Extending ExtendJ with Pattern Matching for instanceof

Klara Lööf

E21, Lund University, Sweden

kl8843lo-s@student.lu.se

Cornelia Norén Vosveld

E21, Lund University, Sweden

co0808no-s@student.lu.se

Abstract

This paper explores adding new language constructs to ExtendJ, an extensible Java compiler based on the JastAdd framework. ExtendJ is a modular compiler for testing new language features, but its current support only extends to Java 11. With the goal to make ExtendJ more compatible with newer Java versions, this paper focuses on the implementation of instanceof with pattern matching, a feature that was first previewed in Java 14. This feature simplifies code by combining type checking and casting into one step, removing the dependency between the two and preventing ClassCastExceptions.

Our implementation is backwards compatible, as all changes have been treated as add-ons to ExtendJ to still allow the continued use of older Java versions. The implementation uses a rewrite-based approach in order to utilize the existing properties of the standard instanceof construct, introduced in Java 1, and thus avoiding repetitive code. Support was implemented for `if-` and `while`-statements, including more complex conditions using the AND operator.

We evaluated using an extensive test-suite alongside several new targeted test cases. The results show that the extension successfully passes all tests without changing how the compiler treats older Java versions, thereby increasing ExtendJ's compliance with newer Java standards.

1 Introduction

The extensible Java compiler ExtendJ is a compiler based on the JastAdd language and implemented using Reference attribute grammars, RAGs [5]. In ExtendJ, formerly JastAddJ, changes can be added as extensions using already existing modules without modifying the existing compiler [11]. The way ExtendJ is built makes it useful for developing static analysis and prototyping new language constructs to the Java language, in contrast to other Java compilers where extensibility is not the main goal. ExtendJ was created with modularity and extensibility in mind.

In the JastAdd language the order in which attributes are stated does not matter [11]. This means additions and extensions with new attribute grammar can be added in new files as additions to the already existing compiler logic. When adding static analysis one can then use all of the already implemented useful attributes in your analysis. The modularity and opportunity to easily add new aspects to the compiler also makes ExtendJ useful for prototyping new Java language constructs. In order for this functionality of ExtendJ to be complete, ExtendJ needs to be compliant with the Java language. When prototyping a new language construct it is important that the compiler is up to date on Java.

As the Java language evolves and new versions are released compilers must be updated. In order to handle progressions of the language constructs large changes to the source code of the compiler can be needed, however since ExtendJ built to be extensible this makes it easier. ExtendJ currently supports Java versions up to Java 11 [8]. In order for ExtendJ to support more of the Java language, we intend to extend the compiler with language constructs that has been released since Java 11.

2 Background

2.1 Instance of with pattern matching

In Java, we often work with general objects that could be one of several different types. For example, you might have a variable labeled as a generic "Animal," but at runtime, it could actually be a "Dog" or a "Cat." Before we can use more specific features, we have to verify exactly what the object is. If we don't check this and accidentally try to treat a Dog like a Cat, the program will crash. To prevent this, Java uses the instanceof operator to check the type before performing any actions. A common use of the the instanceof operator is to accompany it with a cast expression casting that object to the tested type. The problem is that the cast is dependent on the instanceof check, and if the type in the instanceof expression changes, the cast must also be updated to prevent errors. A situation where a ClassCastException occurs is depicted in the example below.

Course paper, EDAN70, Lund University, Sweden
January 26, 2026.

```
if (obj instanceof Cat) {
    Cat c = (Cat)obj;
    System.out.println(c.age());
}

if (obj instanceof Dog) {
    Cat c = (Cat)obj;
    System.out.println(c.age());
}
```

In this example, we change the type used with the instanceof operator, but forget to modify the cast expression. Apart from the construct being error-prone, it is also redundant, and to prevent this, pattern matching for instanceOf was introduced as a preview in Java 14 and released in Java 16 [6]. The following example of an instanceof expression using earlier Java version shows the need to cast the object in order to use it.

```
if (obj instanceof Cat) {
    Cat c = (Cat)obj;
    System.out.println(c.age());
}
```

Using pattern matching instead the code can be reduced to:

```
if (obj instanceof Cat c) {
    System.out.println(c.age());
}
```

When using instanceof with pattern matching, the object is directly cast to the type and assigned to a variable if it matches the pattern. This makes the use of instanceof easier and the code more readable. In order to use this new language construct with ExtendJ we have to extend the compiler.

