Extending the ExtendJ Java compiler with Java 9 support

Sebastian Hjelm
D13, Lund University, Sweden
dat13shj@student.lu.se

Markus Olsson
D13, Lund University, Sweden
dat13mol@student.lu.se

Abstract

This report covers our extension of the ExtendJ compiler to cover parts of the Java 9 specification. We will discuss the changes we have implemented and how they work in detail. We have also made an evaluation which verifies that our extension has little performance impact on the existing compiler.

1 Introduction

ExtendJ is an extensible Java compiler which enables compiler developers to add new language features as an extension, without changing the existing compiler. The compiler was initially developed with support for only Java 4 [2], and it has been extended to support later versions, including Java 7 [9] and Java 8 [6]. This report focuses on enhancing this compiler with some of the new features that were added in Java 9.

Adding new features to a language inherently requires new code to be written, however depending on how the existing compiler is implemented it can be more or less difficult to integrate. ExtendJ uses *reference attribute grammars* (RAGs) [5] and aspect oriented programming to make it easier to implement new features as an extension.

With RAGs semantic analysis can be implemented declaratively instead of imperatively. Attrbutes can be refined in a module to change behaviour. Therefore, implementing a new feature for ExtendJ usually only involves creating a new module with extensions to the existing code. The only exception to this would be if some part of ExtendJ would need to be refactored to make it easier to add the new module.

There are also other extensible compilers, like Polyglot. Polyglot is another Java compiler, but has some notable differences to ExtendJ. Polyglot does not use RAGs to do semantic analysis and it compiles to Java source code instead of byte code [8].

In 2017 the specification for Java 9 was released, and in order for ExtendJ to remain relevant and useful to research it should be updated to support Java 9. The major new feature in Java 9 is the module system, which allows archives of Java code to have a predefined API that is the only visible code from outside. The specification also contains a few minor changes to Try-With-Resources, the @SafeVarargs annotation, identifier names, etc. [4]. We have implemented a few of the minor changes, but implementing the module system would require too much work for this project.

There is no published work relating to a Java 9 compiler since the specification is so new. However, there are currently two Java 9 implementations; the OpenJDK reference implementation and a beta version of the Eclipse JDT.

This report focuses on the implementation of the minor changes and the performance impact they cause. During the project we found and reported a few issues in ExtendJ, these are discussed in more detail in Section 3.

The rest of this report is organized as follows. In Section 2 we present the small language changes in Java 9 that we have implemented. Section 3 discusses how these changes were implemented and what problems we encountered. Section 4 contains an evaluation of the performance impact of the new module and Section 5 discusses related work. The report is concluded in Section 6.

2 Java 9 changes

Several of the changes that were introduced in the Java 9 specification are minor changes to existing language features. We will briefly cover the changes that we implemented in the following sections.

2.1 Try-With-Resources

Try-With-Resources (TWR) statements where added in Java 7 in order to make resource management simpler. The statement automatically closes its resources when the control flow leaves its body. The exception management is also improved since the TWR handles any exceptions that may be thrown during the initialization and closing of its resources. The following example shows a common resource management pattern in java, before the TWR was added in Java 7:

```
Reader r = null;
try {
  r = new InputStreamReader(/*...*/);
  r.read();
} finally {
  if (r != null) {
      r.close();
  }
}
```

By using a TWR statement the code can be simplified; the following shows equivalent code using Java 7:

Course paper, EDAN70, Lund University, Sweden
December 22, 2017.

```
try (Reader r = new InputStreamReader(/*...*/)) {
  r.read();
}
```

In Java 9 the TWR statements were extended to allow the programmer to declare the resource variable before the try-statement. The condition for this is that the variable must be *effectively final* i.e. it is either final or only assigned once. The resource also has to be assigned before the try-statement ends. An example of this in action is as follows:

```
Reader r = new InputStreamReader(/*...*/);
try (r) {
  r.read();
}
```

Here we create the resource before the try-statement. Also note that the variable is not final, although it is only assigned once.

