiler in the limited scope of a master's thesis. We had previously used the JastAdd system, but had never worked with the ExtendJ compiler before starting the thesis. However, the implementation of the new features is not fully compliant and there are limitations to it, such as the lack of type projections. There are also likely more issues that have not been identified due to the lack of real-world projects to test it on. The limitations of the precision evaluation are clear from Table 4.3 with the almost complete absence of Java 9-11 features.

5.2 Difficulties Evaluating

The main issue we faced was building real-world projects to evaluate the performance and precision of ExtendJ. Despite the significant time spent on finding and compiling Java 9, 10, and 11 projects, we had little success, only being able to compile Disruptor. This project only had a single instance of a Java 9 feature, making it mostly useless for testing the new features. We were able to find one smaller project that used more of the new features, but because of the issues with type inference in ExtendJ, we were not able to compile it.

Similarly to how we could not evaluate the performance of the new features, we could also not test that the new features had been implemented correctly. We could however confirm that we were still able to compile projects that do not use the new features. We found no issues compiling projects with ExtendJ 9, 10, and 11 that we could also compile with ExtendJ 8. The only exception was the removal of the underscore identifier in Java 9, which meant some projects could not be compiled with ExtendJ 9-11 without changing the identifiers first. We did, however, find and report bugs existing in ExtendJ 8 when writing regression tests aimed at testing the new Java features.

Despite not adding support for modules, we did not encounter any significant issues due to this when compiling projects. For the few projects using Java 9, 10, and 11 features, we were able to compile with javac without using the build script, modules were not an issue when compiling with ExtendJ. The Disruptor project is one example of this since it uses modules, but could still be compiled both with javac 11 and ExtendJ 11 after removing the use of modules.

5.3 Performance

The performance of the extended compiler when the new features are not used, does likely not represent the performance when the features are used. However, it is still possible to discuss the overhead of the added features. None of the added features add any significant number of attributes, or what we estimate to be computationally or memory intensive operations. Considering that the added features are not used for most of the projects in the analysis, it should be expected that the compilation time and memory usage only increase slightly when adding support for each new Java version. This increase would be because of new attributes and the additions to existing attributes.

The ANOVA results from the performance analysis in Section 4.4 show no difference in compilation time when compiling the projects with ExtendJ 8, 9, 10, or 11. This means there is no significant compilation time overhead when compiling projects with the different ExtendJ versions. However, this can only be stated for projects that do not use the features introduced in Java 9-11.

For all projects, excluding ExtendJ, the compilation time slow-down was below a factor of 3, for both steady-state and start-up results. However, for ExtendJ, the slow-down was a factor of 4.0 for steady-state when comparing ExtendJ 8 with javac 8 and at most a factor of 3.1 when comparing the other compiler versions. Since ExtendJ is the largest project, the increasing slow-down factor is concerning for the performance of very large projects. However, ExtendJ is an outlier and testing this further on projects with SLOC counts at or above that of ExtendJ is necessary to speak more conclusively on the matter. A slow-down factor of around 3 should not limit the usability of ExtendJ on modern computers, but would make it less viable for performance critical applications.

There are two main points of discussion concerning the memory consumption evaluation. The first one is that the ANOVA results presented in Table 4.4 show no statistically significant difference between the ExtendJ versions when compiling any project. This means that it is likely that there is no significant increase in memory usage overhead, despite the added features. The second one is that the largest difference in memory usage between javac and ExtendJ was for javac 9 and ExtendJ 10 with a factor of below 6 (5.56) for Gson. A memory increase by a factor of 6 is concerning and could potentially limit the viability of ExtendJ for some use cases. When using ExtendJ as a normal Java compiler this should not be a limiting factor, since even the large projects compiled in this thesis have a memory usage in a reasonable range. However, when using ExtendJ in a tool such as IntraJ, where the compilation is done in the background to continuously analyze the code, the memory difference between javac and ExtendJ could be more concerning.

Another thing of note that can be seen in Figure 4.3 is the gap in memory usage between javac 8 and javac 9. We speculate that this is due to several memory optimizations between the release of Java 8 and Java 9. The difference in performance between javac 8 and 9 can also be seen in Figures 4.1 and 4.2, where the start-up compilation time for javac 8 is seemingly faster than for its counterparts.

5.4 Future work

There is ongoing work on improving the type inference for ExtendJ, which is the main limitation of the compiler. Finishing this could potentially make it possible to add type projections to ExtendJ 10 and 11 since they depend on very exact types.

To fully evaluate the precision of ExtendJ 9, 10, and 11 they need to be used to compile real-world projects using the implemented Java features. This would likely find issues with the implementation that have not been identified with the regression tests. For ExtendJ 9 to be compliant with the Java specification support for modules needs to be added. The remaining issue for the diamond operator described in Section 3.1.5 also needs to be addressed, which might depend on improved type inference.