2.2 The compiler pipeline

When a compiler processes source code, it is passed through a structured pipeline that translates the source code into byte code. The ExtendJ compiler utilizes the JastAdd framework. The architecture is centred around the evolution of an Abstract Syntax Tree(AST). The scanner breaks the raw code into small tokens that are then passed to the parser, which organizes them into an AST. The AST represents the logical hierarchy of the program and consists of nodes representing different constructs, such as loops or variable declarations[2]. These nodes can have children to represent related data, for example, a 'Plus' node would have two children representing the numbers being added. All possible structures of the tree and the relationships between nodes are formally defined by a grammar file. This file acts as a blueprint, ensuring the compiler only builds trees that follow the formal rules of the language.

Once the AST is built, the compiler performs attribute evaluation, or semantic analysis to analyse the codes meaning. Rather than using a manual search to find information, you define declarative properties called attributes directly on the AST nodes. Attributes enrich the tree with vital information, such as Name- or Type Analysis. Also, semantic errors (like type mismatches) are identified by checking these computed attributes against the language rules.

Once the AST is fully attributed and verified to be error-free, the compiler moves on to the final stage, generating the byte code.[11].

2.3 Rewrites

In JastAdd, rewriting is a mechanism that is used for replacing AST nodes with a rewritten version of the node. They can either apply to all occurances where the specified node has been accessed, or be used with conditions to further tailor the rewrite [3]. Figure 1 and 2 is an example of how a rewrite rule can be used to replace a until-loop with an "inverted" while-loop.

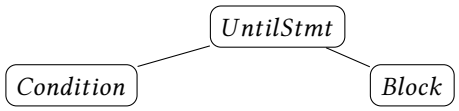

Figure 1. Before rewrite

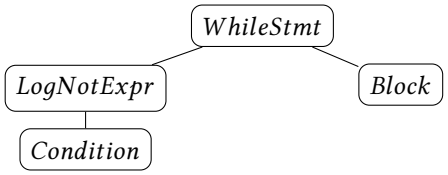

Figure 2. After rewrite

In this case, all until-loops are replaced with while-loops, where the condition has been inverted using a binary not operator. This way, all properties of the while-loop can be utilized without the need to implement the same properties for the until-loop.

3 Implementation

As mentioned earlier, the goal of this implementation was to add support for instanceof with pattern matching. Certain limitations were identified during the course of the project. In the end, the additions made enables the use of instanceof with pattern matching as the sole condition or as operands in

conditions using AND-operands, in if-statements as well as while-statements. To implement this new language construct, several additions had to be made to the latest release of ExtendJ.

Support for instanceof existed in the first version of ExtendJ, that supports Java 4, and the grammar for it can be seen in the example below.

```
InstanceOfExpr : Expr ::=
    Expr TypeAccess:Access;
```

Currently, ExtendJ works for Java version up to Java 11. One benefit of how ExtendJ is structured is that one can jump between versions, which is done by treating all new version supports as add-ons. For this reason, any implementation done for Java 14 support should be separate from the existing code. This essentially means that modifying the existing instanceof-structure, implemented as Java 4 support, wasn't an option since it would give versions before Java 14 properties they shouldn't have. For this reason, a new node type **InstanceOfPMExpr** with the same structure as the existing **InstanceOfExpr** but with an added pattern variable was added to the AST. Pattern variables are also introduced as a new node, **PatternVar**.

```
InstanceOfPMExpr : InstanceOfExpr ::=
    Expr TypeAccess:Access PatternVar;
PatternVar ::= <ID:String>;
```

The new node type **InstanceOfPMExpr** should be able to appear in all places the **InstanceOfExpr** can. To ensure this, we chose to have **InstanceOfPMExpr** inherit from **InstanceOfExpr**. This means that any parent node expecting the common instanceof node, also will accept instanceof with pattern matching.

3.1 Core Implementation

In order to utilize as much of the existing constructs as possible, we wanted to rewrite instanceof with pattern matching to look like an ordinary instanceof use. The easiest approach for this was using a rewrite rule. The desired structure was the one of a standard instanceof expression as depicted in section 2.1, where the variable is initiated in the block and cast to the type. As mentioned in 2.3, this means we could reuse attributes defined for instanceof, along with necessary analyses, such as type checking and name analysis.

Figure 3 and 4 shows how the AST is changed after applying the rewrite rule. After the rewrite, the if-statement will work the same way as that of an if-statement using the standard instanceof.