2.2 SafeVarargs

`@SafeVarargs` is an annotation that was added in Java 7 to allow programmers to silence warning messages from variable arity methods. When you declare a variable arity method with a *non-reifiable* element type (i.e. a type that cannot be determined at runtime) you will get a warning. The following example demonstrates this:

```
<T> T[] toArray(T... objs) {
  return objs;
}
```

This method takes a variable amount of generic objects and puts them in a list. When compiling it you will get the following warning:

```
warning: [unchecked] Possible heap pollution from
    parameterized vararg type T
```

The warning means that the method could possibly cause a class cast exception by incorrectly using its argument. It is used conservatively and will appear even if there is nothing wrong with the code, like in the example above. If the implementation is correct you do not want it to appear when compiling. The annotation will solve this problem.

In Java 9 the applicability of the annotation was extended to allow it to be used on private methods.

2.3 Underscore identifier

Compilers with support for Java 8 should emit a warning when the programmer uses an identifier consisting of only a single underscore. In the Java 9 specification the single underscore has been reserved as a keyword. Now you will get compilation errors for code that uses it as identifiers.

3 Implementation

We implemented the changes that we discussed in Section 2. For each feature we started by writing small compliance tests. We made sure that they compiled in OpenJDK 9 but not in ExtendJ. We then implemented the changes to make the tests compile. The following sections discuss our implementations in detail.

3.1 Try-With-Resources

The extension to the Try-With-Resources statement affects both the syntactic analysis, the semantic analysis and the code generation. This meant that a lot of existing code was involved during the implementation.

Our first idea was to extend the existing parser rules with new productions and adding new nodes to the abstract syntax tree (AST). It turned out that the change was not that simple due to the underlying design of the existing code.

Previously the AST nodes where hard coded to use declarations, since the TWR statements only accepted declarations. By keeping the existing code unchanged the only thing we could do about this was to add a new subclass to the resource declaration and make it act like it was a resource reference. The following code shows how this was implemented in the AST node definition:

```
ResourceDeclaration : VariableDeclarator ::=
    ResourceModifiers ResourceType:Access;
ResourceReference : ResourceDeclaration ::= ;
```

Since our resource reference was a declaration under the hood it still interacted with the rest of the compiler like a declaration which caused issues with type checking, name analysis, code generation, etc. The code got really convoluted and messy and in the end we had to decide to take another route.

Our second and final solution was to simply rewrite most of the existing code. Using the modularity of ExtendJ we excluded some of the original files and reimplemented them in a way that is more general. By doing this we could get rid of the parse rules and AST node definitions that caused problems with our first approach, while keeping most of the attributes and the code generation. We wanted to reuse as much of the existing code as possible.

After replacing the AST nodes the TWR statement now used general statements in its resource list instead of variable declarations which made it possible for us to implement the new functionality. The structure we ended up with is displayed in Figure 1.

Once the AST nodes were replaced and the parse rules were rewritten we had to update some attribute definitions to make it work with both resource declarations and references, we did this by refining several attributes. At the same time we also extended the semantical analysis to check that the resources were valid variables, i.e. that they are effectively final local variables that are subclasses of `AutoCloseable` [4].

Lastly we needed to update the code generation. After some minor changes to adapt to the new AST-structure it worked for the old type of resources with variable declarations. The existing code needed the resources to be placed

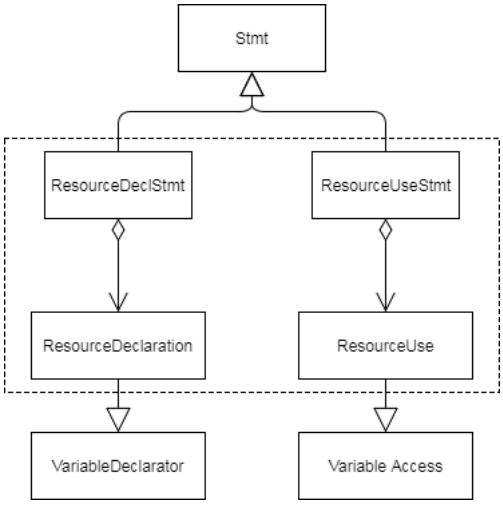

Figure 1. The structure of our AST nodes for Try-With-Resources. The classes within the dashed box are our additions.

inside a specific local variable for it to work. Previously this was not a problem since the variables were declared inside the try-statement and could therefore easily be stored into the variable when generating code for it. Now on the other hand the variable could have been declared much earlier in the code meaning that its code generation is unrelated to the try-statement. We handled this by simply loading its value into the local variable that the try-statement used. The following code implements this:

```
VarAccess access = resource.getExpr();
VariableDeclarator var = access.decl();
var.type().emitLoadLocal(gen, var.localNum());
var.type().emitStoreLocal(gen, resourceIndex);
```

In the above code `resource` is the resource variable (in this case a variable access) and `resourceIndex` is the memory position where the resource should be stored. The code simply retrieves the declaration of the variable and tells it to emit a load instruction followed by a store instruction to copy the variable reference to the resource index, via the stack.

