Extending JastAddJ to Java 8

Erik Hogeman

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2014-14
Extending JastAddJ to Java 8

Erik Hogeman

June 8, 2014

Abstract

JastAddJ is an extensible Java 7 compiler built using the meta-compilation system JastAdd. One of the main features of JastAddJ is that it supports modular extensions, making it well suited to use as a base compiler when implementing various types of language extensions or new languages built on top of Java. This thesis examines the possibility of extending JastAddJ with a module supporting the latest Java release, Java 8.

The features introduced in Java 8 include anonymous methods, or lambdas, and a way to refer to already existing methods. To facilitate the use of these features, other updates to the language include new interface methods, intersection type casts and an update to the Java overload resolution algorithm. An implementation of most of these features is presented and evaluated with respect to completeness, modularity, code size, compilation time and memory usage. An extensive unit testing suite was also constructed and evaluated.
Acknowledgements

I would like to thank my examiner Görel Hedin and my supervisor Jesper Öqvist for their valuable input, both on the practical implementation and the writing of the thesis.
## Contents

1 Introduction 6

2 Background 7
  2.1 Graph Notation ................................. 7
  2.2 Compiler Basics ................................. 8
    2.2.1 Scanning ................................. 8
    2.2.2 Parsing ................................. 9
    2.2.3 Semantic Analysis ......................... 10
    2.2.4 Code Generation ............................ 10
  2.3 JastAdd ........................................ 10
  2.4 The Java 8 Language Constructs ............... 11
    2.4.1 Functional Interfaces ..................... 11
    2.4.2 Lambda Expressions ....................... 12
    2.4.3 Method and Constructor References ........ 13
    2.4.4 Interface Methods .......................... 13

3 Scanning and Parsing Java 8 14
  3.1 Lambda Expressions .............................. 14
    3.1.1 Handling the Ambiguities With a LALR(1) Parser Generator ...................... 17
  3.2 Method and Constructor References ............. 19
  3.3 Intersection Type Casts ........................ 21
  3.4 Default Methods ................................ 22
  3.5 Package Modifier ................................ 23
  3.6 Testing the Parser ................................ 23
    3.6.1 Syntax Error Check .......................... 24
    3.6.2 Parse Tree Check ............................ 24
    3.6.3 Structured Print Check ...................... 24

4 Functional Interfaces 24
  4.1 Semantic Rules .................................. 25
    4.1.1 Subsignatures ................................ 25
    4.1.2 Checking Return-Type-Substitutability ........ 27
    4.1.3 Function Type ................................ 27
  4.2 Implementation of Semantics Checking .......... 28
    4.2.1 Subsignature Analysis ........................ 28
    4.2.2 Return-Type-Substitutability Analysis ........ 29
    4.2.3 Computing the Function Type ................. 31

5 Contexts and Target Types 32
  5.1 Assignment Contexts ............................. 33
  5.2 Invocation Contexts ............................. 33
  5.3 String Contexts ................................ 34
  5.4 Casting Contexts ................................ 34
  5.5 Numeric Contexts ................................ 34
  5.6 Poly Expressions ................................ 34
  5.7 Computing the Contexts .......................... 35
  5.8 Computing the Target Types ...................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Discussion</td>
<td>74</td>
</tr>
<tr>
<td>12.1 Problems Encountered</td>
<td>74</td>
</tr>
<tr>
<td>12.2 Parser Implementation in javac</td>
<td>75</td>
</tr>
<tr>
<td>12.3 Future Work</td>
<td>75</td>
</tr>
<tr>
<td>13 Conclusions</td>
<td>76</td>
</tr>
<tr>
<td>Appendices</td>
<td>77</td>
</tr>
<tr>
<td>A Test Coverage</td>
<td>77</td>
</tr>
</tbody>
</table>
1 Introduction

JastAddJ [6] is an extensible Java compiler built using the meta-compilation system JastAdd [7]. One of its main features is that it supports modularity, which means that new features can be implemented in separate modules without having to change anything in the original code. This makes JastAddJ very suitable to use as a base compiler for implementing language extensions or new languages on top of Java.

One example of such an extension done on top of JastAddJ is a null type checker, where the extended compiler can infer certain expressions to never be null [9]. Another example is an extension where several small, expensive remote calls can be sent in a single batch instead by using certain syntax in the extended compiler [10]. Language extensions can only be built on top of the Java versions actually supported by JastAddJ though, and it is thus interesting to keep JastAddJ up to date with the latest Java version.

JastAddJ currently supports Java 4, 5, 6 and 7 implemented as different modules built on top of each other [8]. Each module contains separate folders for files related to the abstract grammar, scanning, parsing, frontend and backend. Since the different versions are implemented as separate modules, any preferred version of Java can be built at any time, and potential bug fixes or changes to an earlier Java version will affect the later versions.

The goal of this thesis work was to extend JastAddJ with support for the, as of writing, latest Java version called Java 8 in a similar fashion to the previous versions. Java 8 introduces anonymous methods to Java, and also provides several changes to facilitate the use of these. Building and evaluating an extensive testing suite was also part of the goal in order to test the correctness of the compiler. The specific questions regarding the implementation that this thesis aims to answer are the following:

- Is the implementation complete? If not, what features remain to be implemented?
- To what extent could the new features be added modularly?
- Are the implemented features correct? How extensively does the test suite cover Java 8, and how many of these tests do JastAddJ pass?
- How well does the implementation perform regarding code size, memory usage and compilation time?

The most important results from this thesis work was thus the the practical implementation of a Java 8 module, and the solutions to the different problems that had to be solved in the implementation which are described in this thesis. The implementation will become available for download at http://www.jastadd.org. Also, the extensive testing suite which is used to verify that the proposed solutions are realistic to implement and works as intended. The thesis itself is split into the following sections:

- Section 2 presents a basic introduction to different important concepts relevant to the thesis, and introduces the new features of Java 8.
- Section 3 describes how the parsing and scanning of Java 8 was implemented.
• Sections 4 - 9 describes the semantic analysis of the different Java 8 features, both the theory and how the implementation was done.
• Section 10 explains the code generation of the new Java 8 features.
• Section 11 presents an evaluation of the implementation, and thus the results to the questions asked in this section.
• Section 12 discusses some different aspects of the work as a whole.
• Finally, section 13 concludes the thesis by summarizing the main results.

2 Background

This section contains basic information that is the foundation for a lot of discussion later in the thesis. First, the terminology used in the graphs throughout the thesis is explained. Then a short introduction to general compiler concepts which are referred to throughout the thesis. Lastly a basic introduction to the new Java 8 language constructs.

2.1 Graph Notation

Throughout this thesis, graphs are used to explain different concepts in the compiler. In this section, the different symbols used in the graphs are explained, and can be used as a reference when reading the graphs later in the thesis.

Figure 1: A table of the different symbols used in graphs.

Figure 1 shows the different graph symbols, each assigned a number. The definition of the symbols are as follows:

1. A node is represented by a circle. It is specifically nodes in the abstract syntax tree built by the parser that is referred to, see section 2.2.

2. A transparent arrow represents parent-child relationship between two nodes. If a transparent arrow starts in some node \( a \) and ends in some node \( b \), then \( b \) is a child of \( a \).
3. A filled arrow represents control flow. This shows the order in which events occur between nodes, for example if some node $a$ tries to access a method or attribute in some node $b$, this is represented with a filled arrow going from $a$ to $b$. If some value then is returned to $a$, this could be represented by yet another filled arrow from $b$ to $a$.

4. A dashed arrow represents that one or several nodes are skipped. If for example some node $a$ is a child of some node $b$, but not a direct child, this could be represented by a dashed, transparent arrow. Data being passed from $b$ to $a$ through several other not shown nodes is an example where a dashed, filled arrow could be used.

5. Numbered arrows represent an order of control flow. Only filled arrows are numbered, and represents in what order attributes are accessed between nodes.

2.2 Compiler Basics

This section explains some important, basic concepts in compiler construction which are used throughout the thesis. In general, a compiler is divided into the front end and the back end. Exactly what each does can vary a bit depending on the target platform and language, but in this section the focus will be on a Java compiler like JastAddJ.

The main tasks of the compiler front end is to parse the source program and perform syntactic and semantic analysis, while the back end translates a semantically correct program to the target code. To parse the source code a scanner and parser are used in combination, and these can be either hand written or generated automatically using scanner and parser generators. In this thesis, only generated scanners and parsers are discussed since that is what is used in the JastAddJ compiler. A scanner locates tokens, which are things like identifiers, keywords and operators, which it then forwards to the parser. The scanner typically does not know much about the overall grammatical context, but simply finds one token at a time. The parser on the other hand expects different types of tokens depending on the context, and the basic functionality is to check that a list of tokens make up a valid syntax according to the grammar.

Usually, a so called abstract syntax tree is also constructed during parsing by using custom code that is executed after a parser production has been matched. The abstract syntax tree is basically a big tree which represents the parsed program by using different nodes to represent different syntactical constructs. For example, see the code below:

```
5 + 7;
```

This will typically result in a small subtree with three nodes, the parent representing an addition, with children representing the constants 5 and 7. This can also be seen in figure 2.

The semantic and type analysis is then typically performed on this tree.

2.2.1 Scanning

In JastAddJ, the scanner generator JFlex is used to generate a scanner from a scanner specification. The generated scanner uses states and regular expres-
sions to match tokens, which it sends to the parser. These states and regular expressions are all described in the specification.

### 2.2.2 Parsing

Before discussing the specific parser generator that is used by JastAddJ, two common parsing techniques will be discussed, which are LL(k) and LR(k) parsing. LL parsing, which is top-down parsing, means that the parser decides on what construct to parse after looking only at the \(k\) first tokens together with the current context. The \(k\) is referred to as the lookahead, and how large it must be depends on how many tokens in the beginning of a production that can be shared with another production in the current context [11, p. 217-233].

To parse a grammar describing a complex programming language syntax, LL parsing might be too weak. In such a case an LR parser can often be used. LR is a type of bottom-up parser, which means it does not decide on what construct it is parsing until it has seen all the tokens in it. To achieve this a so called token stack is used, and as long as the parser can not decide on what construct it is currently parsing new tokens are put on the stack using an action called shift. When enough tokens have gathered on the stack that a new construct can be recognized, those tokens are popped and merged together, creating the recognized construct which is then put on the top of the stack. This action is called reducing, and when different reductions can be done depends on the current state in combination with the lookahead. As with LL parsers, the \(k\) represents the amount of lookahead. If the grammar is ambiguously written, the parser sometimes will not know whether or not to shift or reduce, since both actions are possible. This is called a shift/reduce conflict [11, p. 233-259].

A version of LR parsers are the so called LALR parsers, which is an LR parser with some states merged together in order to decrease the memory requirements of the parse table. In practice, LALR(1) is one of the most used parsers since it is strong enough to parse a wide range of grammars while not using an overly enormous parsing table. In JastAddJ, the Beaver parser generator has been used which is an LALR(1) parser generator. In the parsing specification, different types of productions representing statements, expressions and other types of language constructs are described, which Beaver then automatically generates a parser from [11, p. 266-270].
2.2.3 Semantic Analysis

When it has been made clear that the source code represents a valid syntax, it must also be checked that the semantic rules are followed. For example, the code below is valid Java syntax, but should still generate an error because it is semantically illegal to assign something of type `Object` to a variable of type `Integer`.

```java
// Semantic error
Integer i = new Object();
```

These rules are usually checked by performing analysis on the abstract syntax tree.

2.2.4 Code Generation

When the program has been checked for syntactic and semantic errors, the abstract syntax tree must be translated to the target code. Sometimes the AST is first translated to some intermediate code, which is in turn optimized and then transformed to the target code. What the target code is will of course vary depending on what platform the compiler is built for. For Java, the so called Java bytecode is used and can be executed by the Java virtual machine. In JastAddJ, the backend is responsible for translating the AST to Java bytecode.

2.3 JastAdd

JastAdd is a meta-compilation system that facilitates the construction of compilers, and is the tool used to build JastAddJ. The abstract grammar structure in a compiler can be specified in special grammar-files read by JastAdd, and JastAdd will then automatically generate the Java files representing the node types of the abstract syntax tree. JastAdd supports so called attribute grammars, which will be explained in more detail below. JastAdd also supports a declarative programming style, which means the attributes can be defined without specific order and in any number of files. Because of this, JastAdd provides modularity.

The JastAdd attributes are based on attribute grammars, which are a formal way to describe attributes in an AST computing certain values [12]. The attributes in JastAdd are an extended version of these attributes which can also result in a reference to other nodes in the AST, called reference attributes [13]. Attributes are implemented using Java code, and can take parameters and return values just as normal Java methods. They are not allowed to have side effects though, an attribute must compute the same value each time it is accessed, and in practice many of these attributes are cached so that they only need to be computed once.

The different types of attributes used in the Java 8 implementation and mentioned in this thesis are the following:

- Synthetic attributes. A synthetic attribute located in some node $N$ is defined in the same node and can only refer to other attributes located in $N$, or attributes located in children of $N$. In other words, when using synthetic attributes, information flows upward in the AST.
• Inherited attributes. An inherited attribute located in some node $N$ is actually defined in an ancestor-node of $N$, and can refer to attributes defined in the parent. When using inherited attributes, information thus flows downward in the AST. Inherited attributes also supports something called broadcasting, where an inherited attribute automatically flows down to all children even without explicitly defining the attribute in every node.

• Non-terminal attributes. NTA attributes are attributes that become a part of the AST by being evaluated to a sub-tree to the node where the attribute is located.

• Collection attributes. Collection attributes can be used to gather some data from many other nodes. The actual collection, which could be a map or a list, is defined in some node. Then several other nodes can be defined to contribute to this collection by adding some kind of value to it. Using collection attributes, mutating operations can be performed several times safely on a single attribute.

JastAdd also supports so called rewrites, where some node in the AST can be changed to another type of node when a certain criteria is met. These rewrites are performed as soon as the node is accessed the first time. There is also support for redefining the behaviour of pre-existing attributes, which is called 'refining' the attributes.

2.4 The Java 8 Language Constructs

This section contains an introduction to the new language constructs in Java 8. Section 3 describes the grammar in detail, and the following sections will each explain the specific semantic rules gradually.

The main features in Java 8 are:

• Functional interfaces

• Lambda expressions

• Method and constructor references

• Static and default interface methods

In order to facilitate the use of the new constructs, updates to the specification for method type inference and method overload resolution is also included in Java 8, together with some slight changes to the bytecode specification.

2.4.1 Functional Interfaces

Some interfaces are in Java 8 now regarded as so called functional interfaces, which is basically an interface that contains only a single abstract method, and can thus be described with only that method. This does not necessarily mean that the interface actually only contains one method though, but in case there are several abstract methods they have to be compatible, which is explained in more detail in section 4.1. There is also a new annotation interface introduced in Java 8 called FunctionalInterface, which acts as a marker guaranteeing that
a certain interface is functional. If a non-functional interface is marked using this annotation, the compiler must produce a compile-time error.

An example of functional interfaces can be found below:

```java
@FunctionalInterface
interface A { void m(); }
interface B { void m(); }
interface C extends A, B { }
```

All three interfaces above are functional since C only contains a single method in practice.

### 2.4.2 Lambda Expressions

Lambda expressions are basically a way to quickly instantiate a functional interface by only providing the single abstract method in the interface without using overly verbose syntax. For example, consider the Java 7 code below:

```java
Thread t = new Thread(new Runnable() {
    public void run() {
        //Some code
    }
});
t.start();
```

This is a fairly common way to create a new thread in Java, and since the alternative is to create explicit classes implementing Runnable for each different thread, an anonymous inner class is used instead. This code is overly bulky though, with a lot of meaningless wrapping. Since Runnable is a functional interface, in Java 8 the same thing can be done with the code below:

```java
Thread t = new Thread(() -> {
    //Some code
});
t.start();
```

Thus, only the method (which is implicitly 'run' in this case) must be specified. The lambda expression is here converted to an instance of Runnable with the method run as defined by the lambda. A lambda expression is also a closure, and can thus reference and capture final variables declared outside the lambda body, as in the example below:

```java
int a = 5;
() -> a + 6  //Can reference and capture 'a'
```

More detailed, a lambda expression defines a set of parameters and a body separated by the new arrow operator, which together construct the method represented by the lambda. The parameters can be empty as in the examples above, or they can be declared parameters like in methods. They can also be inferred parameters which only specify a name, and in this case the types are inferred by the compiler depending on what functional interface that is targeted by the lambda. The body is either a block or a single expression. Some valid examples of lambdas can be found below:
() -> { }  
(a, b) -> a + b  
(int a, int c, boolean b) -> b ? new Integer(a) : new Integer(c)  
a -> { return a + 1; }  
a -> a + 1;  
() -> System.out.println("A lambda");

2.4.3 Method and Constructor References

Sometimes, the behaviour wanted from a lambda is already defined in an existing method. In Java 8 such a method or constructor can be 'referenced' using the new double colon operator. This is basically the same thing as a lambda expression, except that instead of specifying the implementation of a new, anonymous method, an already existing method or constructor is instead referenced. This means that just as for lambdas, method and constructor references are only compatible with functional interfaces, and the single method in the interface thus is defined by the referenced method or constructor.