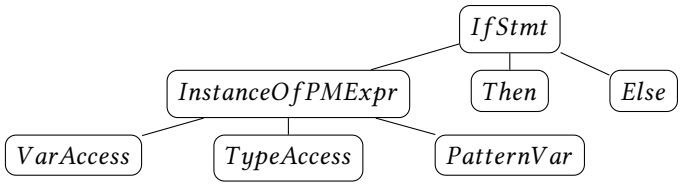

Figure 3. Before rewrite

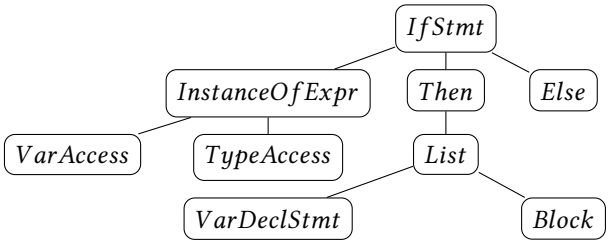

Figure 4. After rewrite

To implement this approach, some "helper"-methods were needed. These were implemented as subtrees, inserted in the AST. We generated several new subtrees for different purposes like finding uses of pattern matching as well as removing all pattern variables from the conditions. With this approach, the instanceof with pattern matching will work in the exact same way as without.

All the subtrees created regarding the **InstanceOfPMExpr** node can be used for both If- and While-statements. However, the rewrite rule and the subtrees restructuring the statement will differ depending on the statement. For example, when having an If-statement, we have to take potential Else-statements into consideration when creating the new statement. Since rewrite rules are applied to specific nodes, different ones had to be implemented for both statements.

3.2 Identifying InstanceOfPMExpr occurrences

To make sure the rewrite only applies to If- and While-statements containing the node **InstanceOfPMExpr**, we needed to add a condition to the rewrite rule. This was done by creating a boolean attribute **hasPatternInstanceOf()** that checks the condition of the statement, and returns a true value if an occurrence of **InstanceOfPMExpr** is found. On all node types except **InstanceOfPMExpr** and **AndLogicalExpr**, the value of the boolean attribute will be false. For the **InstanceOfPMExpr** node type, the attribute will always be true and in case of **AndLogicalExpr**, we will recursively look trough the left and right nodes and return true if we find an **InstanceOfPMExpr** occurrence, otherwise false. The **hasPatternInstanceOf()** attribute is used as the condition in the rewrite rule for `if-` as well as `while-`statements,

meaning that the rewrite rule only applies when the boolean attribute is true.

3.3 Reconstructing the Condition

To get the desired structure of the condition, that of a standard instanceof use, we needed to replace all **InstanceOfPMExpr** nodes with **InstanceOfExpr**. This was done with an attribute **withoutPattern()** that similar to **hasPatternInstanceOf()** exists on the **Expr** node and is overwritten at the **InstanceOfPMExpr** node. It returns a new **InstanceOfExpr** that uses the same **Expr** and **TypeAccess**. Since our add-on handles the use of logical AND-expressions, the **withoutPattern()**-attribute is also overwritten at the **AndLogicalExpr** node, and was implemented by recursively extruding unpatterned instanceof expressions from left and right operands.

3.4 Creating the new Statements

Apart from restructuring the condition of the if- or while statement, variable declarations with cast expressions had to be added in order for it to work like the standard use of instanceof where the object is cast to a new variable in the body of the statement. Since the construct allows the use of one or multiple instanceof operators when using AND-operators, another attribute **getInstanceOf()** was needed in order to extract all instanceof nodes. Variable declarations with cast expressions are created for each instanceof node and added to the beginning of the statement body, in order for it to be visible for the rest of the block. A new if- or while statement node is created using all these changes and additions, and replaces the original node.

4 Evaluation

For ExtendJ, there is an existing test suite consisting of around 1800 tests, testing different properties up until Java 11. Using this test suite we created several new tests to evaluate the new language construct. As the 1800 test that were created prior to our extension ran each time we tested, we ensured that our changes did not impact ExtendJs behaviour in other, unexpected, ways. The tests that we have created test the instanceof with pattern matching construct in extensive ways. Using assertion tests to compare the output of a file with the expected output we ensure correct behaviour. In the basic tests we ensure that the new construct can be parsed and that instance of condition returns the right boolean value. Additionally the pattern matching variable is tested to make sure it is declared correctly and visible in the correct scope of the code. The language construct is tested in both if- and while-statements. The tests explore the AND operator, both with multiple instanceof expressions and with other logical expressions. In the tests we have used both Java Objects such as **List, ArrayList, LinkedList** to analyse the instanceof operation as well as instances of classes created for this project, such as the **Cat** and **Dog** examples from the Background section.