3.2 SafeVarargs

Since the SafeVarargs annotation is only used during compilation to suppress warnings the change to its applicability only affects the semantic analysis of the compiler.

After writing the tests for this feature it was possible to use their compile errors to track down where the code needed to be changed. There was one single JastAdd attribute responsible for validating the usage of the annotation. By creating a new static aspect and refining the attribute we could change its behaviour to fit the new specification, see the code below.

```
refine SafeVarargs
eq MethodDecl.hasIllegalAnnotationSafeVarargs() =
  SafeVarargs.MethodDecl.
      hasIllegalAnnotationSafeVarargs() &&
      (!isVariableArity() || !isPrivate());
```

Here we reuse the previous definition but also allow private variable arity methods.

3.3 Underscore identifier

The change to make the single underscore a reserved keyword only affects the lexical analysis of the compiler since it changes how the input is tokenized.

Our first idea of how this could be implemented was to add a new token to the scanner which consists only of an underscore and giving it higher priority than the identifier token. This is the same way other keywords are implemented. However, this solution turned out to be hard to implement due to the optimizations that are performed when the ExtendJ compiler is generated. Since the underscore keyword has no valid use in the language the compiler generator optimizes it away.

Our second idea was instead to modify the definition of identifier tokens to exclude single underscores and this is what we implemented in the end. Even though this simple change at first glance seems trivial it turned out to be quite tricky to get right.

The first problem was identifying in which source files the identifier token is defined. Over the course of the different Java versions there have been many changes to the lexical analysis in ExtendJ; in particular there have been multiple changes to the definition of identifiers. ExtendJ handles this by including and excluding different files depending on which Java version you are targeting which made it harder to know which files are actually used.

The second problem was to include the new files in a correct way. Macros and rules need to be ordered relative to each other to make the scanner work correctly. If the inclusion order is incorrect you will get obscure errors when you build the compiler.

The final implementation was two small changes. First we created new macros that define valid characters for Java identifiers. We then used these definitions to redefine the identifier token. See the code below:

```
character = ([:jletterdigit:]|[\ud800-\udfff])
start = !(!([:jletter:]|[\ud800-\udfff])|"_")

(("_")({character})+|({start})({character})*)
```

The `character` macro represents all characters that are allowed in identifiers, while the `start` macro represents all characters that are allowed as the first character in an identifier, except for the underscore.

The identifier definition, on the bottom line, forces an identifier to either start with an underscore followed by at

least one other character or lets the identifier start with anything *but* an underscore optionally followed by other characters.

Note that `:jletterdigit:` and `:jletter:` are macros that are predefined in the compiler and correspond to any identifier character and any identifier start character respectively.

We also want to point out that in the real implementation the macros and the identifier definition were put in different files in order for the inclusion order to be correct.

3.4 Issues we encountered

During the implementation and evaluation we encountered several issues with the ExtendJ compiler. All of them have been reported. The following lists gives an overview of them:

- SafeVarargs with private methods was already implemented. When we ran our compliance tests for the changes to SafeVarargs we realized that they incorrectly compiled without error in ExtendJ 8.
- Misleading compiler error messages for SafeVarargs. When the SafeVarargs annotation was used on public non-static, non-final methods the compiler error message was misleading.
- The property that defined the location of the Java libraries has been changed. When ExtendJ is starting it tries to fetch the Java libraries using a built-in system property that was removed with the release of Java 9. This needs to be changed to point to the correct directory
- Compiling some of the open source projects during the evaluation resulted in ExtendJ crashing. The stacktraces hinted that this happened due to infinite recursion and nullpointer exceptions.