For further development on ExtendJ and other Java compilers, a framework for testing would be helpful. This could consist of a corpus of Java projects combined with information about what Java features are used, similar to Table 4.3 as well as a way to compile them. If the framework could enable testing of all features and performance measurements when compiling projects using Java features from different Java versions, it could simplify development on Java compilers like ExtendJ.

Chapter 6
Conclusions

In this chapter, we answer the research questions based on the discussions in Chapter 5. The research questions are re-introduced to help readability.

RQ1: How modularly extensible is the ExtendJ Java compiler?
RQ2: What are the limitations in performance and precision compared to the javac compiler?

RQ1

The main feature of ExtendJ is its extendibility and modularity. To evaluate this, we have extended ExtendJ to support the majority of Java 9, 10, and 11 features. The new features could be implemented mostly modularly, with one notable exception being due to the limiting way the feature was originally implemented.

In addition to the modularity, the extensions could be made concisely requiring only a total of 909 lines of code. Considering this and the fact that ExtendJ could be extended with multiple features to support a large part of Java 9-11 within the scope of a master's thesis, we conclude that ExtendJ is extensible to a high degree.

Since the extensions could be made modularly and that we found ExtendJ to be extensible to a high degree, we conclude that ExtendJ is modularly extensible to a high degree.

RQ2

We have measured the performance of the extended ExtendJ compiler to see if it performs reasonably well compared to the reference Java implementation. Regarding the performance limitations for the extended ExtendJ compiler, we can only draw conclusions about the overhead of the added features, since finding and compiling projects using the features were mostly unsuccessful. For compilation time, we can conclude that ExtendJ generally runs within a factor of 3 both when measuring steady-state and start-up performance, but this might be increased for larger projects and projects using the Java 9-11 features.

We have also measured memory usage and conclude that ExtendJ's memory usage is up to a factor of 6 greater than the reference implementation. The higher memory usage and compilation time are significant, but should not be a limiting factor on modern systems, as even large projects can be compiled. It may, however, impact ExtendJ's usefulness as a static analysis tool, but no definitive statements can be made concerning this.

References

[1] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. *OOPSLA 2007,Montreal, Canada,– ACM Sigplan Notices*, 42:1 – 17, 2007.

[2] Torbjörn Ekman and Görel Hedin. The JastAdd system — modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14 – 26, 2007.

[3] Idriss Riouak, Christoph Reichenbach, Gorel Hedin, and Niklas Fors. A Precise Framework for Source-Level Control-Flow Analysis. *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), Source Code Analysis and Manipulation (SCAM), 2021 IEEE 21st International Working Conference on, SCAM*, pages 1 – 11, 2021.

[4] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of nonnull types for Java. *Journal of Object Technology*, 6(9):455 – 475, 2007.

[5] Idriss Riouak, Gorel Hedin, Christoph Reichenbach, and Niklas Fors. JFeature: Know Your Corpus. *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), Source Code Analysis and Manipulation (SCAM), 2022 IEEE 22nd International Working Conference on, SCAM*, pages 236 – 241, 2022.

[6] Jesper Öqvist. *Implementation of Java 7 features in an extensible compiler.* LU-CS-EX: 2012:13. Department of Computer Science, Faculty of Engineering, LTH, Lund University, 2012.

[7] Erik Hogeman. *Extending JastAddJ to Java 8.* LU-CS-EX: 2014:14. Department of Computer Science, Faculty of Engineering, LTH, Lund University, 2014.

[8] JFlex. `https://jflex.de/`[Online; accessed 19 April 2023].

[9] Beaver. `https://beaver.sourceforge.net/`[Online; accessed 19 April 2023].

[10] Jesper Öqvist and Görel Hedin. Extending the JastAdd Extensible Java Compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, page 147–152, New York, NY, USA, 2013. Association for Computing Machinery.

[11] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java® Virtual Machine Specification Java SE 9 Edition, 2017. `https://docs.oracle.com/javase/specs/jvms/se9/jvms9.pdf` [Online; accessed 2 March 2023].

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings 15*, pages 327–354. Springer, 2001.

[13] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[14] Didier Parigot, Gilles Roussel, Etienne Duris, and Martin Jourdan. *Attribute grammars: a declarative functional language*. PhD thesis, INRIA, 1995.

[15] Görel Hedin. An introductory tutorial on JastAdd attribute grammars. *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, pages 166–200, 2011.

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

[17] Anthony M Sloane, Lennart CL Kats, and Eelco Visser. A pure embedding of attribute grammars. *Science of Computer Programming*, 78(10):1752–1769, 2013.