The general syntax for both types of references look like the following:

`location::method;`

The left hand side, called 'location' in the example, specifies where the method or constructor is located. For method references, this can be both a type or a primary expression, while for constructor references it has to be a class or array type. The right hand side is simply the name of the method being referenced for method references, or `new` for constructor references. Both the left hand side and the right hand side can include a list of type arguments, depending on if the location type or referenced method/constructor is generic. Note though that no method arguments are specified as these are inferred by the compiler.

Some examples of valid method and constructor references can be found below:

```
// Method references
ArrayList<Integer>::get;  
a[0]::<String>method;  
this::<String>method;  
b.super::<String>method;

// Constructor references
ArrayList<Integer>::new;  
ArrayList::new;  
int[]::new;
```

2.4.4 Interface Methods

In order for the new programming constructs in Java 8 to be able to be used to their fullest, updates to the Java standard library is required. Changes in interfaces risk breaking old code though, and in order to avoid this, Java 8 expands the types of methods that can be declared in an interface with static methods and so called default methods. They are declared using the `static` and `default` modifiers respectively. These both have in common that they must
be declared together with a method body, and not a semicolon like for abstract methods. Since all interface methods were abstract before, the fact that you can define complete methods together with a body is in itself a new feature in Java 8.

Static interface methods work like the static class methods, they are invoked by using the interface name as qualifier instead of an instance of the interface, see the example below:

```java
interface A {
    static int m(int i) {
        return i + 5;
    }
}

public void method() {
    // Call to static interface method
    int i = A.m(5);
}
```

Default methods are a bit different. Their name comes from the fact that they provide a ‘default implementation’ of a specific method, and are declared with a full body like static methods. Default methods are inherited by any class that implements the interface, and can thus be accessed as regular instance methods. See the example below:

```java
interface A {
    default int m(int i) {
        return i + 5;
    }
}

public class Inner implements A {
    public void method() {
        // Call to default method
        int i = m(5);
    }
}
```

3 Scanning and Parsing Java 8

Java 8 introduces several new forms of syntax that have to be parsed, some of which lead to potential conflicts with older language constructs. The biggest parsing challenges are caused by less predictability than before of where generics can appear and reuse of the already existing default keyword in interface methods. The different specific language constructs and their main parsing challenges will be discussed in this section.

3.1 Lambda Expressions

Lambda expressions can take several different forms, which leads to different parsing problems. Some typical lambdas can be seen below:
// Declared parameters with block body
(int a, int b) -> {
    if(a > 4)
        return 3;
    else
        return b;
}

// Inferred parameters with expression body
(a, b) -> System.out.println(a + b);

// Single inferred parameter
a -> a + 3;

// Empty parameters
() -> 3;

Every lambda expression starts with some (potentially empty) lambda parameters, followed by a right arrow, and finally ends with a lambda body that is either a block or an expression. The right arrow operator is a new operator, and thus needs to be added to the scanner. Since '-' and '>' can not be found in succession in any other context, this operator in itself caused no parsing conflicts.

The right arrow operator is also completely unique for lambda expressions, thus when the parser finds such an arrow it knows for sure that it is currently parsing a lambda expression. The lambda parameters can either be empty, inferred or declared. Declared parameters share the same structure as normal method parameters, while an inferred parameter is simply an identifier. Especially the declared parameters can be problematic to parse since generic types are allowed, which can conflict both with generic casts and less than expressions. For example, see the below three lines of Java 8 code:

(SomeType<String, Integer> s) -> { }; //Lambda expression
(SomeType<String, Integer>) s; //Generic cast
(SomeVariable < OtherVariable) //Less than expression

As can be seen in this example, quite a bit of code might have to be parsed before the parser can be sure of which construct that should be reduced to. One thing to note though, is that both for lambda expressions and casts, there is always a parenthesis preceding the generics, which means they can only conflict with less than expressions enclosed in parentheses. The lambda conflict is thus the same problem that had to be solved when generic type casts were introduced in Java 5, a problem which was already solved in JastAddJ.

Empty parameters and inferred parameters do not cause any particular parsing problems. When it comes to parsing the right hand side of the lambda, the block lambda body is not problematic since it has braces to mark the end and beginning, combined with the fact that a right arrow has already been found which limits the context to lambda expressions. There is an ambiguity in the grammar for expression lambda bodies though that can give conflicts in a parser, for example see the lambda expression below:

(x, y) -> x + y + 5; //Lambda expression with expression body
The *Java language specification version 8* (JLS8) makes it clear that the whole expression (the entire \(x + y + 5\)) is part of the lambda body, but it also states that lambda expressions are to be represented as a primary, which is the construct with highest precedence in Java [1, Ch. 15.27.2][1, Ch. 15.8]. This means a lambda expression could potentially be an operand to any operator. This, combined with the fact that a general expression (the construct with lowest precedence) is expected after a lambda means the above line can be legally parsed in several ways according to the structure of the grammar. These ways have been outlined using parentheses below:

\[
((x, y) \rightarrow x + y + 5); // A lambda expression \\
((x, y) \rightarrow x + y) + 5; // A lambda expression + 5 \\
((x, y) \rightarrow x) + y + 5; // A lambda expression + y + 5
\]

The correct way to parse this is the top example (where the parser reduces after reaching the semicolon), but the parser also has two other chances of reducing too early (at each plus). JLS8 states that to the right of the arrow there is one whole expression, so in general, the right hand side of a lambda arrow should always be reduced as much as possible before the whole lambda expression is reduced. These ambiguities in the grammar structure can lead to shift/reduce conflicts in a LALR(1) parser, but exactly how many of these there are will vary depending on how the grammar specification is written. Examining these conflicts, we note the following:

**Statement 1.** When parsing an expression lambda body with an LALR(1) parser, the correct parse action is always to shift whenever possible.

**Reasoning.** First, the following assumptions are made:

- All language constructs excluding the lambda expressions are already described unambiguously and correctly by the grammar, and can parse correctly.
- The lambda expression body is the only part of the lambda expression that is ambiguous. For example, if only lambdas with a block body are allowed temporarily, then there would be no conflicts and the correct parse tree would be built.

To reason why this will always work, we consider the potential case where the parser reduces when it could have shifted. We first notice that for every expression being parsed, there is some other, larger construct that is expecting one complete expression of some type. This construct made the parser enter a context where it expects an expression next. If several complete expressions are expected in the larger construct, there has to be some kind of separating character between them to avoid ambiguities, which means that in such a case we can simply look at one of those complete expressions expected. Since one complete expression is expected to be reduced to in the end, this leads to the conclusion that when an expression construct is being parsed there is always a point where the parser reaches a character that can not be shifted until this complete expression has been reduced. Such a character has to exist, and in the most extreme scenario it would at least be the end-of-file token. In Java, expressions are not allowed to end a program though so there will always be some other character. Thus we can assume that by continuing to shift new tokens
while parsing a lambda expression body, we will eventually reach a token that require the parser to reduce the complete lambda expression before it can shift again.

Now, assuming the parser reaches a reduce/shift conflict, we can be certain this is a conflict between reducing some number of tokens on the top of the token stack to a lambda or to continue shifting. Since we have assumed conflicts are introduced only together with the lambda expression body, and not for example with the lambda parameters, we can also be certain that the only conflicting option introduced by the lambda expression is the added option to reduce in a situation where only shift would have been possible with lambdas excluded. Assuming we have a lambda mixed into an expression, the first part of the lambda found by the parser will be the lambda parameters, which parse unambiguously together with other expressions, and thus there can be no conflict added because of the option to shift these lambda parameters. This leads to the conclusion that the only possible conflict introduced in this case is when a lambda expression body is being parsed, and the result so far can be reduced early to a complete lambda expression.

If the parser reaches a point where reduce is the only option, there are two different cases where this is possible. The first one is when some of the tokens on the stack need to reduce in order to be able to parse the continuation of the lambda expression body. For example, say we have a name, followed by an asterisk and another name on the top of the stack, and the next lookahead is a plus sign. The 3 top tokens on the stack need to be reduced to a multiplication expression first to be able to shift the plus. This is the same as continuing to parse more of the complete expression to the right of the arrow, and will add new tokens on the stack. The second case is when the final separating character has been reached, and according to the reasoning earlier the complete lambda expression must now reduce before a shift is possible. Thus, all tokens that could possibly be a part of the lambda expression body have already been placed on the stack, and according to the first case been reduced as required along the way. Thus, before the result is reduced to a lambda, the complete expression to the right of the arrow will be on the stack, which means that the whole expression to the right of the arrow will become part of the expression lambda body.

Even if there are new lambdas in the right hand side of some other lambda, the same reasoning holds. No lambda will be reduced until the final separating character is reached, which means everything will be on the token stack once the lambdas start reducing.

3.1.1 Handling the Ambiguities With a LALR(1) Parser Generator

Assuming the grammar ambiguity mentioned in the above section leads to shift/reduce conflicts using some LALR(1) parser generator, this needs to be dealt with in some way. Potential solutions with their respective advantages and disadvantages are discussed below.

3.1.1.1 Parser Default Action

One way of solving this problem is to simply do nothing. Most, if not all LALR(1) parser generators use shift as the automatic action in the case a
shift/reduce conflict should appear. This includes Beaver, the parser generator used for JustAddJ. As mentioned in statement 1, shifting if possible will lead to the correct end result and thus even if nothing is done to handle these conflicts the parser will produce the correct tree. The main advantages to this approach is that it requires no work and there is no need to reorganize old grammar constructs. This will also produce the correct result since we know that the parser will always shift when it should. The main disadvantage is that it can produce a large amount of warnings. This risks silencing future warnings (because the compiler designer might miss them), overall reduces the readability of the parser outputs and is generally annoying.

3.1.1.2 Suppressing Warnings With Directives
If the parser generator used supports it, using a directive to simply suppress the warnings is also an option. For example, Bison uses the \%expect directive to specify how many conflicts that are expected at a certain production. If the number match perfectly with the actually produced number of conflicts, Bison does not produce any warnings. The main advantages to this approach is that it is easy, requires a very small amount of work and still removes all the warnings. The main disadvantage is that, depending on what directive is used, warnings that should not be silenced could be potentially be silenced if the compiler designer is not careful enough. For example with the Bison directive, only the number of warnings decide whether or not to silence them. Care has to be taken that one warning is not simply replaced with another, which will be silently resolved.

3.1.1.3 Explicit Precedence
Another solution is to remove the ambiguities by specifically telling the parser what to do at each conflict by providing explicit precedence instructions for each relevant production. Most parser generators have directives that can be used to set an explicit precedence for each grammar rule, and any potential shift/reduce conflicts can be manually solved in this way. Since we always want to shift in this case, directives must be inserted in the parser that make sure shifting is chosen over reducing. For this to work, the relative precedence amongst all terminals representing operators must be set correctly.

The main advantage to using explicit precedences is that all warnings from the parser generator can be silenced, making the output readable. It is also fairly easy to implement, and requires no large reorganizations of the old grammar productions. It has some disadvantages though that need to be considered. First, the precedence rules have to be written correctly, or the parser will not produce the correct tree. Depending on how precedence is currently enforced in the parser, errors in the operator precedences might only be noticeable when a lambda expression is used, which will require specific testing. This is especially bad considering that introducing no extra precedences at all will guarantee a correct parse tree because of the parser default action.

Another drawback is that precedence has to be added to some grammar rules that have nothing to do with expressions, which does not really make much sense except for manually solving this very specific case. This might have to be considered in future expansions of the compiler if these grammar rules are somehow modified or reused. For example, say some ambiguous grammar is
implemented by mistake reusing these constructs which now has explicit precedence. The parser might now silently resolve some conflict in an incorrect way. Also, if it is somehow required in a future expansion to expand the expression construct, this might produce new warnings that are related to the expression lambda body if precedence is not immediately set correctly. These warnings could be confusing to the compiler designer.

Explicit precedences is the solution chosen and implemented in the JastAddJ Java 8 module, since there is no directive in the Beaver parser generator to silence warnings.

3.1.1.4 Rewriting the Grammars
Another strategy could be to rewrite these grammar productions in such a way that the structure of the productions themselves remove the ambiguities. The main problem is caused by the fact that a lambda expression can appear as a left operand in all expression types, so if that could be prevented by changing the left hand productions to one which can not contain a lambda expression, this could be one approach to solve the ambiguities. This would require every expression containing a lambda to be reduced to a different type of expression than the ones which do not contain any lambdas, in combination with preventing the expressions that can contain lambdas from appearing as a left hand side operand in any expression.

The main disadvantage of this approach is that it requires the entire expression hierarchy to be duplicated, which makes the resulting code harder to understand and maintain. This is probably not a good solution considering that there are several other, easier options.

3.2 Method and Constructor References
Apart from lambda expressions, the other main form of completely new syntax are the method and constructor references. Some typical examples can be found below:

```java
// Method reference with expression
a[0]::method;

// Method reference with super and explicit type arguments
a.b.super::<Integer, String>method;

// Method reference with type
ArrayList<Integer>::get;

// Constructor reference with class
HashMap<Integer, String>::new;

// Constructor reference to array
int[]::new;
```

As can be seen above, a new separator represented by a double colon has been specifically introduced for this syntax, which needs to be added to the scanner. As for the right arrow, this is as simple as adding it to the scanner specification.
The double colon separator is unique for method and constructor references, which means that once the parser reaches this separator it can be sure it is currently parsing a method or constructor reference.

Unlike lambdas, the right hand side of these expressions is simple to parse and contain no conflicts. The left hand side does cause some very tricky parsing problems though, mostly related to generics. As with lambda expressions, both method and constructor references are proposed to be represented as a primary construct, which means that every single expression in all contexts can be a method reference. Unlike lambdas though, when using a generic type with method references the generics do not have to be preceded by a parenthesis, which means that suddenly every single expression can now start with either a generic type or a less than expression. This is a new parsing challenge that did not had to be taken into account before Java 8.

An example of the conflict can be seen below:

```java
someVariable < otherVariable;  //Less than expression
someType<otherType>::method  //Method reference
someType<otherType>::new  //Constructor reference
```

There is another very tricky parsing problem introduced by method references which is also related to generics, and it might thus be a good idea to solve both these problems at once. In order to illustrate the problem, an example can be found below:

```java
//A method call with 3 parameters
methodCall(variableA < variableB, variableC, variableD);
```

```java
//A method call with 1 parameter (a method reference)
methodCall(typeA<typeB, typeC, typeD>::method)
```

```java
//A method call with 1 parameter (a constructor reference)
methodCall(typeA<typeB, typeC, typeD>::new)
```

As can be seen in the above example, an entire type argument list of arbitrary length might have to be traversed before the parser can decide what it is parsing.

Both conflicts discussed so far in this section are caused by the fact that when a '<' character appears, the parser does not know if what is coming next is a less than expression or a method or constructor reference. If the '<' token could be represented as a different symbol when it is used in a less than expression compared to when it is used in method/constructor references, all conflicts would be solved. Unfortunately, the scanner will not know the difference normally since it looks at one token at a time, but with some tweaking it can be done.

For the implementation in JastAddJ, a scanner wrapper with a token buffer was implemented, which can be seen in figure 3. This scanner wrapper reads tokens from the normal generated scanner and passes them to the parser. The main idea of the scanner wrapper is to start a lookahead of tokens from the real scanner when needed, and store the tokens which are previewed in a local buffer instead, so that the order in which they should be sent to the parser is preserved. When the scanner wrapper finds a '<' token, a token lookahead is initiated which previews some number of tokens and inserts them in the buffer, and depending on the result of the lookahead analysis either the normal '<' token or one specific for method references is sent to the parser.
3.3 Intersection Type Casts

In Java 8 a new type of casts were added, using an intersection type. For example:

(A & B & C)var;

As can be seen above, the only difference compared to normal type casts is that after the first type, now either an ending right parenthesis or a `&` token can follow. For the specific case when the first type is a generic type, this does not introduce any new conflicts since this is no different from normal generic type casts. The non-generic version introduces a tricky conflict though, as can be seen in the example below:

(Type1 & Type2 & Type3) var //Intersection type cast
(var1 & var2 & var3) | var4 //And-expression

The parser here has to process an arbitrary number of tokens before it can decide whether it is parsing an and-expression or an intersection type cast. To solve this, the token after the final enclosing right parenthesis need to be processed before the tokens inside the parentheses are reduced. In most cases, the tokens that are legal after a cast body are different from the tokens that are legal after an and-expression, but there are some cases where this is not true which require special attention. An example can be found below:

Figure 3: Scanner wrapper used to preview tokens
// And-expression in an if-statement
if(a & b & c)
    methodCall();

// Intersection type cast of method call
(a & b & c) methodCall();

In this case, the tokens following the parenthesized expression are exactly the same but both legal. There is a similar case for example using a while statement. To prevent this, not only the token following the parentheses needs to be considered, but also the token before so that the surrounding context can be used when making the parsing decision. This can be achieved by adding additional functionality to the scanner wrapper mentioned in section 3.2.