4 Evaluation

We wanted to ensure that our implementation works correctly without a significant performance impact. In order to validate our extension we wrote some compliance tests to verify that the extension adheres to the Java 9 specification. We have also evaluated the performance impact by comparing the compilation time with earlier versions of ExtendJ as well as with the OpenJDK compiler. Finally we counted the *source lines of code* (SLOC) of ExtendJ with and without our extension to see if the amount of new code is reasonable.

4.1 Compilation speed

bitWhen comparing the compilation speed we measured only for programs written in Java 8. The reason for this is that we only implemented a subset of the Java 9 features and it does not make sense to compare a compiler with partial support for the specification with a complete implementation such as OpenJDK 9. Instead we mainly compared our compiler with ExtendJ 8 which makes it possible to determine if the changes caused any overhead, but we also included a comparison with OpenJDK 8.

The tests were performed by compiling different open source projects (see Table 1) 30 times each and then computing 95% confidence intervals for their execution times. We did this to be sure whether or not there was any statistically significant overhead introduced by our extension. For each of the test runs we measured the total running time from startup of the compiler JVM to when the instance terminated as well as the steady-state running time. The steady-state running time is relevant to make sure that the performance evaluation is statistically rigorous [3].

biThe results are summarized in Figure 2. As the figure shows there is no statistically significant difference between the compilation speed of ExtendJ 8 and our updated version (except for a single project). However, there is a noticable difference in performance compared to OpenJDK; we saw similar differences in compilation time between ExtendJ and OpenJDK as was presented in previous work [9].

When compiling Cayenne and Antlr ExtendJ crashed which caused the compilation to halt, this is probably the reason the compilation time decreased for those data points.

Project	Revision	SLOC
SLogic[1]	6ac825c	14787
Antlr	f3b5ce1	34359
Checkstyle	49d74e7	51278
Corbertura	3b0cd52	52478
Ant	463d198	104270
Castor	442661e	166322
Argouml	19847	195363
Cayenne	bf37208	198477

Table 1. The open source projects we used when measuring the compilation time.

4.2 Java 9 compliance

In order to verify that our changes are compliant with the Java 9 specification we wrote 22 test cases. The tests are designed to use the new features in every possible way as well as testing invalid use cases. An example of such a test is as follows:

```
// Test illegal identifier name
// .result=COMPILE_FAIL
public class Test {
  private void m() {
    String _ = "_";
  }
}
```

This test verifies that it is invalid to use underscores as identifier names; compiling it results in compilation errors.

[1] https://github.com/Sebastian-0/SLogic

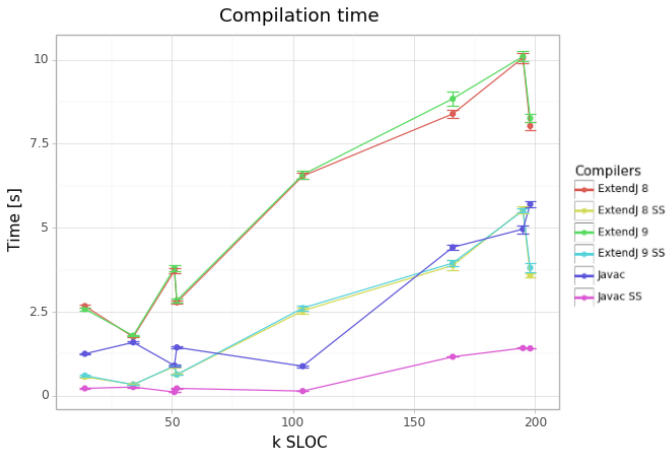

Figure 2. Average compilation time and confidence intervals when compiling the open source projects in Table 1.

We used ExtendJ's regression tests repository[2] as a framework to write our compliance tests. They test many of the features from Java 8 and below by verifying both that correct code compiles and runs, and that faulty code get errors. We made sure that our compiler passes all the existing regression tests. This process was almost seamless, however some old tests needed to be changed since they used a single underscore as an identifier.

4.3 Source lines of code

We analyzed the implementation size by measuring the SLOC of our compiler extension and the base version of ExtendJ. The measurements were made with CLOC [1] using custom filters to include the scanner, parser and JastAdd files. The results of the measurements are presented in Table 2. Clearly the amount of new code is relatively small compared to the total size of the compiler and corresponds roughly to an increase in size of 0.80% which is reasonably small considering the changes we have made. The SLOC used in the evaluation was also made with CLOC.