[18] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1):39–54, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[19] Harald H Vogt, S Doaitse Swierstra, and Matthijs F Kuiper. Higher order attribute grammars. *ACM SIGPLAN Notices*, 24(7):131–145, 1989.

[20] Eva Magnusson and Görel Hedin. Circular reference attributed grammars—their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, 2007.

[21] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *ACM SIGPLAN Notices*, 21(7):85–98, 1986.

[22] OpenJDK. `https://openjdk.org/`[Online; accessed 19 April 2023].

[23] Project Jigsaw. `https://openjdk.org/projects/jigsaw/`[Online; accessed 20 April 2023].

[24] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java® Language Specification Java SE 9 Edition, 2017. `https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf` [Online; accessed 2 March 2023].

[25] Brian Goetz. JEP 286: Local-Variable Type Inference, 2016. `https://openjdk.org/jeps/286` [Online; accessed 16 February 2023].

[26] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java® Language Specification Java SE 10 Edition, 2018. `https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf` [Online; accessed 2 March 2023].

[27] Brian Goetz. JEP 323: Local-Variable Syntax for Lambda Parameters, 2017. `https://openjdk.org/jeps/323` [Online; accessed 16 February 2023].

[28] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 57–76 – 76, Department of Electronics and Information Systems, Ghent University, 2007.

[29] Edsger Wybe Dijkstra. Notes on Structured Programming, 1970.

[30] David A Wheeler. SLOCCount. `https://dwheeler.com/sloccount/`[Online; accessed 18 April 2023].

EXAMENSARBETE Extending the ExtendJ Java Compiler
STUDENTER Johannes Aronsson och David Björk
HANDLEDARE Idriss Riouak (LTH)
EXAMINATOR Görel Hedin (LTH)

Kan man utöka en Java-kompilator?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Johannes Aronsson och David Björk**

En kompilator är ett program som kan översätta kod som är enkel att skriva och förstå till kod som är effektiv att köra. ExtendJ är en Java-kompilator som är specifikt designad för att vara enkel att utöka och denna uppsats testar detta genom att utöka den och utvärdera dess prestanda.

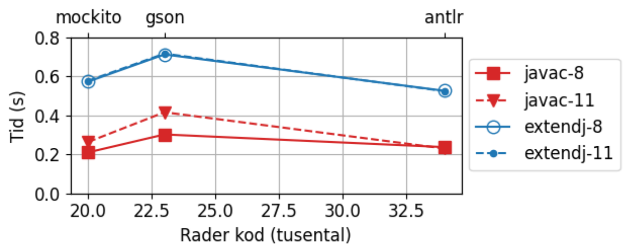
Kompilatorer är viktiga inom mjukvaruutveckling. De gör det möjligt för programmerare att skriva läsbar kod som sedan kompileras till ett format som gör det möjligt och snabbt för en processor att köra programmet. Java är ett modernt programmeringsspråk och Java-kompilatorer som javac är komplexa och svåra att modifiera.

För att utforska nya sätt att utveckla kompilatorer skapades ExtendJ Java-kompilatorn som är skriven på ett utökningsbart sätt. ExtendJ stödjer just nu Java 4–8 och i detta arbete hade vi som mål att utöka kompilatorn till Java 9–11. Detta för att utvärdera om den är utökningsbar och hur kompilatorns prestanda påverkas av utökningar. Det finns idag flera forskningsprojekt som använder sig av ExtendJ, och en utökning av kompilatorn skulle betyda att dessa blir mer aktuella.

Vi byggde ut ExtendJ och analyserade hur korrekt den fungerar samt hur dess prestanda är jämfört med javac. Då kompilatorer är komplexa program så måste de testas på existerande projekt. Om ExtendJ kan kompilera projekt som använder sig av ändringarna i Java 9–11 så är den till stor del korrekt. Även prestandan måste mätas på existerande projekt för att kunna dra relevanta slutsatser.

En stor del av arbetet lades på att hitta och kompilera projekt som använder tilläggen i Java 9–11. Sådana projekt kunde inte hittas och därför fick tilläggen testas på Java 4–8 projekt. Vidare arbete med detta behövs för att kunna analysera ExtendJ ordentligt.

Vi drog tre slutsatser från undersökningen. Den första var att ExtendJ är i hög grad utökningsbar. Den andra var att kompileringstiden och minnesanvändningen för den utökade kompilatorn är oförändrad för Java 4–8 projekt. I Figur 1 visas kompileringstiden för tre av de projekt vi kunde kompilera. Slutligen drog vi slutsatsen att kompileringstiden är som mest tre gånger längre och minnesanvändningen sex gånger högre än för javac.



Figur 1: Hur kompileringstiden för tre projekt beror av vilken kompilator och version som används.