The updated version of the scanner wrapper still performs the same token lookahead for method references as described in section 3.2, but in addition also handles intersection casts. The idea used to solve this problem is to use a marker token to help the parser make the right decisions [14]. This marker token should precede every intersection type cast with a non-generic first element and only this, which will make the grammar completely unambiguous. See the example below, where the marker token is referred to as INTERCAST:

// Intersection type cast of method call
(a & b & c) methodCall();

// No longer any conflict due to marker token
INTERCAST (a & b & c) methodCall();

The addition to the scanner wrapper keeps a reference to the previously scanned token all the time, and then perform a token lookahead when a left parenthesis is found. Using these two pieces of information, the scanner wrapper will decide whether or not to send an INTERCAST marker token to the parser before sending the current left parenthesis.

3.4 Default Methods

The types of methods that are legal to define as an interface member has changed in Java 8. More specifically, complete method bodies are now legal to define as long as the modifier default or static is used in the method header. Static interface methods are also legal. Some examples can be found below:

```java
public interface MyInterface {
    default int m1() { return 3; }
    static int m2() { return 2; }
}
```

The main changes that this causes to the parser is that some new modifiers are now legal to parse for interface methods. Specifically the use of the default keyword introduces some conflicts that needs to be handled, even though the context in where it is used differs. To explain why, see the example below:

```java
public interface MyInterface {
    public strictfp static class MyClass { }
    public strictfp default void myMethod() { }
}
Since the legal modifiers differ for classes and interface methods, these are also parsed as different productions. This introduces a conflict since the normal class declaration uses normal modifiers, while the method declaration uses interface modifiers. Thus when the parser has processed the public modifier, it needs to decide whether or not to reduce it to a normal modifier or an interface modifier, which it cannot do yet. One way to get around this is to simply add the `default` keyword to the normal modifiers, but if this is done the contexts where a `default` keyword can be expected increases which creates a conflict with how the `default` keyword is used in switch statements. An example can be found below:

```java
switch(var) {
    case 'a': int i = 0;
    default class { }
    default: int j = 0;
}
```

When the first `default` keyword is reached in the code above, the parser must decide whether or not to reduce the whole line above to a specific case production or to keep shifting in order to find out that there is more to parse before a reduce is possible. Of course declaring a default class is not semantically legal, but that does not matter since the parser will not be able to correctly parse normal switch-statements due to the conflict.

For the JastAddJ Java 8 module implementation, in order to avoid these conflicts a special construct was used to parse interface methods. The basic idea is to split it into two different constructs, the first one is only expecting normal modifiers while the other one expects zero or more normal modifiers, followed by one `default` keyword, followed by zero or more interface modifiers. Doing it like this requires some list manipulations while building the abstract syntax tree, but it solves all conflicts. The first conflict is solved since until a `default` keyword is found, the parser can always reduce to a normal modifier. The second case is solved since the `default` keyword is never added to the normal modifiers. This does allow the parser to accept some modifiers that are not semantically legal for interface methods, but this can be handled in the semantic analysis instead.

### 3.5 Package Modifier

A new package modifier was added in Java 8, which can be used to explicitly declare a class to have package access. Adding parsing support for this introduces no new challenges since there is no conflict by simply adding the package keyword to the already existing modifier production.

### 3.6 Testing the Parser

Java 8 introduces many new tricky parsing challenges, and being able to test only the parser isolated from the semantic analysis could thus be useful. How to best test the parser is not a trivial problem though, and some different techniques that can be used are discussed in this section.
3.6.1 Syntax Error Check

A compiler will generate syntax errors whenever an incorrect parsing structure is found, and one way of testing the parser might thus be to simply check whether syntax errors are generated or not for some different test files. For some files, syntax errors could be expected while for others no errors could be expected. This approach both has the advantage that it is easy to implement, and that even if the internal grammar productions should change, the tests will not be affected as long as the specified language syntax does not change. Should that be the case though, tests would probably have to be rewritten no matter what testing method is being used.

A major drawback using this technique is that it does not actually check that the correct parse tree is being built, which is not good enough since we do not want test files which parse incorrectly in reality to actually pass.

3.6.2 Parse Tree Check

Another approach could be to implement a function that prints the parse tree, and then test that the resulting output when parsing some test files does not differ from a predefined, correct parse tree. This solves the problem of the actual parse tree structure not being tested, but has some other major drawbacks. The biggest disadvantage using this method is that it is very static, and even the most minor changes in internal grammar could require most of the tests having to be rewritten. The predefined parse trees also have to be checked manually that they are correct, which could take a lot of time and might be prone to errors. Using this approach is thus basically the same thing as simply manually checking every parse tree computed by every test file, and since this needs to be done every time the internal grammar representation changes, the tests themselves become redundant.

3.6.3 Structured Print Check

One way of making the tests less static while still testing the parse tree is to print the program code itself and check against that instead of the whole tree. By implementing a print function which adds parentheses around each sub-expression, the actual tree-form can be seen in the resulting output, but with much less internal information leaking out making the tests more robust for parser changes. Changing the parser constructions might require changes to the print function, but at the very least the tests themselves should not be affected. This strategy is the one used to test the Java 8 parsing functionality in JastAddJ.

4 Functional Interfaces

When the parse tree has been correctly constructed, the next step is to perform a semantic analysis. The semantic rules are explained in detail in the Java language specification, and it is then the job of the compiler to check that the program abides by these rules. In order to do this, those rules must first be well understood, and the following sections will thus each begin with an explanation of the semantics.
In this section, functional interfaces will be discussed. First the semantic rules will be presented, and then the details of the actual implementation.

4.1 Semantic Rules
A functional interface is an interface which in practice only contains a single, abstract method (it is allowed to contain any number of non-abstract methods). As mentioned in section 2.4.1, several abstract methods can exist in an interface and the interface can still be considered functional. In this section, the rules defining whether or not an interface is functional will be presented. First, all abstract methods in the interface (both local and inherited) are collected, and then among these the following procedure can be used to check if an interface is functional:

- First, all methods with signatures that equals a public method already defined in Object are removed from the check.
- If the number of remaining methods is zero, the interface is not functional.
- If the number of remaining methods is exactly one, the interface is functional.
- If the number of remaining methods is more than one, one of the methods must be able to override the rest.

To compute if a method \( m_1 \) can override another method \( m_2 \), two things must be checked. First, \( m_1 \) must be a so called subsignature of \( m_2 \), and \( m_1 \) must be return-type-substitutable for \( m_2 \). The exact meaning of these properties will be explained in sections 4.1.1 and 4.1.2 [1, Ch. 9.8]. Some examples of functional interfaces can be found below [4, Part A]:

```java
//Functional, only one method
interface A { public void method(); }

//Functional, only one method not in Object
interface B extends A { public boolean equals(Object o); }

interface C { <A> public A method(int a); }
interface D { <B> public B method(int a); }

//Functional, both signatures are equal with the same return type
interface E extends C, D { }
```

4.1.1 Subsignatures
A method signature is a combination of the method identifier, type parameters and formal parameters. For signature \( s_1 \) to be a subsignature of signature \( s_2 \), either \( s_1 \) is equal to \( s_2 \), or \( s_1 \) is equal to the erasure of \( s_2 \), where erasure means that all generic parameters are removed, and type variables are changed to their leftmost bound. To check that a signature equals another, there are three different checks that must be performed [1, Ch. 8.4.2]:

1. The same type parameters
2. The same name

3. The same formal parameters

The name is simply the identifier used when declaring the method. The formal parameters are the parameters which define what type of variables that can legally be passed as arguments to the method. An example of the type parameters of a method and their bound can be found below:

```java
public <A extends ArrayList<Integer>, B> void method() {
}
```

The method `method` above has two type parameters, `A` and `B`. The bound of `A` is `ArrayList<Integer>`, and the bound of `B` is implicitly `Object`.

Two methods must share the same name, otherwise they cannot have the same signature. To check if two methods have the same type parameters, the following two checks are performed:

- Are the number of type parameters the same?
- Do all type parameters in the first method have the same bounds as the type parameters in the second method?

Another important thing to notice is that when comparing the signature of two methods, their respective type parameters must be adapted to each other, which means that the position and not the name is what is relevant when comparing two type variables declared by the methods being compared. Also, the bounds must be checked in such a way that the order does not matter [1, Ch. 8.4.4]. Two examples can be found below:

```
// Example of two methods with same type parameters
public <A, B, C extends Serializable & List<A>> void method();
public <B, A, C extends List<B> & Serializable> void method();
```

```
// Example of two methods with different type parameters
public <A, B, C extends A> void method();
public <B, A, C extends A> void method();
```

Note that since the position is what is relevant in the examples above, `A` from the first method is considered the same as `B` in the second method, and so on.

Checking that the formal parameters are equal means that the number of parameters are the same, and that each parameter at a certain position in the parameter list have the same type in both methods. Also here the type parameters have to be adapted to each other as explained above, so that the position is taken into account instead of the name. Two examples can be found below:

```
// Example of two methods with same formal parameters
public <A, B> void method(A a, ArrayList<B> b);
public <B, A> void method(B b, ArrayList<A> a);
```

```
// Example of two methods with different formal parameters
public <A, B> void method(A a);
public <B, A> void method(A a);
```

Note that in the examples above, since the type parameters `A` and `B` are declared in reverse, they must also be used in reverse in the formal parameters.
4.1.2 Checking Return-Type-Substitutability

Return-type-substitutability approximately means that if method \( M_1 \) is return-type-substitutable for some method \( M_2 \), then any value returned from \( M_1 \) could also safely be returned from \( M_2 \). For example, if \( M_1 \) returns \( \text{ArrayList<Integer> \text{ and } List<Integer}> \), since an \( \text{ArrayList<Integer}> \) is a subtype of \( \text{List<Integer>} \), any value returned from \( M_1 \) could safely also be returned from \( M_2 \). The other way around does not hold though, since a return value from \( M_2 \) could be a \( \text{LinkedList} \) or some other list incompatible with the return type of \( M_1 \).

In general, two methods are return-type-substitutable if both have void types or the same primitive type. If both methods have reference return type, then in general \( M_1 \) is return-type-substitutable for \( M_2 \) if the return type of \( M_1 \) is a subtype of the return type of \( M_2 \) [1, Ch. 8.4.5].

As with formal parameters, type parameter adaption might have to be performed. Some examples can be found below:

```java
// Example of two methods which are return-type-substitutable
public int method();
public int method();

// Example of two methods which are return-type-substitutable
public <A, B extends A> A method();
public <B, A extends B> A method();

// Example of two methods which are return-type-substitutable
public <A, B extends ArrayList<A>> ArrayList<A> method();
public <B, A extends ArrayList<B>> A method();

// Example of two methods which are not return-type-substitutable
public <A extends Integer, B extends A> B method();
public <B extends Double, A extends B> A method();
```

4.1.3 Function Type

Every functional interface has a so-called function type, which describes the single abstract method in the interface. The function type contains a set of type parameters, return type, formal parameters and a list of thrown types. The type parameters, return type and formal parameters are the same as the abstract method found to override the rest of the methods in the functional interface.

The throws clause requires some more work to compute though. The basic idea is that the function type is only allowed to include types which could potentially be thrown by all abstract methods in the functional interface, otherwise the function type would not correctly describe a single method which can replace all abstract methods in the interface. Since some of the abstract methods might be generic and throw type variables, while some might not, this has to be taken into account when computing the thrown types [1, Ch. 9.8]:

Some examples of function types taken from the JSR335 can be found below [4]:

27
interface X { void m() throws IOException; }
interface Y { void m() throws EOFException; }
interface Z { void m() throws ClassNotFoundException; }
interface XY extends X, Y {}
interface XYZ extends X, Y, Z {}

// XY has function type ()->void throws EOFException
// XYZ has function type ()->void (throws nothing)

interface A {
    List<String> foo(List<String> arg) throws IOException, SQLTransientException;
}
interface B {
    List foo(List<String> arg) throws EOFException, SQLException, TimeoutException;
}
interface C {
    List foo(List arg) throws Exception;
}
interface D extends A, B {}
interface E extends A, B, C {}

// D has function type (List<String>)->List<String> throws EOFException, SQLTransientException
// E has function type (List)->List throws EOFException, SQLTransientException

4.2 Implementation of Semantics Checking

This section explains how the semantic rules outlined in section 4.1 have been implemented in JastAddJ.

An attribute computing whether a certain interface is functional or not was implemented. The first step is to collect the abstract methods which was trivial. Once the relevant methods have been collected, the number of methods can be checked, and if it is equal to 0 or 1 a result can be returned immediately. Otherwise, the signatures have to be compared to each other before a result can be returned. More specifically, a subsignature check and a return-type-substitutable check had to be implemented. These existed already in JastAddJ, but did not take type parameter adaption (see section 4.1.1) into consideration, thus these attributes had to be rewritten for the functional interface analysis. How this was done is explained in more detail below.

4.2.1 Subsignature Analysis

An attribute for checking whether a method’s signature is a subsignature to some other method’s signature or not needed to be implemented. This is trivial to implement in general, with one problem, which is how to implement the comparison of type variables. Sometimes when comparing two types from different methods for equality, two type variables should be regarded as equal even if
they have different names. Assuming two type variables are being compared for equality, we note that the information required about them to do the comparison correctly are the following:

1. The position in the type parameter list where it was declared.
2. Whether it was declared in a generic method or not.

Whenever two type variables are compared, first it should be checked if they are both declared in a method. If that is the case, the positions can simply be compared, and if they equal then the comparison should return true, otherwise false. If they are not declared in a generic method, the declarations can be compared and if they equal then true should be returned, otherwise false. The reason why it is enough to simply check if they were both declared in some method is because we are only comparing abstract interface methods. An interface is only legal to declare in a top scope or static context, and can thus never exist nested inside a method since a method can not declare static classes.

We also know that any type variable that shows up in a comparison was either declared by one of the methods being compared, or it was declared in an outside scope, for example an interface. Since an interface can never be nested in a method, a variable coming from a method cannot come from an outside scope, which means that such a type variable has to come from one of the methods being compared. Thus we can be certain that in such a case it is definitely enough to simply compare the positions.

It would therefore be practical to have attributes for the two properties mentioned above. This could be done relatively easy using inherited attributes, and a typical example of the data flow in the implementation can be seen in figure 4.

With these attributes implemented two types can be compared for equality with recursion. Trivial types like primitive types and reference types which are not parameterized can be checked for equality very easily with a single comparison. Non-trivial types like parameterized types or array types have children with the other types contained in them, so when comparing two non-trivial types this can be done recursively, checking one level at a time, and should the search complete all the way down without finding any non-equalities true should be returned. Whenever two type variables must be compared, the attributes mentioned above for position and method declaration can be used in combination. An example of two parameterized types being compared can be found in figure 5.

When two types can be compared to each other, comparing two formal parameter lists to each other is as simple as checking one parameter at a time, and if all checks pass the formal parameters are deemed as equal. For bound lists, the check is a bit more complicated since the order of the bounds does not matter, but only that the same number of the exact same types exist in both bound lists.

4.2.2 Return-Type-Substitutability Analysis

As stated in section 4.1, return-type-substitutability is easily checked for primitive and void return types, while reference types are a bit more difficult. Since
Figure 4: Data flow in the implementation of type variable position and method attributes.

Figure 5: Example of how types can be checked recursively while taking type variable substitution into account.

the types being compared now only requires one of them to be a subtype to the other, and not that the types are equal, this opens up for some tricky situations, see the examples below:

```java
interface X { <A, B extends A> A execute(int a); }
interface Y { <C, D extends C> D execute(int a); }

interface XY extends Y, X {}```

30
The interface $XY$ above is functional, because $D$ behaves like a subtype of $A$.

```java
interface M {
    <A, T, S extends List<A>> List<T> execute(ArrayList<S>[] a);
}
interface N {
    <A, S, T extends List<A>> T execute(ArrayList<T>[] a);
}
interface MN extends M, N {
}
```

The interface $MN$ above is not functional, because the methods are not return-type-substitutable.

If two types containing type variables are checked for subtype, the type variables will eventually be compared to each other assuming the check did not already fail before that could happen. What needs to be added then is a check that returns true if the two type variables have the same position and are both declared by the methods being compared. The bounds also need to be involved in the check. Using interface $XY$ from the example above, when it is checked if $D$ is a subtype of $A$ the following steps are performed:

1. It is checked if $A$ and $D$ are both declared by methods and at the same position. In this case, this is false, so the check continues.
2. $D$'s bound is instead checked now. Thus, the check continues to examine if $C$ is a subtype of $A$.
3. To do this, it is first checked if $C$ and $A$ are both declared by methods and at the same position. This is true, so the whole subtype check returns true.

### 4.2.3 Computing the Function Type

The function type contains four fields of information representing the single abstract method. These are type parameters, return type, formal parameters and legal thrown types. For a functional interface, there is always a method found which can override all other methods, and the return type, type parameters and formal parameters can be fetched directly from this method. Computing these are thus very easy, and the main problem is to compute the legal thrown types.

To find the legal thrown types, all the abstract methods’ thrown types has to be compared to all throw clauses in all the other abstract methods. Should there only exist one abstract method the thrown types can be fetched directly from that method. Otherwise, it is checked that all thrown types being added to the function type are equal to or a subtype of at least one thrown type in every clause. Depending on if the single abstract method is generic or not, this process is slightly different.