Compiler	SLOC
ExtendJ Java 8	38453
ExtendJ Java 9	38761

Table 2. The SLOC of ExtendJ with or without our Java 9 extension.

5 Related work

Currently the only other Java 9 compilers that we know of are OpenJDK and Eclipse JDT, which is still in beta.

[2]https://bitbucket.org/extendj/regression-tests

Apart from ExtendJ there are several other compilers that are implemented to be extensible, the following are some examples.

- Polyglot - A compiler that compiles to Java code. The idea is to compile an extended version of Java to ordinary Java code that can later be compiled to byte code. This compiler is designed to be extensible and make it easy to add new language constructs and create domain specific languages [8].
- JaCo - An extensible Java compiler that is implemented using a combination of concepts from functional and object-oriented programming [10].
- Cetus - A source-to-source compiler designed to parallelize code. The base version of the compiler only works with C code but it can be extended to work with other languages [7]. Unlike ExtendJ this compiler does not enable the programmer to add new language constructs to a language, instead it has a framework that makes it parallelize code for many languages.

While all of these compilers features extensibility none of them uses RAGs like ExtendJ. This goes to show that there have been many different ideas on how to implement modularity over the years.

6 Conclusion

We have implemented partial support for Java 9 in the ExtendJ compiler and by doing this we have demonstrated that the modularity of using RAGs works in practice. However there have been some problems along the way.

While using RAGs in theory allows a compiler to be fully modularized it also demands good code design and smart code structure to work in practice. If the structure is not good enough you may be forced to make more drastic changes. When we first tried to implement the extension of the Try-With-Resources statement we ran into this problem; in the end we had to replace a lot of code rather than extending it to make the extension work.

In contrast it was much easier to implement the changes to the SafeVarargs-annotation: we only had to refine a single attribute to make it work. Granted that the change was a minor one this still shows how useful the modularized framework can be.

Another drawback of using the modular design is that every time a new Java version is released, another layer of source code is added in the compiler. Every time this happens it gets harder and harder for new developers to orient themselves in the code. We had some problems with this when we changed the definition of identifier tokens, both with finding the relevant files and including the new ones in the correct order.

We only implemented a subset of the new features that were added in the Java 9 specification so there is much room for future work. There are both some minor changes left

to implement including changes to the diamond operator, private methods in interfaces, etc. and the major addition of the new module system. Future projects may also make a full performance comparison between OpenJDK 9 and ExtendJ to get a good idea of their relative performance. The last such comparison was made when ExtendJ was extended to Java 7.

Acknowledgments

We want to thank our supervisor for all his valuable feedback and help during the project.

References

[1] 2017. CLOC. http://cloc.sourceforge.net. (2017). Accessed: 2017-12-05.

[2] T. Ekman and G. Hedin. 2007. The Jastadd Extensible Java Compiler. *SIGPLAN Not.* 42, 10 (Oct. 2007), 1–18. https://doi.org/10.1145/1297105.1297029

[3] A. Georges, D. Buytaert, and L. Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of OOPSLA 2007*. ACM, Montreal, Canada, 57–76. http://doi.acm.org/10.1145/1297027.1297033

[4] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith. 2017. The Java® Language Specification Java SE 9 Edition. (Aug. 2017). https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf

[5] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000).

[6] E. Hogeman. 2014. Extending JastAddJ to Java 8. (2014). Student Paper.

[7] S. Lee, T. Johnson, and R. Eigenmann. 2004. *Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 539–553. https://doi.org/10.1007/978-3-540-24644-2_35

[8] N. Nystrom, M. Clarkson, and A. Myers. 2003. Polyglot: An extensible compiler framework for Java. In *Compiler Construction*. Springer, 138–152.

[9] J. Öqvist and G. Hedin. 2013. Extending the JastAdd extensible Java compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*. 147–152. https://doi.org/10.1145/2500828.2500843

[10] M. Zenger and M. Odersky. 2001. Implementing Extensible Compilers. In *Proceedings of ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages*. https://infoscience.epfl.ch/record/64402