Comparing the latest version of ExtendJ that was available before our extension, for Java 11, to our extension, the Java 11 jar-file passes 1830 tests and our jar-file passes all 1846 tests. This shows that our extending of ExtendJ increases the compilers support for the Java language.

In order to further prove the differences in support for the Java language between the versions we would have liked to test the different versions using a complete Java language test suite to show that after our implementation we would pass more tests than before. Unfortunately there are no available inexpensive complete test suites for the Java language, so due to economic constraints that was not an option for this project.

5 Related work

ExtendJ has previously been extended in different stages. ExtendJ was extended with support for Java 7 by Öqvist and Hedin in 2013 [7]. The paper focuses on the implementation of two constructs, *Try-With-Resources* and the *Diamond* operator. In 2023 a Master's Thesis by Aronsson and Björk implemented support for Java 9, 10 and 11 [8]. Both of these papers include a much larger implementation that involves many more language constructs than our paper. However the same methodology and structure is used. In neither of these papers **rewrite** is used as primary solution.

There is an extensible compiler framework for Java called Polyglot [9]. Polyglot, developed at Cornell University, differs from ExtendJ in that it is not based on reference attribute grammar however the focus on extendability is shared. Polyglot has extensions that supports Java 7 with some key features of Java 8 implemented [1].

Silver is an attribute grammar specification language developed at University if Minnesota [4], like JastAdd one of the main goals for implementing the language was extendability. AbleC is a compiler for the C language based on Silver that allows for creating extensions for new language constructs without modifying the source code of the compiler [10].

6 Conclusion

The implementation of instance of with pattern matching into ExtendJ works for if- and while-statements, as the sole condition or with multiple other logical expressions using the AND operator. This increases the compilers support for the current Java language. For future work the next step in the implementation would be to enable the use of the instance of with pattern matching with more logical operators such as OR and NOT operands. Using the already established foundation of this project, we believe this would be easy to

implement. Because of the way ExtendJ was implemented, the new pattern variable we implemented for this project can easily be reused, perhaps for implementing pattern matching for switch statements which was previewed in Java 17.

References

[1] Accessed Jan 2026. Polyglot. A compiler front end framework for building Java language extensions. (Accessed Jan 2026). https://www.cs.cornell.edu/projects/polyglot/

[2] Lund University Department of Computer Science. 2025. JastAdd Concept Overview. (2025). https://jastadd.cs.lth.se/web/ Accessed 5 January 2026.

[3] Lund University Department of Computer Science. 2025. JastAdd Reference Manual. (2025). https://jastadd.cs.lth.se/web/ Accessed: 27 November 2025.

[4] Jimin Gao Lijesh Krishnan Eric Van Wyk, Derek Bodin. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75 (2010), 39–54.

[5] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000), 285–428.

[6] Guy Steele Gilad Bracha Alex Buckley Daniel Smith Gavin Bierman James Gosling, Bill Joy. 2021. The Java® Language Specification Java SE 16 Edition. (2021), 655–657.

[7] Görel Hedin Jesper Öqvist. 2013. Extending the JastAdd Extensible Java Compiler to Java 7. *PPPJ '13: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (2013), 147 – 152. https://doi.org/10.1145/2500828.2500843

[8] David Björk Johannes Aronsson. 2023. *Extending the ExtendJ Java Compiler*. Master's thesis. Lund University, LTH.

[9] Michael R. Clarkson Nathaniel Nystrom and Andrew C. Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. *Compiler Construction* (2003), 138–152.

[10] Travis Carlson Eric Van Wyk Ted Kaminski, Lucas Kramer. 2017. Reliable and automatic composition of language extensions to C: the ableC extensible language framework. *Proceedings of the ACM on Programming Languages Volume 1, Issue OOPSLA* (2017), 1–29. https://doi.org/10.1145/3138224

[11] Görel Hedin Torbjörn Ekman. 2007. The jastadd extensible java compiler. *ACM SIGPLAN Notices* 42, 10 (2007), 1–18. https://doi.org/10.1145/1297105.1297029