If the method is not generic, then the thrown types are not compared directly but their erasures are. Any types then found to belong to the function type are added in their erased version. This makes sense since a non-generic function type can not refer to type variables, and since type variables are erased to their bounds, all thrown types added to the function type will still be types that could possibly be thrown by every abstract method in the interface. See the example below:
interface A { <T extends IOException> void m() throws T; }
interface B { void m() throws Exception; }
interface C extends A, B { }

// C:s function type: () -> void throws IOException

In the example above, when computing the legal thrown types for C’s function type, it is checked if Exception is a subtype of the erasure of T, which is IOException. This is false, so Exception is not added to the function type. Then it is checked if IOException is a subtype of Exception. This is true, so IOException is added to the function type.

If the method is generic, then the thrown types are compared in their original form but while taking type parameter adaption into account. An example can be found below:

interface A {
    <T extends IOException> Number m() throws T;
}
interface B {
    <S extends IOException> Integer m() throws Exception;
}
interface C extends A, B {}  

// C:s function type: <S extends IOException> () -> Integer throws S

When computing the legal thrown types for C’s function type, it is checked if Exception is a subtype of T. It is not, so Exception is not added. Then it checks if T is a subtype of Exception, which it is, so the first type variable in the function type’s type parameter list is added to the thrown types. Note that the type variable thrown by C is its own type parameter and not T.

interface D { <T extends IOException> void m() throws T; }
interface E { <S extends IOException> void m() throws S; }
interface F extends D, E { }

// F has function type <T extends IOException> () -> void throws T

In the example above, both T and S are found to be valid thrown types, but since they are in practice the same type (the first type parameter in the function type), they are only added a single time in total to the resulting function type.

5 Contexts and Target Types

In Java 8, context for expressions has been more explicitly defined compared to Java 7. Five types of contexts are defined in the JLS8, and depending on where an expression is found, it might be located in one of these particular contexts. The five types of context are:

- Assignment contexts
- Invocation contexts
- String contexts
• Casting contexts
• Numeric contexts

Java 8 also defines something called a *target type*, which is the type that is expected in a certain context [1, Ch. 5].

### 5.1 Assignment Contexts

Expressions which are the right hand side of some assignment are in assignment context. Assignment context does not only apply to normal variable assignments or variable initializations, but also in return statements, array initializers and other constructs which in the end lead to an assignment.

The target type in assignment contexts is the type of the left hand side, or the type that the right hand side is going to be assigned to [1, Ch. 5.2]. See the code below for some examples of assignment context with target type: 

\[ x_1 \] below is in assignment context with target type `int`:

```java
int i = x1;
```

\[ x_2 \] and \[ x_3 \] below are in assignment context, and target type for both is `String`:

```java
String[] s = {x2, x3};
```

\[ x_4 \] below is in assignment context with target type `List`:

```java
public List method() {
    return x4;
}
```

### 5.2 Invocation Contexts

Expressions which are arguments to method calls, constructor invocations or class instance creation expressions are located in invocation context. The target type is the type of the corresponding parameter in the method declaration or constructor that is invoked. Special care needs to be taken for variable arity methods, where the target type can become the component type of the last parameter [1, Ch. 5.3]. See the code below for examples of invocation contexts:

\[ x_1 \] below is in invocation context with target type `int`:

```java
class Test {
    public Test(int i) { }
}
```

```java
new Test(x1);
```

\[ x_2 \] and \[ x_3 \] below are in invocation context, and target type of both is `List`:

```java
public int m(int i, List... l) { return 4; }
```

```java
int i = m(4, x2, x3);
```
5.3 String Contexts
String contexts only apply to expressions which are the operand to a binary '+'-operator, and only when the other operand is of type String. The target type is always String [1, Ch. 5.4].

5.4 Casting Contexts
Expressions which are located inside a cast are in casting context. The target type is the type specified by the cast [1, Ch. 5.5].

5.5 Numeric Contexts
Expressions which are operands to an arithmetic operator are in numeric context. The target type is the type which the operand should be promoted to [1, Ch. 5.6].

5.6 Poly Expressions
Some expressions use their target type to compute their own type, and they are thus dependent on knowing their target type before they can compute their own. In order to differentiate between expressions where the target type influences the resulting type and which expressions that have the same type no matter what, a new type of expressions is introduced called poly expressions. A poly expression is thus an expression whose type may be influenced by the target type. Non-poly expressions are referred to as standalone expressions, and their types are not dependent on the target type [1, Ch. 5.6]. The following expressions can be poly expressions:

- Parenthesized expressions are poly expressions if the expression inside the parentheses is a poly expression. Otherwise they are standalone expressions [1, Ch. 15.8.3].
- Class instance creation expressions are poly expressions if they use the diamond syntax from Java 7 and appear in assignment of invocation context. Otherwise they are standalone expressions [1, Ch. 15.9]. An example of the diamond syntax can be found below:

```java
// The type 'Integer' is inferred here
ArrayList<Integer> = new ArrayList<>();
```

- Method invocation expressions are poly expressions if the following conditions all hold, as defined in the JLS8:
  - The invocation appears in an assignment context or an invocation context
  - The invocation does not provide explicit type arguments
  - The method to be invoked is a generic method
  - The return type of the method to be invoked mentions at least one of the method's type parameters

Otherwise they are standalone expressions [1, Ch. 15.12].

Conditional expressions have been categorized into three different types in Java 8. Boolean conditional expressions, numerical conditional expressions and
reference conditional expressions. To put it simply, a boolean conditional expression is a conditional expression which always result in a type of boolean or Boolean, while a numerical conditional expression always result in a numeric value. The reference conditional expressions are all other conditional expressions.

A conditional expression is a poly expression if it is a reference conditional expression and it appears in assignment context or invocation context, otherwise it is a standalone expression [1, Ch. 15.25].

Lambda expressions and method and constructor reference expressions are always poly expressions [1, Ch. 15.13][1, Ch. 15.27].

5.7 Computing the Contexts

Five inherited attributes were defined to represent the different contexts. All five return a boolean value representing whether or not an expression is located in a certain context or not. The attributes needs to be inherited since only the parent knows about the context of its children. For example, a cast expression should set its expression child’s cast context attribute to true, and an assignment expression should set the assignment context attribute for the right hand side expression to true. There are also some special cases which are less obvious, for example a conditional expression should propagate its own context attributes to both expressions to the right of the question mark if it is a poly expression [1, Ch. 15.25.3].

Examples of the data flow when setting three of the attributes can be seen in figures 6 and 7.

![Figure 6: Example of the data flow when three of the context attributes are computed.](image)

5.8 Computing the Target Types

An expression defined in a certain context will always have a target type which needs to be computed. Doing this is in general similar to computing the context attributes, and an inherited attribute can be used to delegate the correct type to the expression targeting it. For example, an assignment expression can set
Figure 7: Example of the data flow when three of the context attributes are computed.

the target type of the right hand side expression to equal the type of the left hand side expression, and a cast expression can set the target type of the right hand side expression to the type inside the parentheses.

For normal methods, the method will delegate the correct target type to all return statements in the method body, see the example below:

```java
public int method(boolean b) {
    if (b)
        return a1;
    else
        return a2;
}
```

The method above finds the type `int` in its signature and delegate this to the two return statements, and `a1` and `a2` both get target type `int`.

This means lambda expressions must also delegate a target type to the return statements in a lambda body in a similar fashion. A lambda does not have any explicit return type though, so the relevant type is instead looked up using the targeted function type. The return type in that function type is delegated as target type to the returns inside the lambda. This will also handle the case where a lambda returns another lambda, and each lambda being returned will get the correct target type. For example, see the code below:

```java
interface A<T> { B<T> execute(int a); }
interface B<T> { C<T> execute(int a); }
interface C<T> { D<T> execute(int a); }
interface D<T> { int execute(T a); }

A<String> a = p1 -> p2 -> {
    p3 -> p4 -> {
        return p1 + p2 + p3 + p4.length();
    };
};
```
The target type for the four lambdas above becomes \( A\langle\text{String}\rangle, B\langle\text{String}\rangle, C\langle\text{String}\rangle \) and \( D\langle\text{String}\rangle \) respectively. Target function type for the different lambdas become:

1. \((\text{int})\rightarrow B\langle\text{String}\rangle\) (throws nothing)
2. \((\text{int})\rightarrow C\langle\text{String}\rangle\) (throws nothing)
3. \((\text{int})\rightarrow D\langle\text{String}\rangle\) (throws nothing)
4. \((\text{String})\rightarrow \text{int}\) (throws nothing)

An image of how the target type is computed for the different lambdas in the code above can be found in figure 8.

Figure 8: Example of the target type being computed in a nested lambda.

Computing the target type for expressions in invocation context also require special attention. Invocation context includes method accesses and class instance creation expressions, and the target type thus depends on what method declaration or constructor that is found to match the corresponding access. There is a possibility that no valid method or constructor is found, in which case an unknown type can be set as target type.
6 Lambda Expressions

This section will focus on the semantics of lambda expressions. To make the following sections easier to follow, the type analysis of lambda expressions have been separated from the rest of the semantic rules.

6.1 Semantic Rules

The semantics of lambda expressions can be split into two different parts, one for the lambda parameters and one for the body. The semantic rules concerning the parameters can be found in chapter 15.27.1 in the JLS [1, Ch. 15.27.1], and are almost identical to the rules regarding parameters in normal methods. Some examples can be found below:

// Illegal, can't mix inferred and declared parameters
(int a, b) -> {}

// Illegal, can't use modifiers with inferred parameters
(final a) -> {}

// Illegal, only final is allowed for declared parameters
(public int a) -> {}

// Illegal, can't redefine parameter name
(int a, int a) -> {}

// Illegal, can't shadow local variable
int a = 3;
(int a) -> {}

// Illegal, mixed array notation not allowed with variable arity
(int... a[]) -> {}

The rules for lambda bodies are described in JLS chapter 15.27.2 [1, Ch. 15.27.2], and are quite similar to the rules for method bodies, but there are some differences. The main points are summarized below:

- When the this and super keywords are used in a lambda body, they represent the same thing that they do in the scope outside the lambda, and are thus only legal if the scope where the lambda is defined allows it.

- If every return statement in a lambda block body is empty (does not return any value), the block body is said to be void-compatible.

- If no return statement in a lambda block body is empty, and the block cannot complete normally, then the block body is said to be value-compatible.

- It is illegal for a lambda block body to be neither void nor value-compatible.

- Every local variable or parameter referenced in a lambda body must be either final or effectively final (see below).

- Every local variable that is used but not declared in a lambda body must be definitely assigned before the lambda expression.

As can be seen in the rules above, the concept of effectively final is used for local variables. This concept was briefly introduced already in Java 7, and for a variable to be considered effectively final it must behave like it is final in practice. Or in other words, adding the final modifier in front of an effectively
A parameter or variable with an initializer is considered effectively final simply if it is never assigned to or the operand to a decrement or increment operator. A variable which is not initialized may be assigned to if it is definitely not assigned before that and still be considered effectively final. It is not allowed to appear as the operand in a decrement or increment operator though [1, Ch. 4.12.4].

6.2 Lambda Type Analysis

Apart from the semantic rules mentioned in section 6.1, the compiler also has to perform a type analysis of all lambda expressions to ensure they are not used incorrectly, in which case an error should be reported. The type analysis for lambda expressions can be roughly divided into the following steps:

- Checking that a lambda is used in a legal context.
- Checking that the lambda’s targeted type is a functional interface type whose function type is compatible with the lambda.
- In case the lambda uses inferred parameters, their types must be inferred and their uses checked.

In case the lambda satisfies the above conditions, the type becomes the same type as the target type. A lambda is legal to define in three different contexts, assignment context, invocation context and cast context. Three examples can be found below:

```java
// Assignment context
FuncInterface f = () -> 5;

// Assignment context
return () -> 5;

// Cast context
FuncInterface f = (FuncInterface)() -> 5;

//Invocation context
someMethod(() -> 5);
```

A lambda defined in one of these contexts will always have a target type (see section 5), which must be a functional interface, which in turn has a function type. The lambda expression must then be congruent to that function type, which is defined as below by the JLS8 [1, Ch. 15.27.3]:

A lambda expression is congruent with a function type if all of the following are true:

- The function type is not generic.
- The number of lambda parameters is the same as the number of parameter types in the function type.
• If the lambda parameters have explicitly-declared types, these types are the same as the parameter types of the function type.

• Where the lambda parameters are assumed to have the same types as the function type parameter types:
  
  - If the function type's return type is void, then the lambda body is either a statement expression or a void-compatible block.
  
  - If the function type's return type is a (non-void) type $R$, then either
    
    i) the lambda body is an expression that is compatible with $R$ in an assignment context, or
    ii) the lambda body is a value-compatible block, and each result expression is compatible with $R$ in an assignment context.

The reason for the restriction on not allowing generic function types is that a lambda cannot declare type variables, so it does not make any sense for it to represent a method declaring type variables. What remains then is to check that the lambda's throws clause is compatible with the thrown types in the function type, which means that any checked exception (expressions which are not a subtype of RuntimeException) that might be thrown by a lambda must be equal to or a subtype of some type in the throws clause of the targeted function type. If the lambda uses inferred parameter types, these types should be inferred directly from the formal parameters in the targeted function type.

6.3 Implementation of Semantics Checking

This section focuses on the implementation of the semantic rules outlined in section 6.1. The implementation of the type analysis explained in section 6.2 is described in its own section.

6.3.1 Name Analysis

The name analysis includes checking that parameters are not redeclared, that variable uses have a declaration in scope and that no illegal shadowing is done. The main part of the name analysis consists of finding the variable declaration corresponding to each variable use, which in JastAddJ is done using an inherited attribute that can look up a specific variable. Calling this attribute results in traversing the tree upwards until a matching declaration is found, which also provides shadowing automatically. The name analysis for lambdas mostly consisted of implementing this variable lookup also for lambdas, and some examples of the data flow can be found in figures 9, 10 and 11.

6.3.2 Void/Value Compatible Analysis

A lambda block body needs to be either void compatible or value compatible, and thus the return statements in a lambda need to be located and checked. Using a collection attribute (see section 2.3), all the return statements for a specific lambda block body are gathered. It is then simple to iterate over the return statements and check each one.
Figure 9: Data flow in name analysis. In this example, a statement using variable v initiates a lookup for the corresponding declaration, and the lambda finds the requested declaration in the lambda parameters.

Figure 10: Data flow in name analysis. In this example, a statement using variable v initiates a lookup for the corresponding declaration, and the lambda does not find the requested variable in the lambda parameters and thus delegate the task further upwards in the AST.

6.3.3 Effectively Final

Local variables and parameters referenced in a lambda but declared in an outer scope must be final or effectively final, or else a compile-time error occurs. In order to implement this, two types of information is required. First, whether a
Figure 11: Data flow in name analysis. In this example, a parameter initiates a variable lookup to check for duplicates. The parent node finds a duplicate and returns it.

local variable or parameter is effectively final or not must be computed. Second, whether a specific use of a variable or parameter creates the requirement of effectively final or final must be computed. For example, a local variable might not be effectively final or final, but if it is only referenced in a context where this is not a requirement no error should occur. Though it is not directly related to lambdas, inner classes are also mentioned in this section. This is because in Java 8, the requirement that variables referenced inside but declared outside the inner class must be final has been relaxed to effectively final instead.

Which variable accesses which introduces the requirement of effectively final can be found by comparing the immediate enclosing lambda of a variable access with the immediate enclosing lambda of the corresponding variable decl. For example, see the code examples below:

```java
int a = 2;
FuncInt f1 = () -> {
    int b = a;
}
```

Access and declaration of `a` above are not enclosed by the same lambda, effectively final is thus required.

```java
int c = 3;
FuncInt f2 = () -> {
    int d = c;
}
```

Access and declaration of `c` above are enclosed by the same lambda, effectively final is thus not required.

Because of this, an attribute was implemented called enclosingLambda which computes the immediately enclosing lambda for a certain node. It was implemented using JastAdd broadcasting (see section 2.3), and an example of this broadcasting can be seen in figure 12.

With the enclosingLambda attribute, it only remains to compute which variables and parameters that are actually effectively final. To check whether a certain variable declaration or parameter declaration is effectively final, the AST is
Figure 12: Shows how the value of enclosing Lambda is broadcast down the tree and changed when a lambda expression is encountered.

First traversed upwards until the node is found that defines the scope of the declaration. This node could for example be a block, a for loop or an if statement. From this node, the search then continues downwards to find potential nodes where the variable in question is modified. An example of when it is checked whether or not a lambda parameter is effectively final can be found in figure 13.

Figure 13: The data flow when a lambda parameter is checked for effectively final. The request traverses up to the root of the lambda expression, which in turn delegates to the lambda block to search for assignments to the parameter.

Inner classes are handled similarly, if the declaration associated with a certain variable access is located in an outer class, the variable must be final or effectively final or else an error is generated.
6.4 Implementation of Type Analysis

This section focuses on the implementation of the type analysis of lambda expressions, as described in section 6.2. The steps that need to be implemented can be summarized as follows:

1. Check whether or not a lambda is located in assignment, invocation or cast context. A compile-time error should be generated if it is located in none of these.

2. Checking the target type for the lambda. If the target type is not a functional interface a compile-time error should be generated.

3. Checking congruency to the target function type. If lambda and target type are found not to be congruent, a compile-time error should be generated.

4. Inferring the types for eventual inferred lambda parameters which then in turn become available for further type checking once their types are known.

Checking that a lambda is located in a valid context, and that the target type is a valid functional type is trivial. What remains then is to check that the lambda is congruent to the targeted function type.

6.4.1 Checking congruency

When the context and target type attributes have been computed for a specific lambda, the lambda can then be checked for type correctness by checking if it is congruent to its targeted function type. The targeted function type is retrieved from the targeted functional interface. Checking congruency when the function type is available is straightforward, and simply means doing the comparisons described in section 6.2. One additional thing must be handled though, see the example below:

```java
interface A<T> { T execute(int a); }

A<String> a = (int b) -> "some string";
```

Obviously, the lambda should be congruent to the targeted function type, since the type variable `T` is explicitly specified as a `String`. The type variable substitution must be done also in the targeted function type, or the lambda will not be considered congruent. Thus, every specific parameterization for a specific generic interface will have its own function type.

When the congruency has been checked to be valid, the type of the lambda expression is simply set to the same type as the target type, or otherwise an unknown type.

6.4.2 Parameter inference

Inferring the type of eventual parameters requiring inference is straightforward when the lambda type analysis is done. Assuming the type of a lambda has been computed and no errors were generated, the lambda will have a type
which corresponds to a functional interface with a function type, and the lambda parameters must then have types which equal the types of the formal parameters in the function type. Thus the types can simply be copied from the function type to the lambda inferred parameters. For example, see the code below:

```java
interface A<T> { B<T> execute(int a); }
interface B<T> { C<T> execute(int a); }
interface C<T> { D<T> execute(int a); }
interface D<T> { int execute(T a); }

A<String> a = p1 -> p2 -> {
    return p3 -> p4 -> {
        return p1 + p2 + p3 + p4.length();
    };
};
```

// Target function type for the different lambdas:
// 1: (int)->B<String> (throws nothing)
// 2: (int)->C<String> (throws nothing)
// 3: (int)->D<String> (throws nothing)
// 4: (String)->int (throws nothing)

Inferring parameters p1 - p4 would then be the same as setting their types to their respective function type parameter types. Thus, p1 - p3 will have the inferred type int, while p4 has the inferred type String. This means the final return will indeed give a result of type int, which matches the expected return type of the final lambda expression.

7 Method and Constructor References

This section focuses on the type analysis of the new method and constructor references.

7.1 Semantic Rules

Method and constructor references can be split into four different cases, which can be seen in the code examples below:

- Method reference with an expression, refers to a method called ‘method’ located in the object referenced by the expression (a[0]):
  `(a[0]):method;`

- Method reference with reference type, refers to a method called ‘get’ located in the type `ArrayList<Integer>`:
  `ArrayList<Integer>::get;`

- Constructor reference with class type, refers to a constructor for the class `ArrayList<String>`:
  `ArrayList<String>::new;`
• Constructor reference with array type, refers to a synthetic constructor which creates new arrays of type int[][]:

    int[][]::new;

There is also support in the syntax for explicit type arguments when referring to generic methods, see the example below, where the type variables T and S are explicitly set to the types Integer and String:

    public <T, S> void method(T t, S s) { }

    this::<Integer, String>method;

The type analysis for method and constructor references mostly consist of locating the correct method or constructor referenced, and to check if it is compatible with the targeted function type. To locate a method or constructor, the Java overload resolution algorithm is used, which is the algorithm that computes which method declaration that should be invoked for a specific method call. One problem with this is that normally, the overload resolution algorithm takes the types of the arguments in the method call as input, for example see the code below where it is the second method declaration being invoked:

    public void method(int i) { }
    public void method(String s) { }
    method("a string");

But neither method references nor constructor references include any arguments, so what types should be input to the algorithm? The compiler infers these argument types from the targeted function type and use those as input. See the example below:

    interface A { void m(String s); }
    public void method(int i) { }
    public void method(String s) { }
    A a = this::method;

The method referenced above is the second method declaration, since the type String is fetched from the method in interface A and used in the lookup.

More generally, the type analysis of method and constructor references are done in the following steps:

• Checking that the method or constructor reference is used in a legal context.

• Checking that the method or constructor reference’s targeted type is a functional interface type whose function type is compatible with the reference.

Just as for lambdas, the valid contexts for a method or constructor reference are assignment, cast and invocation contexts. To check for compatibility against the function type, once again the concept of congruency is used. A method or constructor reference is considered congruent to a function type if the following conditions hold:
• The method or constructor reference define a single compile-time declaration.

• Either the return type of the function type is void, or the return type of the compile-time declaration is compatible in assignment context to the return type of the function type.

The compile-time declaration mentioned above is the method or constructor found using the overload resolution lookup. As for lambdas, a method or constructor reference satisfying the conditions above will have the same type as its target type [1, Ch. 15.13.2].

One question that remains for the compile-time declaration is where the actual lookup should be performed. We now know how the type arguments used for input to the algorithm are found, but the algorithm must also search a specific type for valid methods or constructors. What type this is varies depending on the type of method or constructor reference. Each case is discussed in the following sections.

7.1.1 Method Reference With Expression/Super

This is the case where the method is referenced by using an expression as left hand side. If the expression does not end with super, then the type where the compile-time declaration should be searched for is found by evaluating the expression, and the resulting reference refers to the type to search. If the expression ends with super, the type referred to by that super access is the type to search. The expected compile-time declaration is an instance method, and should this search result in a static method, a compile-time error is generated [1, Ch. 15.13.1].

7.1.2 Method Reference With Reference Type

When a method is referenced using a reference type as left hand side, two different phases of lookup are performed, which both can lead to finding a compile-time declaration. If both do find one, an ambiguity error is generated.

In the first phase, the type referred to by the left hand side is where the search takes place, and all the arguments from the targeted function type are used for the search. A static method is expected, and should this search result in an instance method, a compile-time error is generated.

The second phase only runs if the targeted function type has at least one parameter, and the type of the first parameter is a subtype of the reference type in the method reference. As in the first case, the left hand side type is where the search is carried out, but this time only using the arguments following the first argument in the function type. Thus, if the first phase uses \( n \) arguments during the search, this step uses \( n - 1 \) arguments. If the reference type is raw, an attempt at inferring a valid parameterization is done by looking at the first parameter in the function type before searching for the method. The search then takes place in the parameterized type. An instance method is expected, and should this search result in a static method, a compile-time error will be generated.
The fact that a method referenced in a reference type and not an instance of that type can result in an instance method might sound strange, and is explained using an example below:

class Test {
    interface A {
        int m(Test t, int i);
    }
    int local = 2;

    public int method(int i) {
        return i + local;
    }

    public static void main(String[] arg) {
        // Method reference
        A a = Test::method;
    }
}

For the method reference above, phase 2 finds an instance method. The lookup was done in the type Test, using the type int as input to the lookup.

The behaviour of the method m in a will become:

public int m(Test t, int i) {
    return t.method(i);
}

As can be seen in the above example, if phase 2 finds an instance method, the resulting instantiation of the functional interface will use the first parameter in the single method to qualify an access to the compile-time declaration (the instance method), and use the following parameters as arguments to this access [1, Ch. 15.13.1].

7.1.3 Constructor Reference With Class Type

For constructor references where the left hand side is a class type, the constructor is simply searched for in that class. In the case that the class type is raw, the type arguments should be inferred using the same algorithm as the ‘diamond’ feature in Java 7 uses. The search is then carried out in the inferred type [1, Ch. 15.13.1]. An example of the diamond feature can be seen below:

// Infers the type argument 'Integer'
List<Integer> a = new ArrayList<>();

7.1.4 Constructor Reference With Array Type

Constructor references using an array type as left hand side are unique since arrays do not use constructors. Thus there is no need to search for a compile-time declaration in this case, since every array type has one implicitly. Instead, it has to be checked that the targeted function type also only uses a single parameter, and that the type of that parameter is assignable to an int. The
return type of the implicit compile-time declaration is of course the array type specified by the constructor reference [1, Ch. 15.13.1].

7.2 Implementation of Semantics Checking

The type analysis of method and constructor references is implemented using the following steps:

- Check that the reference is defined in a legal context
- Check that the target type is a functional interface
- Use the information in the targeted function type to locate a single compile-time declaration
- Check that the compile-time declaration return type is compatible with the function type return type

The checks for context and target type are exactly the same as for lambdas, and will thus not be repeated here. The rest of this section will focus on how the compile-time declaration is found.

As mentioned in section 7.1, the compile-time declaration is searched for using the standard overload resolution algorithm. Since this algorithm is already implemented for the case where a method access locates what method declaration that should be invoked, an easy way to implement the lookup is to reuse this implementation. This can be achieved by creating synthetic method calls where the argument types are taken from the targeted function type, and the algorithm can then be run on these. For this to work, these synthetic method calls must be added as non-terminal attributes, or the overload resolution algorithm can not be run on them.

For the case where explicit type arguments are provided, a parameterized method call can be created instead. Also, for the case with reference type method references, where two different lookups are performed, two different synthetic method calls must be constructed. Some examples can be found below:

```java
class Test<S> {
    interface A { void m(int a); }
    interface B { void m(String a); }
    interface C { void m(Test<Integer> t); }

    public void method(int a) { }
    public <T> void method(T t) { }
    public void method() { }

    // #1
    A a = this::method;

    // #2
    B b = new Test()::<String>method;

    // #3
    C c = Test::method;
}
```
For method reference 1 above, a synthetic method call is created that looks like this: `this.method(int)`.

For method reference 2 a synthetic method call is created that looks like this: `new Test.<String>method(String)`.

For method reference 3, two synthetic method calls are created, `Test.method(Test)` and `Test<Integer>.method()`.

The same strategy can be used for constructor references, except that instead of creating a synthetic method call, a synthetic class instance creation expression is instead constructed. Some examples can be found below:

class Test {
    class Inner {
        public <T> Inner(String a) { }
    }
    interface A { ArrayList<Integer> m(); }
    interface B { Inner m(String a); }
    interface C { ArrayList<String> m(int a); }

    // #1
    A a = ArrayList<Integer>::new;

    // #2
    B b = Inner::<String>new;

    // #3
    C c = ArrayList::new;
}

For constructor reference 1 above, a synthetic class instance creation expression is created that looks like this: `new ArrayList<Integer>()`.

For constructor reference 2 a synthetic class instance creation expression is created that looks like this: `new<String> Inner()`.

For constructor reference 3 a synthetic class instance creation expression is created that looks like this (note the diamond): `new ArrayList<>[int]`.

An example of how a complete constructor reference with the new NTA can look like can be seen in figure 14.

Array references are a special case since they do not have a constructor, and no NTA is built. The properties of the implicit compile-time declaration can be taken from the array reference instead, and a check can be carried out to see that the targeted function type is compatible with the implicit compile-time declaration.

### 8 Interface Methods

This section focuses on the new interface methods. The semantics revolve around the new default methods, but the fact that interface methods can now be static must also be taken into account.
8.1 Semantic Rules

The semantic analysis for interface methods can be roughly divided into two parts: checking that the modifiers are compatible with the method format, and checking inheritance/overriding related things. The rules concerning the modifiers are fairly simple; the main principle is that a method can only be abstract, static, or default, and thus the modifiers specifying these cannot be mixed. A method specified as static or default must provide a body, while an abstract method must not provide a body. The rules regarding the actual body is basically the same as for ordinary methods [1, Ch. 9.4][1, Ch. 9.4.3].

When it comes to inheritance and overriding, the default methods introduce some new problems that were not present before, related to multiple inheritance. For example, see the code below:

interface A {
    default void m(int i) { }
}
interface B {
    default void m(int i) { }
}

public class Inner implements A, B {
}

The class Inner above inherits the method $m$ from both interfaces, and a compile-time error is generated. It is thus now possible to inherit multiple methods with the exact same signature, which is caused by the fact that default
methods introduce a form of multiple inheritance in Java. To protect from this, there has to be semantic rules. The basis for these semantics can be found in the rule below:

- It is illegal for a class or interface to contain a default method whose signature equals the signature of another method in the same class or interface, whether or not the second method is abstract or not. It does not matter if the methods are inherited or not.

Then there are specific rules stating in which cases a class or interface inherits a certain method. Thus there are many cases where it is perfectly legal to implement several interfaces each containing a default method with the same signature. The point is that as long as only one of those are actually inherited, there is no conflict. The meat of the semantic rules in this section is thus related to when a class or interface inherits or does not inherit a certain method, and eventual conflicts then depend on whether or not a duplicate can be found amongst the inherited methods. Static interface methods are never inherited, neither by classes nor by interfaces. Whether or not a default method is inherited is then controlled by overriding. The main rules are specified below [1, Ch. 9.4.1]:

- If a class implements or an interface extends another interface containing a default method \( m_1 \), but there already exists a method \( m_2 \) in the class or interface which is a subsignature of \( m_1 \), then \( m_1 \) is not inherited.

- If a class implements or an interface extends some other interfaces containing two default methods, \( m_1 \) and \( m_2 \), and \( m_2 \) already overrides \( m_1 \) in some interface, then only \( m_2 \) is inherited.

An example of the second rule can be found below:

```java
interface A {
    default void m(int i) { }
}
interface B extends A {
    default void m(int i) { }
}
public class Inner implements A, B {
}
```

The class `Inner` above now only inherits \( m \) from \( B \), since this method already overrides \( m \) from \( A \). Thus there is no error.

For classes, there is one more rule:

- If a class implements an interface containing a default method \( m_1 \), and extends a class containing a method \( m_2 \), and \( m_2 \) is a subsignature of \( m_1 \), then only \( m_2 \) is inherited.

This rule makes sure methods inherited from classes have priority over interface methods [1, Ch. 8.4.8].

When it comes to shadowing static interface methods, it is illegal for a static method to have a signature which is a subsignature of some inherited method. Since static interface methods are not inherited, there is no problem in declaring
a method with a subsignature of a static interface method in a superinterface [1, Ch. 9.4.1].

When a default method has been overridden, it can still be accessed by using the super keyword. The syntax looks like: Interface.super.method(), see the example below:

```java
interface A {
    default void m(int i) {
    }
}

interface B extends A {
    default void m(int i) {
        // Will invoke the method 'm' in A
        A.super.m(i);
    }
}
```

This is a new usage of the super keyword. Before, a qualifier used with super meant to access the superclass of whatever that was used as qualifier, but in the above example it means to access the interface that is used as qualifier. The qualifier in this case must be a direct superinterface or implemented interface, but even then it is not always legal. There can be no other superinterface or implemented interface that extends the interface being qualified, in that case a compile-time error is generated. Basically, it is not possible to 'bypass' any other interface when using this syntax [1, Ch. 15.12.1]. See the examples below:

```java
interface A {
    default void m(int i) {
    }
}

interface B extends A {
    default void m(int i) {
    }
}

interface C extends B {
    default void method(int i) {
        // Not legal, A is not a direct superinterface
        A.super.m(i);
    }
}

class Inner implements A, B {
    public void method(int i) {
        // Not legal, B extends A and cannot be bypassed
        A.super.m(i);
    }
}

class Inner2 implements A {
    public void method(int i) {
        // legal
    }
}
8.2 Implementation of Semantics

The implementation of the semantic analysis of interface methods was fairly straightforward. Checking that no modifiers conflict in an illegal way is trivial and will not be discussed. Whether an interface method is abstract or not was extremely trivial before (all interface methods were abstract), but requires a simple check in Java 8 (which is also very trivial). An interface method which has neither a static nor a default modifier is automatically considered abstract. There was already a check in JastAddJ which reports errors if an abstract method declared a body, or if a non-abstract method does not declare a body, so simply updating the requirements for when a method is abstract automatically provides the relevant error checks.

In JastAddJ, inherited methods are collected in a hash map data structure, and the rest of the analysis was mostly to make sure only methods that are actually supposed to be inherited according to the rules in section 8.1 are inserted into this data structure. A check is then performed to make sure that there are no illegal conflicts between the methods in the map. The type lookup for the super keyword also needed to be modified, since a super access qualified by an interface should result in an access to the actual interface. All direct superinterfaces or direct implemented interfaces also need to be analyzed in order to make sure the use of a qualified interface with super is actually legal, as specified in section 8.1.

8.3 Bytecode Changes

In order for the new interface methods to be executable, changes to the Java virtual machine specification are also included in Java 8. Before Java 8, all interface methods were required to set a certain flag in the bytecode representing the abstract property, but that is not required anymore and the Java Virtual Machine will consider interface methods without this flag callable. For class files of a version below 52 (the Java 8 version), it is still required that all interface methods set the abstract flag [3, Ch. 4.6].

This also requires changing how the standard library is parsed. When classes from the Java standard library are imported, JastAddJ must read those classes from the standard folder where the library is located. These classes are supplied as class files, and thus in bytecode. Because of this JastAddJ includes a bytecode reader, which basically reads bytecode and translates it to parts of the AST so that it can be included in the semantic analysis. This bytecode reader was changed so that it can now mark certain methods from the standard library as default when they were neither static nor abstract.

9 Method Overload Resolution Algorithm

Java 8 includes a large update of the old overload resolution algorithm, which will be discussed in this section. This is the algorithm that finds the cor-
rect method declaration or constructor declaration corresponding to a certain method access or constructor access.

9.1 Problems With the Old Algorithm

The old algorithm was completely dependent on the types of the arguments as input to the overload resolution algorithm, which does not work any longer with the new Java 8 language constructs [2, Ch. 15.12]. For example, see the code below:

```java
interface A { void m(int i); }
interface B { void m(int i, String s); }

public int method(A a) { return 4; }
public int method(B b) { return 5; }

int i = method((int i1) -> { });
```

Using the old algorithm, the type of the lambda argument must be computed before the algorithm can be run. But remembering what was stated in section 6.2, the type of the lambda is dependent on its target type, which will be either A or B depending on the result of the overload resolution algorithm. This creates a circular dependency which the old algorithm simply does not handle. The exact same problem also applies for method and constructor references, whose types are also dependent on the target type. Thus, the new algorithm needs to be constructed in a way such that types are only used where it is safe to do so.

This is achieved by using the property of poly expressions introduced in section 5.6. Since only poly expressions are dependent on their target type, the type of standalone expressions can safely be computed even in the middle of the overload resolution algorithm.

9.2 Algorithm Description

The new algorithm still carries out the same five steps as the old algorithm, but the details of each step has been modified. The five steps, which will all be explained in more detail, are the following:

- **Step 1**: Gather all potentially applicable methods, which are methods that are candidates for being chosen.
- **Step 2**: Find all methods which are applicable by strict invocation, and if at least one such method was found go to step 5.
- **Step 3**: Find all methods which are applicable by loose invocation, and if at least one such method was found go to step 5.
- **Step 4**: Find all methods which are applicable by variable arity invocation.
- **Step 5**: If more than one method was found before arriving at this step, choose the most specific method amongst the remaining ones.
Note that these steps also include type inference for generic methods in case that is required, but this section will focus on methods which do not require type inference to be performed. Also, the descriptions below will focus on methods, but work exactly the same when looking up constructors [1, Ch. 15.12.2].

9.2.1 Step 1: Gathering the Potentially Applicable Methods

The point of this step is to gather a selection of methods which have properties that make them likely candidates for being chosen as the method to be invoked, and thus some methods which are obviously wrong are rejected at this stage. In Java 7, gathering potentially applicable methods consisted of gathering all methods with the same name and compatible arity to the method access [2, Ch. 15.12.2.1]. For example, see the code below:

```java
// Method #1
public void method(int i1) { }
// Method #2
public void method(String s, int i) { }
// Method #3
public void method(float f, int... i) { }
```

// Both methods 1 and 3 are potentially applicable
method(2);

// Both methods 2 and 3 are potentially applicable
method(2, 3);

// Only method 3 is potentially applicable
method(2, 3, 4);

In Java 8, this has been expanded by introducing a new property which defines when a certain argument expression is considered **potentially compatible** with a type. Note that the above states ‘argument expression’ and not ‘argument type’, which is because poly expressions are not safe to compute the type for.

The point of this property is to let certain poly expressions remove some methods early from the algorithm which are obviously wrong because the structure of the poly expression does not match the type. For example, a lambda or method reference is not potentially compatible with a type which is not a functional interface, and a lambda is not potentially compatible with a functional interface whose function type has incompatible arity or return type. A method or constructor reference is not potentially compatible with a functional interface if there are no potential compile-time declarations at all that can be found using the argument types in the corresponding function type. Standalone expressions are always considered potentially compatible. Some examples can be found below:

```java
interface A { void m(int i); }
interface B { int m(String s); }

public void m(int i1, int i2) { }
```

// Method #1
public void method(A a) { }
// Method #2
public void method(B b) { }

Only method 2 is applicable for the method call below, since the void return type in A is not compatible with the lambda:

```java
method(p -> p);
```

For the method call below, no method is potentially applicable since no compile-time declaration can ever be found using the argument types in A or B:

```java
method(this::m);
```

In Java 8 it is thus not enough any more to simply compare arities, it has to be checked that every argument expression in the method call is potentially compatible to the corresponding parameter type in the method declaration being considered [1, Ch. 15.12.2.1].

### 9.2.2 Step 2: Check Applicability by Strict Invocation

A new property has been introduced in Java 8 which defines when a certain expression is pertinent to applicability or not. The point of this property is to mark certain poly expressions as too vague to let the overload resolution algorithm depend on them, since these expressions depend too much on their target types.

Expressions are in general considered pertinent to applicability, but with some exceptions defined. Lambda expressions are considered not pertinent to applicability if they use inferred lambda parameters, or if they somewhere return an expression which is in turn not considered pertinent to applicability. Method and constructor references are considered pertinent to applicability only if they are exact [1, Ch. 15.13.1], which means that no matter what target type the method reference uses when looking up the compile-time declaration, there can only exist exactly one possible compile-time declaration.

This step must then check that all arguments can be assigned to their respective targeted type in a strict invocation context. This means that boxing is not allowed, so for example the type int is not compatible with Integer in a strict invocation context [1, Ch. 5.3]. For standalone expressions, the type is simply computed, while for lambdas and method references, a congruency check is performed to check for compatibility. Any argument which is not pertinent to applicability will be passed to the next step for further checking [1, Ch. 15.12.2.2].

### 9.2.3 Step 3: Check Applicability by Loose Invocation

This step is exactly the same as checking for applicability by strict invocation, except that the checks performed only require the argument to be compatible with its targeted type in a loose invocation context. This means that boxing and unboxing is now allowed, and int for example would be compatible with Integer in a loose invocation context [1, Ch. 5.3][1, Ch. 15.12.2.3].
9.2.4 Step 4: Check Applicability by Variable Arity Invocation

This step is only executed if the potentially applicable method being considered is a variable arity method. In practice, this step is exactly like the loose invocation check except that the variable arity parameter in a variable arity method is also considered [1, Ch. 15.12.2.4]. For example, see the code below:

```java
public void m(String s, int... i) { }
```

m("string", 3, new Integer(2));

The method m above is applicable by variable arity invocation.

9.2.5 Step 5: Choose Most Specific Method

Assuming steps 2-4 found exactly zero or one applicable method, then the algorithm will immediately succeed or fail after that, but there is a possibility of several methods being considered applicable. In that case, the most specific one is chosen. In Java 7, this check was done by simply comparing two methods pairwise to each other and performing a subtype check of the parameter types [2, Ch. 15.12.2.5]. For example, see the code below:

```java
public void m(List<Integer> l) { }
public void m(ArrayList<Integer> l) { }
```

m(new ArrayList<Integer>());

Both methods above are applicable, but since ArrayList is a subtype of List, the second method is chosen as the most specific.

In Java 8, the type can not be computed for poly expressions, and thus the above check does not work for all types of expressions anymore. Just ignoring every poly expression would not be very flexible, and there are several cases where a certain method is an obvious better choice depending on the actual poly expression. To account for this, in Java 8 a certain type can be considered more specific than another type for a certain expression. In other words, a certain type can sometimes be considered more specific than another type, or sometimes not, depending on the argument expression being considered.

If type A is a subtype of type B, then A is always considered more specific than B regardless of the expression being considered. Adding to this, in the case there is no subtype relationship between A and B, one can still be considered more specific if both are functional interface types and the expression being considered is a lambda with declared parameters or an exact method or constructor reference. In this case, the return types of the function types in the two functional interface types being compared are used. Some examples can be found below:

```java
interface A { int m(int a); }
interface B { Integer m(int a); }
```

```java
public Integer method(int a) { return a; } // Method #1
public void m(A a) { } // Method #2
```

58
public void m(B b) { }

Since the return type for the method call below is a primitive type, which is more compatible with A than B, method 1 is considered most specific:

m((int a) -> a);

For the method call below, since the return type of the only possible compile-time declaration for the method reference is a reference type, which is more compatible with B than A, method 2 is considered most specific:

m(this::method);

The checks are then carried out similarly to Java 7, but with the argument expressions also being considered. When a most specific method has been chosen, it must also be checked that each argument is actually compatible with its target type. For example, expressions which were not considered pertinent to applicability have not influenced the algorithm very much, and can easily get an incompatible target type. See the example below:

interface A { String m(String s, int i); }
interface B { int m(String s, int i); }

public int method(A a, int i) { return 3; }
public int method(B b, Integer i) { return 4; }

int i1 = method((p1, p2) -> p1, new Integer(4));

In this case, since the lambda is not considered pertinent to applicability, the second method will be chosen by applicability in strict invocation context. The lambda will then get target type B, which is not a valid target type for the lambda in question. Had instead declared parameters with types String and int been used, the lambda would have helped influence the algorithm to not pick the second method, and instead pick the first method by applicability in loose invocation context, resulting in the valid target type A [1, Ch. 15.12.2.5].

9.3 Algorithm Examples

This section will provide some concrete examples showing different aspects of the algorithm. See the code below:

interface A { void m(String s, int i); }
interface B { int m(int i); }
interface C { Integer m(int i); }

public int m(int i) { return i; }
public int m2(int i) { return 4; }
public String m2(Integer i) { return "h"; }

// Method 1
public void method(A a) { }

// Method 2
public void method(B b) { }
public void method(C c) {

public void test() {
    // Access 1
    method((s, i) -> System.out.println(s + i));

    // Access 2
    method(this::m);

    // Access 3
    method(this::m2);

    // Access 4
    method((int i) -> new Integer(i + 4));

    // Access 5
    method(i -> System.out.println(i));
}

For the first method access above, only method 1 becomes potentially applicable. Method 1 is then chosen by compatibility in strict invocation context.

For the second one, methods 2 and 3 are both potentially applicable. Methods 2 and 3 are then both chosen by compatibility in strict invocation context. Since the method reference is exact, the compile-time declaration helps to choose the most specific method, which becomes method 2, since the compile-time declaration has primitive return type.

For the third one, methods 2 and 3 are both potentially applicable. The method reference is not exact and thus not pertinent to applicability. Both methods are thus immediately chosen in strict invocation context. Since the method reference is not exact, it can not help to choose a most specific method. Two methods are thus most specific, this results in a compile-time error.

For the fourth one, methods 2 and 3 are both potentially applicable. Both are then chosen in strict invocation context. The lambda uses declared parameters, and can thus help choose a most specific method. Method 3 is chosen since the lambda expression body has a reference type.

For the final one, no method is potentially applicable, and this results in a compilation error.

9.3.1 Overload Resolution and Diamond in Java 8

An interesting effect of the new target types and overload resolution algorithm can be found in the diamond feature introduced in Java 7, which is a way to infer what type arguments to use in a class instance creation expression, see the example below:

// The type 'String' is inferred
List<String> = new ArrayList<>();
In Java 7, a class instance creation expression using a diamond could not infer its type arguments from a method access, see the example below [2, Ch. 15.9.3]:

```java
public int method(ArrayList<String> a) { return 3; }
```

```java
// Type argument inferred to Object, compilation error
int i = method(new ArrayList<>());
```

In Java 8 on the other hand, the above is legal. The reason is because the diamond class instance creation expression is a poly expression, so the method overload resolution algorithm will complete before the type of the diamond is even evaluated. The class instance creation expression will then get `ArrayList<String>` as target type, which is used to infer the constructor used.

### 9.4 Implementation

The implementation is overall fairly straightforward, and simply follows the rules outlined in the JLS8. This section will thus not describe everything in detail. Some points of interest are instead summarized below:

- The analysis is dependent on not computing the type of poly expressions, and thus there was a need to implement an attribute which checks whether an arbitrary expression is a poly expression or not.
- For exact method and constructor references, there is a need to compute the only possible compile-time declaration without running the complete overload resolution algorithm. An attribute was thus implemented which returns the only compile-time declaration assuming there is only one, and an unknown declaration if there was zero or more than one found. The attribute that computes whether a method or constructor reference is exact then simply calls the first attribute and checks whether it returns an unknown declaration or not.
- There are several cases where every return statement in a lambda body must be examined. To do this, the collection attribute described in section 6.3.2 is reused to iterate over all the return statements.
- To execute the algorithm, checking lambdas and method/constructor references against arbitrary function types for congruency must be possible. The congruency check has thus been implemented taking this function type as a single parameter. For method and constructor references, this means that a synthetic access must be built for every function type that is checked against. There is also the problem that for a lambda, different targeted function types result in different inferences for eventual inferred parameters. Since lambdas which are not pertinent to applicability are never checked against arbitrary function types, this has simply been ignored in the implementation.

### 10 Code Generation of Java 8

When the semantic and type analysis is complete, the next step is to generate Java bytecode assuming no errors occurred in the front end. This section de-
scribes how the Java 8 back end has been implemented for JastAddJ, and what 
rules and restrictions for the code generation that is specified in the JLS8.

10.1 Lambda Expressions

The JLS8 specifies that the runtime evaluation of a lambda expression should 
result in a reference to a synthetic class, whose properties matches the targeted 
functional interface and lambda expression. The synthetic class should imple-
ment the targeted functional interface and, if the target type is an intersection 
type, every other interface mentioned in the intersection. The class should also 
declare a method which overrides the abstract methods in the functional 
interface, and this method should contain the lambda body. If the lambda body 
is an expression, it should translate to either a return statement or a simple 
evaluation depending on the return type in the function type [1, Ch. 15.27.4].

The code generation then mainly consists of creating and instantiating anon-
ymous classes which follows the above rules. An example of how a synthetic class 
generated from a lambda expression can look like can be seen below:

class Test {
    interface A {
        int m(int a, String s) throws IOException;
    }
    A a = (a, s) -> s.length + a;
}

// The above lambda becomes an instance of the following class:
class AnonymousLambda implements A {
    public int m(int a, String s) throws IOException {
        return s.length + a;
    }
}

A lambda expression should thus behave almost exactly like an instantiation 
of an anonymous class, with the exception that the this and super keywords 
should behave differently. See the example below:

class Test {
    int i = 3;
    Runnable r1 = new Runnable() {
        int i = 2;
        public void run() {
            System.out.println(this.i + "");
        }
    }
    Runnable r2 = () -> {
        int i = 1;
        System.out.println(this.i + "");
    };
}

In the method 'run' above, this.i refers to the variable in the anonymous class, 
which is the one with the value 2. In the lambda though, this.i instead refers 
to the variable in the outer scope, which is the one with the value 3.

62
In order for the lambda expressions to evaluate to references of anonymous classes, it would be useful to represent them internally as a synthetic class instance creation expression, defining an anonymous class constructed according to the specifications in the beginning of this section. This can be achieved by adding an NTA-attribute to the lambda expressions returning a class instance creation expression. Assuming this attribute is correctly built, all the lambda has to do is then to delegate the relevant bytecode generation functions to the attribute.

The actual construction of the anonymous class is fairly straightforward, since all necessary information is available in the lambda expression together with the function type. The only special cases are lambdas with inferred parameters, which need to translate to normal parameter declarations in the method inside the anonymous class, and lambdas with expression bodies which need to translate either to an expression statement or a return statement depending on the return type in the function type. This automatically handles nested lambdas too, since code will be generated so that one anonymous lambda class will return a reference to a new instance of another anonymous lambda class. An example of how such an anonymous lambda class looks like can be found in figure 15.

10.1.1 Drawbacks and Improvements

There are some drawbacks of using the strategies mentioned above that should be handled. Consider when a lambda body is transformed to an anonymous class, the whole body, which could potentially be long, is explicitly copied to the method in the anonymous class. In other words, there will now exist two
complete copies of the same method body. In the case of a single, not nested lambda, the only drawback is that it might be unnecessarily slow to create a new copy of already existing data structures. In the case of nested lambda expressions though, redundant copies are made which gets worse the thicker the nesting is. The effects of this can be seen in figure 16.

In general, the body of a lambda nested in \( x \) layers will also be copied \( x \) times, which is obviously a complete waste. The main argument for why existing data should be copied instead of moved is because it is safer than manipulating the AST manually by moving around nodes. Cached attributes are also flushed in the copied nodes which is a requirement. In this case though, once the frontend is done with the semantic and type analysis, the lambda body will not be accessed again, and there is a method that can be called to recursively flush a subtree of choice. Thus it works to simply move the contents of the lambda body to the anonymous class, and then manually flush the attributes. Since the flushing is recursive it only needs to be performed once, and a check can thus be added which makes sure that only lambda expressions which are not nested in other lambda expressions actually call the flush method. The moving strategy creates zero redundant copies of the data.

### 10.2 Method and Constructor references

The code generation of method and constructor references is similar to that of lambda expressions in that a synthetic, anonymous class should be created in both cases. The rules regarding the structure of the synthetic class is also pretty much the same, and just as for lambda expressions the code generation has been
implemented by creating an NTA with a class instance creation expression. The main difference is what the method in the anonymous class should do, which will differ depending on what type of reference that has been found. As with the semantic analysis, the four different cases mentioned in section 7.1 should behave in different ways, and have thus been implemented in different ways. For all different method and constructor references, whether the return type in the function type is void or not determines whether or not the result from calling the compile-time declaration should return a value or not [1, Ch. 15.13.3].

10.2.1 Method Reference With Expression/Super

The JLS8 states that for method references with expressions or super as left hand side, the method in the resulting anonymous class should simply call the compile-time declaration using the method parameters as arguments to the method call. Whether or not the result from that method call should be returned is determined by whether or not the return type in the function type is void or not, as mentioned above.

What is unique for the method references with expressions is that the left hand side in this method reference may contain an expression and not only a type, which means this expression has to be evaluated before the compile-time declaration can be accessed. The JLS8 states that the left hand side expression should be evaluated once when the method reference is first evaluated, and any subsequent calls to the method in the resulting instantiation of the anonymous class should not re-evaluate this expression [1, Ch. 15.13.3]. Thus, the method in the anonymous class cannot simply contain a method access qualified with the left hand side expression, since that would result in that expression being re-evaluated every time the method is called. Instead, a synthetic field can be created. Since a field is initialized only once when the class is instantiated, this will result in the correct behaviour [1, Ch. 8.3.2].

An example of how the synthetic class will look like in this case can be seen in the code below:

```java
class Test {
    interface A {
        int m(String s);
    }

    public int method(String s) {
        return s.length();
    }

    public static Test getTest() {
        return new Test();
    }

    public static void main(String[] arg) {
        A a = getTest()::method;
        a.m("A string");
    }
}
```
The method reference above will result in an instance of the following synthetic, anonymous class, where #1 is the generated name for the field (so that it will not conflict with any parameters in the method):

```java
class AnonymousMethodReference implements A {
    Test #1 = getTest();
    public int m(String s) {
        return #1.method(s);
    }
}
```

In case the left-hand side contains the keyword `super` it is instead a type being referred to and the whole expression must then be used as qualifier inside the method, and no field is created.

### 10.2.2 Method Reference With Reference Type

The synthetic class for method references with reference type should be created differently depending on if the compile-time declaration was found in phase 1 or phase 2 (see section 7.1.2). In the case of phase 1, the method in the anonymous class should simply call the referenced method with arguments taken from the parameters, qualified by the left-hand side reference type. For example, see the code below:

```java
class Test {
    interface A {
        void m(int i, double d);
    }

    public static int method(long l, double d) {
        return 4;
    }

    public static void main(String[] arg) {
        A a = Test::method;
        a.m(2, 2.0);
    }
}
```

The method reference above will result in an instance of the following synthetic, anonymous class:

```java
class AnonymousMethodReference implements A {
    public void m(int i, double d) {
        return Test::method;
    }
}
```

When the compile-time declaration is found during phase 2, the resulting synthetic class is a bit different. Instead of simply qualifying the method with the reference type (which would be wrong, since phase 2 finds an instance method), the first parameter in the method becomes the qualifier [1, Ch. 15.13.3]. See the example below:
class Test {
    interface A {
        String m(B<Integer> b, int i);
    }
    class B<T> {
        String method(T t) {
            return t.toString();
        }
    }

    public static int method(long l, double d) {
        return 4;
    }

    public static void main(String[] arg) {
        // Type of B is inferred here to B<Integer>
        A a = B::method;
        B<Integer> b = (new Test()).new B<Integer>();
        System.out.println(a.m(b, 4));
    }
}

The method reference above will result in an instance of the following synthetic, anonymous class:

class AnonymousMethodReference implements A {
    public String m(B<Integer> b, int i) {
        return b.method(i);
    }
}

10.2.3 Constructor Reference With Class Type

For class references, the synthetic class should contain a method which returns a new instance of the class being referred to. This means that the method in the anonymous class should contain a class instance creation expression with eventual arguments taken from the method parameters [1, Ch. 15.13.3]. An example can be seen below:

class Test {
    interface A {
        ArrayList<Integer> m(int i);
    }

    public static void main(String[] arg) {
        A a = ArrayList<Integer>::new;
        ArrayList<Integer> list = a.m(4);
    }
}

The constructor reference above will result in an instance of the following synthetic, anonymous class:
class AnonymousConstructorReference implements A {
    public ArrayList<Integer> m(int i) {
        return new ArrayList<Integer>(i);
    }
}

10.2.4 Constructor Reference With Array Type

Array references are very similar to class references, but should return a new instance of the appropriate array instead of class [1, Ch. 15.13.3]. Also, no type inference needs to be performed. An example can be found below:

class Test {
    interface A {
        int[][][] m(int i);
    }

    public static void main(String[] arg) {
        A a = int[][][]::new;
        int[][][] i = a.m(40);
    }
}

The constructor reference above will result in an instance of the following synthetic, anonymous class:

class AnonymousConstructorReference implements A {
    public int[][][] m(int i) {
        return new int[i][][];
    }
}

11 Evaluation

In this section, the resulting implementation is evaluated and the questions from section 1 are discussed.

11.1 Implementation Completeness and Modularity

In this section, the following questions from section 1 are discussed:

- Is the implementation complete? If not, what features remain to be implemented?
- To what extent could the new features be added modularly?

An overview can be found in table 1. As can be seen in the table, approximately half of the implemented features could be added completely modularly, while the other half required some changes of varying degree to pre-existing attributes. While the implementation of the overload resolution algorithm required the most refines of old attributes, most of these completely redefine the
old attributes and should probably not be updated anyway if the old attributes were to change, and can thus be considered fairly modular even though it refined a large part of the old algorithm.

The least modular part of the implementation is the semantic analysis of the new interface methods. This required several refines of old attributes, where only small selected parts were changed. This is not very good since changes to the old attributes will not affect the Java 8 module, but it was hard to avoid since many rules regarding interfaces and classes are updated in Java 8.

A large part of the new features has been implemented, but there was not enough time to implement everything. The most important (and largest) feature that was not implemented is the update to the type inference done when invoking generic methods. The result of this is that it is currently not possible to invoke generic methods with lambdas and method references as arguments if they need to affect the type inference.

The reason for the question mark in the table is because it was impossible to properly test the code generation of the new interface methods. Unless the major version in the generated class files are explicitly set to 52 or above, the Java virtual machine will not accept default methods in the bytecode. But if the major version is set to 51 or higher, the Java virtual machine requires that a type safety mechanism called stack map tables are implemented, or it will not run the code. Stack map tables are currently not implemented in JastAddJ, and because of this it was impossible to test code generation for the new interface methods. Very small examples that ran correctly were possible to produce, but not enough to draw any conclusions.

11.2 Correctness

In this section, the following question from section 1 will be discussed:

- Are the implemented features correct? How extensively does the test suite cover Java 8, and how many of these tests do JastAddJ pass?

To test the correctness of the implemented features, an extensive testing suite was developed together with the implementation. The test suite contains 888 tests written specifically to test the new Java 8 features, and this section will try to present an overview of what parts of Java 8 that are covered. This section will not present every special case covered by every test, since reading this section would then become the same amount of work as simply reading the test files. Out of the tests 113 are parsing tests, which were useful in the beginning of the work but have become redundant due to all the other tests. They are thus not included in the analysis. Note that the test suite tests both that compilation passes when a condition is met, and also that compilation fails when it is not met.

A summary of the different categories of tests written together with completion rate for both JastAddJ and javac can be found in table 2. For more specific details of test coverage for each category mentioned in table 2, see appendix A.

As can be seen in table 2, neither JastAddJ nor javac can complete all tests. The bugs in JastAddJ which cause some tests to fail have been located and reported. The test cases which javac fails were examined carefully, but in some cases they were deemed to be correct and bug reports were submitted to Oracle.
<table>
<thead>
<tr>
<th>Java 8 feature</th>
<th>JLS</th>
<th>Implemented</th>
<th>Modularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanning, Parsing</td>
<td>-</td>
<td>Yes</td>
<td>Some parser directives added to the Java 4 and 5 parsers, otherwise modular</td>
</tr>
<tr>
<td>Computing functional interfaces and function type</td>
<td>9.8</td>
<td>Yes</td>
<td>Modular</td>
</tr>
<tr>
<td>@FunctionalInterface annotation</td>
<td>9.6.3.8</td>
<td>Yes</td>
<td>Had to refine interface name check</td>
</tr>
<tr>
<td>Computing contexts, target types and poly expressions</td>
<td>5</td>
<td>Yes</td>
<td>Modular</td>
</tr>
<tr>
<td>Lambda semantics</td>
<td>15.27.1, 15.27.2</td>
<td>Yes</td>
<td>Had to change the Variable interface and refine some name checks</td>
</tr>
<tr>
<td>Inner class final variable constraint relaxed to effectively final</td>
<td>8.1.3</td>
<td>Yes</td>
<td>Had to refine variable name check</td>
</tr>
<tr>
<td>Lambda type analysis</td>
<td>15.27.3</td>
<td>Yes</td>
<td>Modular</td>
</tr>
<tr>
<td>Lambda code generation</td>
<td>15.27.4</td>
<td>Yes</td>
<td>Had to refine super and this type analysis</td>
</tr>
<tr>
<td>Method reference semantic checks</td>
<td>15.13</td>
<td>Yes</td>
<td>Modular</td>
</tr>
<tr>
<td>Method reference type analysis</td>
<td>15.13.2</td>
<td>Yes</td>
<td>Modular</td>
</tr>
<tr>
<td>Method reference code generation</td>
<td>15.13.3</td>
<td>Yes</td>
<td>Modular</td>
</tr>
<tr>
<td>Interface methods semantic checks</td>
<td>9.8.4.8</td>
<td>Yes</td>
<td>Had to refine several attributes</td>
</tr>
<tr>
<td>Overload resolution algorithm</td>
<td>15.12.2</td>
<td>Yes</td>
<td>Had to refine the old algorithm</td>
</tr>
<tr>
<td>Handling the new bytecode updates</td>
<td>JVMS 4.6</td>
<td>?</td>
<td>Had to refine the old bytecode reader</td>
</tr>
<tr>
<td>Type inference updates</td>
<td>18</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Intersection types</td>
<td>4.9</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Wildcard parameterized functional interfaces</td>
<td>9.8</td>
<td>No</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: A table showing the completeness and level of modularity of the Java 8 implementation
<table>
<thead>
<tr>
<th>Test category</th>
<th>nbr Tests</th>
<th>JastAddJ fails</th>
<th>javac fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional interface tests</td>
<td>102</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Lambda expression semantic analysis tests</td>
<td>233</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lambda expression type analysis tests</td>
<td>79</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Lambda expression runtime tests</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Method reference semantic tests</td>
<td>79</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Constructor reference semantic tests</td>
<td>47</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Method reference runtime tests</td>
<td>24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Constructor reference runtime tests</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>New interface methods semantic analysis</td>
<td>90</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Overload resolution semantic analysis</td>
<td>47</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Overload resolution runtime tests</td>
<td>29</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>All tests</td>
<td>775</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 2: Overview of all categories of tests, with number of fails per category both for JastAddJ and javac 1.8.0_5
Table 3: Shows the increase of lines of code in both JastAdd code and the generated code. Both have been measured using SLOCcoun.

<table>
<thead>
<tr>
<th>Code type</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>JastAdd</td>
<td>4500</td>
</tr>
<tr>
<td>Generated</td>
<td>18000</td>
</tr>
</tbody>
</table>

1 One of the test cases which javac fails was because of a bug already reported in the bug database. 2

All in all the test suite should provide a fairly solid coverage of the implemented features, but it is impossible to say for sure exactly how correct the implementation is. Some things that have not been tested are the following:

- Runtime tests of the new interface methods. Since trying to run the new interface methods could never be done, runtime tests were not highly prioritized and in the end never done.

- Testing the overload resolution algorithm for constructors. Support for constructors was implemented and quickly runtime tested informally. Real test cases for this were never written though.

- The features not implemented in the JastAddJ Java 8 module are not tested at all. For example, method type inference using lambdas or correct behavior of wildcard-parameterized target types are not tested.

11.3 Performance Evaluation

In this section, the following question will be discussed:

- How well does the implementation perform regarding code size, memory usage and compilation time?

Starting with code size, the Java 8 module has been measured using the program SLOCCount, written by David A. Wheeler, which measures the code size without including whitespace or comment lines. The result is 4500 lines of non-whitespace code. Though this is the Java code written as JastAdd attributes, and the actual code size of the generated compiler is different. The generated Java 7 compiler consists of approximately 90000 lines of code, while the generated compiler with the Java 8 module consists of approximately 108000 lines of code, which is an increase of 18000 lines of code. The results can be seen summarized in table 3.

When it comes to measuring compilation time and memory usage, it would be meaningless to compare with other Java 8 compilers since the JastAddJ module is not a complete implementation, and thus such comparisons have not been made. These are instead compared to the Java 7 implementation to see how much the so far implemented features affect the performance of JastAddJ.

Memory usage is measured in terms of how much heap memory that is required at the minimum for JastAddJ to be able to compile a certain benchmark.

1 http://bugs.java.com/bugdatabase/view_bug.do?bug_id=8037947
2 http://bugs.java.com/bugdatabase/view_bug.do?bug_id=8038996
Table 4: Shows average compilation time for JastAddJ with and without the Java 8 module. The times are computed by taking the average of 40 data points.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Phase</th>
<th>JastAddJ7(s)</th>
<th>JastAddJ8(s)</th>
<th>% increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac</td>
<td>Parsing</td>
<td>1.29</td>
<td>1.09</td>
<td>-18%</td>
</tr>
<tr>
<td></td>
<td>Error check</td>
<td>7.85</td>
<td>8.16</td>
<td>3.8%</td>
</tr>
<tr>
<td></td>
<td>Code gen</td>
<td>3.08</td>
<td>3.10</td>
<td>0.6%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>12.22</td>
<td>12.35</td>
<td>1%</td>
</tr>
<tr>
<td>JastAddJ</td>
<td>Parsing</td>
<td>1.77</td>
<td>2.11</td>
<td>16%</td>
</tr>
<tr>
<td></td>
<td>Error check</td>
<td>8.84</td>
<td>9.65</td>
<td>8%</td>
</tr>
<tr>
<td></td>
<td>Code gen</td>
<td>4.06</td>
<td>4.26</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>14.67</td>
<td>16.02</td>
<td>8%</td>
</tr>
<tr>
<td>Clojure</td>
<td>Parsing</td>
<td>1.41</td>
<td>1.13</td>
<td>-25%</td>
</tr>
<tr>
<td></td>
<td>Error check</td>
<td>4.98</td>
<td>5.26</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Code gen</td>
<td>1.94</td>
<td>2.44</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>8.32</td>
<td>8.83</td>
<td>6%</td>
</tr>
<tr>
<td>hsqldb</td>
<td>Parsing</td>
<td>2.70</td>
<td>2.50</td>
<td>-8%</td>
</tr>
<tr>
<td></td>
<td>Error check</td>
<td>9.01</td>
<td>9.55</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>Code gen</td>
<td>4.58</td>
<td>4.28</td>
<td>-7%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>16.30</td>
<td>16.33</td>
<td>2%</td>
</tr>
</tbody>
</table>

The result of this can be seen in table 5, and as can be seen the required heap memory has gone up by 5% to 20% depending on the benchmark. It is hard to analyze exactly how reasonable these numbers are, but the fact that the number has gone up is in itself reasonable. The new scanner wrapper performs a large number of buffer operations, and several new attributes have been defined which makes several objects in the compiler larger, requiring more memory.

Compile times are measured using the -profile flag, which makes JastAddJ measure several aspects of the compilation process and outputs those once it has finished compiling. For the compile time tests, five times the JastAddJ 8 minimum required heap memory was used for both versions. All measurements started with a warm up round of compiling the benchmark 15 times without measuring the performance, and then 40 more times while measuring. The average was then computed, which can be seen in table 4. The table shows the time required for different benchmarks split into parsing, error checking, code generation and overall compilation time.

As can be seen, compilation time has gone up slightly. This is to be expected, considering the new scanner wrapper and the fact that some updates in Java 8 have changed parts of the semantic analysis that affects also Java 7 programs. For example, doing overload resolution and analyzing interface methods require more work in Java 8 in general, and should thus require more time on average.

Measurements of the scanner wrapper has been made and can be seen in table 6. The table shows total number of buffer insertions and the max length of the buffer at any time during the compilation. As can be seen, approximately 2 buffer insertions are made per line of code, which should undoubtedly slow
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JastAddJ v7</th>
<th>JastAddJ v8</th>
<th>% increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac</td>
<td>200 MB</td>
<td>210 MB</td>
<td>4.8%</td>
</tr>
<tr>
<td>JastAddJ</td>
<td>320 MB</td>
<td>380 MB</td>
<td>16%</td>
</tr>
<tr>
<td>Clojure</td>
<td>100 MB</td>
<td>130 MB</td>
<td>23%</td>
</tr>
<tr>
<td>hsqldb</td>
<td>310 MB</td>
<td>390 MB</td>
<td>20%</td>
</tr>
</tbody>
</table>

Table 5: Shows the approximate amount of heap memory required for a certain benchmark to compile.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Inserts</th>
<th>Max length</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac</td>
<td>46750</td>
<td>99718</td>
<td>30</td>
</tr>
<tr>
<td>JastAddJ</td>
<td>108000</td>
<td>202105</td>
<td>11</td>
</tr>
<tr>
<td>Clojure</td>
<td>37484</td>
<td>74689</td>
<td>12</td>
</tr>
<tr>
<td>hsqldb</td>
<td>168563</td>
<td>201829</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 6: Shows total buffer inserts and buffer maximum length for some different benchmark programs.

down compilation. It is thus strange that several measurements shown in figure 4 indicate that the parser in the Java 8 module has become faster. There is no reason for this to have happened, and it could be a measurement error since it is very difficult to profile Java programs correctly.

12 Discussion

An implementation of most new features in Java 8 including a parser of the complete language has been presented and evaluated. Approximately half of the implemented features could be added modularly, while the rest required some changes to the already written code to varying degrees. All these changes were implemented separately in the Java 8 module though.

The test suite developed turned out to be extremely helpful during the work. Sometimes, implementation of new features or refactoring old ones resulted in very subtle new bugs introduced in the compiler. Several of these types of regressions were caught by the test suite during the work, which probably saved a lot of time all in all. Even though a lot of time was spent writing test cases, these provide no guarantees that the actual implementation is correct, but the end result is more correct than it would have been if test cases were not developed.

12.1 Problems Encountered

In this section, problems encountered during the work will be presented.

- Locating shift/reduce conflicts in the parser was sometimes very difficult.
  The messages from Beaver provide some subtle hints as to what parsing structures that are relevant, but going from there to actually identifying the source of the conflict was very difficult and time consuming. In total, the parser was probably the part of the work where the least amount of actual results per spent time units was achieved.
• Several bugs in JastAddJ were found during the work, some of which caused test cases in the Java 8 testing suite to fail. Finding these bugs in the earlier implementations could be very difficult since a lot of, sometimes quite complex, JastAdd code had to be navigated. It was also time-consuming just to decide whether these were bugs in the older implementation or something caused by the Java 8 implementation.

• In order to test the test suite, all tests were run using javac, where the idea is that if a test case fails there is probably a problem with the test. In most of the cases this turned out to be true, but in some cases there were tests which javac failed because of actual bugs in javac. Deciding whether or not there was a problem with the tests or whether or not there was a bug in javac was tricky and required reading the JLS8 very carefully. Two bugs were found in javac and reported, while one bug was already found to be reported in the bug database.

12.2 Parser Implementation in javac

Implementing a parser for Java 8 using a LALR(1) parser generator was, as described in section 3, far from a trivial task and required implementing a new layer between the generated parser and scanner. It is thus interesting to look at how other implementations of Java 8 have handled the parser, and in this section the alternative used in javac will be examined.

Javac uses a handwritten parser and scanner, which means they do not have the same restrictions as when using a parser generator. Looking specifically at the problem of implementing parsing of the new intersection casts, javac handles that by performing a token lookahead. When a left parenthesis is found in the beginning of an expression, javac calls a method which analyzes the contents of the parenthesized expression and returns a different value depending on what it contains. This method previews tokens from the scanner until a decision can be made. If a right parenthesis is found and no decision can be made yet, the token after that is used at which point it can definitely decide. The strategy used by javac is thus not so different from the JastAddJ implementation, except that no token buffer is needed which will speed up the process.

For the JastAddJ implementation, which needs to function with the earlier JastAddJ modules, writing a handwritten parser was not really a choice. The scanner wrapper can thus be seen as a way to introduce some handwritten analysis in the middle of the process while still in the end using the same scanner and parser generator as the other modules. This introduces some unavoidable overhead.

12.3 Future Work

In this section, some possible enhancements for the future are examined.

• Implement the rest of the missing Java 8 functionality so that JastAddJ supports a complete version of Java 8.

• Implement stack map tables so that JastAddJ can generate class files of version 52 and above which is required for full Java 8 compatibility.

• Develop test cases covering the entirety of Java 8.
• The implementation of the scanner wrapper is quite naive at the moment, and performs a large amount of buffer operations. This is a potential area for optimizations.

• If the module is extended to fully support Java 8, more in-depth evaluation could be performed, comparing the JastAddJ implementation with other Java 8 compilers.

13 Conclusions

The result of the work described in this thesis is an extension to JastAddJ which implements most major features of the new Java 8 standard. This implementation has been evaluated for completeness, modularity, correctness and performance. The resulting implementation is almost complete and with several features successfully added completely modularly, though not all. An extensive testing suite was developed to test the correctness of the Java 8 module, which provides a solid coverage of the new features in JLS8, and for which JastAddJ v8 completes almost every test. Finally, the performance for JastAddJ v8 has become slightly slower compared to v7, and the memory usage has gone up, which is to be expected.
Appendices

A Test Coverage

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>An interface with only one abstract method is considered functional</td>
<td>Public methods in object are not counted towards the single method</td>
</tr>
<tr>
<td>An interface with no abstract methods is not functional</td>
<td>An interface can have non-abstract methods and still be functional</td>
</tr>
<tr>
<td>Non-public methods in object are counted towards the single method</td>
<td>@FunctionalInterface annotation generates compile-time error only if the annotated interface is not functional</td>
</tr>
<tr>
<td>An interface with two or more abstract methods which are not override-equivalent is not functional</td>
<td>An interface with two or more abstract methods where one is override-equivalent with the rest is functional</td>
</tr>
<tr>
<td>Two abstract methods are considered override-equivalent if the erasure of one of them equals the signature of the other</td>
<td>Several tests that test when the signature of two abstract methods are considered equal</td>
</tr>
<tr>
<td>Several tests that test when two methods are considered return-type-substitutable</td>
<td>Several tests that test two methods are considered to have the same type parameters</td>
</tr>
</tbody>
</table>

Table 7: A table showing coverage of tests related to functional interfaces
A lambda block body which is neither void-compatible nor value-compatible generates error

Super and this behave like they would in the scope outside the lambda

A variable referred to but not declared in the lambda must be effectively final

The only legal modifier for declared lambda parameters is final

Shadowing, redeclaration and general name analysis for lambdas, including both declared and inferred lambda parameters, parameters from outside the lambda and variables declared inside or outside the lambda

Variables declared inside a lambda cannot be referred to outside the lambda

Mixed array notation is not legal for lambda parameters with variable arity

| A lambda block body which is neither void-compatible nor value-compatible generates error | Continue and break must refer to a valid control statement inside the lambda or an error is generated |
| Super and this behave like they would in the scope outside the lambda | A variable must be definitely assigned before the lambda if it is referred to in the body |
| A variable referred to but not declared in the lambda must be effectively final | A lambda cannot throw exceptions not listed in the function type |
| The only legal modifier for declared lambda parameters is final | Inner classes can now refer to outer variables which are effectively final |
| Shadowing, redeclaration and general name analysis for lambdas, including both declared and inferred lambda parameters, parameters from outside the lambda and variables declared inside or outside the lambda | A declared lambda parameter annotated with an annotation interface which includes the meta-annotation Target must include ElementType.PARAMETER |
| Variables declared inside a lambda cannot be referred to outside the lambda | Lambda parameters are not allowed to use _ as identifier |
| Mixed array notation is not legal for lambda parameters with variable arity | Only the final lambda parameter is allowed to be variable arity |

Table 8: Coverage of tests related to lambda semantics
<table>
<thead>
<tr>
<th>Lambda expressions are only legal in cast, assignment and invocation context</th>
<th>Lambda expressions may only target functional interface types</th>
</tr>
</thead>
<tbody>
<tr>
<td>The lambda parameters arity must equal the function type arity</td>
<td>Declared lambda parameters must have the exact same type as the function type parameters</td>
</tr>
<tr>
<td>Inferred lambda parameters infer their type from the targeted function type</td>
<td>Inferred lambda parameters can infer their types from a targeted parameterized functional interface type</td>
</tr>
<tr>
<td>Lambda expression bodies with a statement expression with a type different from void is still compatible with a void return type in a function type</td>
<td>Lambda block body is only compatible with value or void return type if the block is value-compatible and void-compatible respectively, and only if all lambda return types are assignment compatible with the function type return type</td>
</tr>
<tr>
<td>Lambda expression bodies which are not statement expressions must have a type which is assignment compatible to the function type return type</td>
<td>Lambda expression bodies with type void are always only compatible with function type return type void</td>
</tr>
<tr>
<td>Lambda expressions get the correct target type in conditional expressions, return statements and array initializers</td>
<td>Lambdas nested in other lambdas, both expression and block bodies, get the correct target type</td>
</tr>
<tr>
<td>A declared lambda parameter with variable arity type is compatible with the corresponding array type in a function type</td>
<td>When a lambda expression targets a parameterized type, the targeted type effectively becomes the substituted version</td>
</tr>
<tr>
<td>A conditional expression in cast context is not a poly expression, and even if the cast contains a functional interface type lambdas in the true or false expression in the conditional will not get the functional interface type as their target type, and a compile-time error is generated</td>
<td>When targeting a functional interface with several abstract methods, the lambda must be compatible specifically with the method that can override the other methods or an error is generated</td>
</tr>
</tbody>
</table>

Table 9: Coverage of tests related to lambda type analysis
<table>
<thead>
<tr>
<th>Lambda with expression body can return a value when executed at runtime</th>
<th>Lambda with block body can return a value when executed at runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>If a lambda refer to a static variable in its body, and if the value of that variable is updated, the changes are noticeable also when executing the lambda</td>
<td>A lambda can be supplied with arguments at runtime and the argument values are correctly received, both for declared and inferred parameters</td>
</tr>
<tr>
<td>Longer lambda bodies including loops and if-statements are correctly executed</td>
<td>Expression lambda bodies with void-type are run as a statement expression during execution</td>
</tr>
<tr>
<td>A lambda in an array initializer works as expected at runtime</td>
<td>A lambda in a conditional expression works as expected at runtime</td>
</tr>
<tr>
<td>A lambda can be returned from a method</td>
<td>An effectively final variable referred to in a lambda keep the correct value even if the scope of the original variable has disappeared by returning from a method</td>
</tr>
<tr>
<td>A lambda can be returned from another lambda during runtime</td>
<td>The <strong>this</strong> keyword refers to the correct scope at runtime</td>
</tr>
<tr>
<td>When <strong>this</strong> is used inside an anonymous class inside a lambda, the <strong>this</strong> still refers to the anonymous class at runtime</td>
<td>The <strong>super</strong> keyword refers to the correct scope at runtime</td>
</tr>
<tr>
<td>A lambda can be recursive and still executed correctly at runtime</td>
<td>Legal exceptions cast from lambdas can be caught outside them at runtime</td>
</tr>
<tr>
<td>Using <code>instanceof</code> on a lambdas and its target type returns <code>true</code></td>
<td>A lambda can capture and retain the value of an effectively final variable containing a lambda</td>
</tr>
</tbody>
</table>

**Table 10: Coverage of tests related to lambda run time tests**
An expression can be used to refer to a method
Both user defined and library methods can be referred to
The return type of the referenced method must be assignment-compatible with the function type return type
Cannot refer to method with different arity from the function type arity
Method references can be returned from lambdas
The parameter types in the targeted function type must be assignment-compatible with the referenced method parameters
Type arguments for referenced generic methods are inferred if possible, includes type variables both in parameters and return type
Type arguments for referenced generic methods can be supplied manually
Both normal and qualified super can be used to refer to a method
Referenced method is only allowed to have a throws clause which is compatible with the thrown types in the targeted function type
super reference not valid in static context
Generates error if super reference results in an abstract method
Cannot use wildcards in type argument list
When referring to an instance method in a raw type, the type argument is inferred from the function type
A method reference with a primary or super in left hand side can not refer to static methods
A method with a reference type left hand side and arity \( n \) must refer to a static method with arity \( n \) or an instance method with arity \( n - 1 \)
If both a valid static and valid instance method is found for a reference type method reference, an ambiguity error occurs
Cannot refer to a method in a primitive type
A reference type method reference referring to a static method is not allowed to have a parameterized type in the left hand side
Any method return type is compatible with targeted function type void return type

| Table 11: Coverage of tests related to method reference semantics |
Type of the constructor referenced must be assignment-compatible with return type in function type

Type arguments for generic constructors can be inferred automatically or supplied manually

Only thrown types compatible with the function type are allowed in the reference constructor thrown clause

If the class type referenced is not accessible from the constructor reference, an error occurs

A function type targeted by array reference must have a single, int-compatible parameter

Calling a method reference invokes the referenced method in question

Tests that this refer to correct scope when used in the left hand side of a method reference

Tests that overload resolution works for method references when there are several potential methods

Tests that lambdas can return method references at run time

Referring to an instance method in a reference type results in the first parameter qualifying the instance method and invoking it

Calling a constructor reference invokes the referenced constructor

Tests that exception thrown from constructor references can be caught outside

Overload resolution works for constructors when there are several potential constructors

Tests that calling an array reference returns a new array of correct length

<table>
<thead>
<tr>
<th>Table 12: Coverage of tests related to constructor reference semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling a method reference invokes the referenced method in question</td>
</tr>
<tr>
<td>Tests that this refer to correct scope when used in the left hand side of a method reference</td>
</tr>
<tr>
<td>Tests that overload resolution works for method references when there are several potential methods</td>
</tr>
<tr>
<td>Tests that lambdas can return method references at run time</td>
</tr>
<tr>
<td>Referring to an instance method in a reference type results in the first parameter qualifying the instance method and invoking it</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 13: Coverage of tests related to method reference runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling a constructor reference invokes the referenced constructor</td>
</tr>
<tr>
<td>Tests that exception thrown from constructor references can be caught outside</td>
</tr>
<tr>
<td>Methods with default modifier must provide method body</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>Method with abstract modifier must provide a semicolon body</td>
</tr>
<tr>
<td>Default methods may be strictfp</td>
</tr>
<tr>
<td>Static interface methods may not hide method from Object</td>
</tr>
<tr>
<td>Most illegal combinations of interface method modifiers are tested</td>
</tr>
<tr>
<td>Tests that a default or static method body follows the same rules as a normal method body regarding return statements and return types</td>
</tr>
<tr>
<td>Cannot reference static interface methods with a qualifying super</td>
</tr>
<tr>
<td>If a class or interface can inherit several default methods with equal signatures, but one of those override all the else somewhere in the inheritance path, then only the method that overrides the others is actually inherited</td>
</tr>
<tr>
<td>A class only inherits default methods when it does not already define a method itself with the same signature</td>
</tr>
<tr>
<td>An interface may not inherit a default method with same signature as a static method in that interface</td>
</tr>
<tr>
<td>Using a direct superinterface qualifying a super to access an overridden method, the direct superinterface is only legal as a qualifier if there exist no other direct superinterfaces which is a subtype of the first one</td>
</tr>
</tbody>
</table>

Table 15: Coverage of tests related to semantics of new interface methods
<table>
<thead>
<tr>
<th>Applicable and compatible lambdas, method references and constructor references receive correct target type when passed as a method argument</th>
<th>Calling a method inside a lambda, supplying an inferred lambda parameter as argument is legal and overload resolution will be performed in the correct order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda passed as final argument in a variable arity method gets correct target type</td>
<td>If the variable argument is a functional interface type, an arbitrary amount of lambdas can be passed as argument</td>
</tr>
<tr>
<td>A lambda is not potentially applicable with a functional interface type if the arity is different</td>
<td>An explicitly-typed lambda will never be applicable in any invocation context if it is not congruent to the targeted function type with regards to return type and parameter types</td>
</tr>
<tr>
<td>A non-pertinent to applicability lambda will always be considered applicable</td>
<td>When choosing most specific method between two variable arity methods, the last parameter type of the first one must be a subtype of the corresponding parameter in the second one</td>
</tr>
<tr>
<td>A lambda with expression body containing a statement expression is potentially applicable with both void and value return type</td>
<td>Most specific method analysis is not done for lambda arguments which are implicitly-typed</td>
</tr>
<tr>
<td>A method that is applicable in strict invocation context is prioritized over loose and variable arity</td>
<td>A conditional expression is non-pertinent to applicability if either its true or false expression is non-pertinent to applicability</td>
</tr>
<tr>
<td>A method or constructor reference is not potentially applicable unless it can locate at least one potential method that is accessible and has correct arity and static-modifier</td>
<td>A method or constructor reference is never considered applicable if it is not congruent with its targeted type</td>
</tr>
<tr>
<td>If a method or constructor reference is inexact, it will always be considered applicable</td>
<td>When deciding if one functional type is more specific than another functional type for an exact method or constructor reference argument, unless the parameter types in both function types are exactly equal it can not be decided which type that is most specific</td>
</tr>
<tr>
<td>When deciding if one functional type is more specific than another functional type, if both function type’s return types are non-functional reference types or primitive types where one is not a subtype of the other, no choice can be made</td>
<td>If several most specific methods exist, an error occurs</td>
</tr>
</tbody>
</table>

Table 16: Coverage of tests related to semantics of overload resolution
| Tests that the correct method is chosen when several options exist but only one is potentially applicable | Tests that implicitly-typed lambdas will affect the choice of which methods that are potentially applicable |
| Tests that the correct method is chosen for both a strict invocation and loose invocation example | Tests that an explicitly-typed lambda will affect the applicability process, but not an implicitly-typed lambda |
| Individual tests for all five bullets that are considered when an explicitly-typed lambda is the argument to a most specific type analysis | Tests that exact method references will affect the applicability analysis and the correct method is chosen |
| Individual tests for all four bullets that are considered when an exact method reference is the argument to a most specific type analysis | Tests that exact constructor references affects the applicability analysis |
| Individual tests for the first two bullets that are considered when an exact constructor reference is the argument to a most specific type analysis | Tests the new diamond behavior as described in section 9.3.1 |

Table 17: Coverage of tests related to runtime of overload resolution
References

Java 8 language specification, May 2014

Java 7 language specification, May 2014

Java 8 virtual machine specification, May 2014

Java specification request 335, final release, May 2014

Brian Goetz, State of the Lambda, final version, May 2014


