Extending LINVAST* with new programming language - Java

*LINVAST - Language-INVariant AST library

Dara Milojković, ...
Marija Katić, katic.marija.97@gmail.com

Software Verification course
University of Belgrade
Faculty of Mathematics

September 25, 2021

Contents

1 Problem description

LINVAST is a set of libraries which provides common AST API for different languages (C and Lua at the moment of writing). LINVAST can theoretically work with any programming language as long as the adapter for that language is written (hence, Language Invariant). The purpose of this project is to extend LINVAST with Java programming language.

Adapters (or, in further text, Builders) serve as an intermediary between ANTLR parse trees and language-invariant ASTs. Builders are used to generate AST from a parse tree and they are implemented differently for every programming language due to native differences in parse trees. Therefore, the main task in the project is to implement (and test) Builder for Java programming language.

2 Implementation details

A Builder for Java was created by creating a type `JavaASTBuilder` extending `ANTLR_GENERATED_BASE_VISITOR<ASTNode>` and implementing

`IASTBuilder<ANTLR_GENERATED_PARSER_TYPE>`. In Figure 1 we can see the relation between the existing system and the added component.

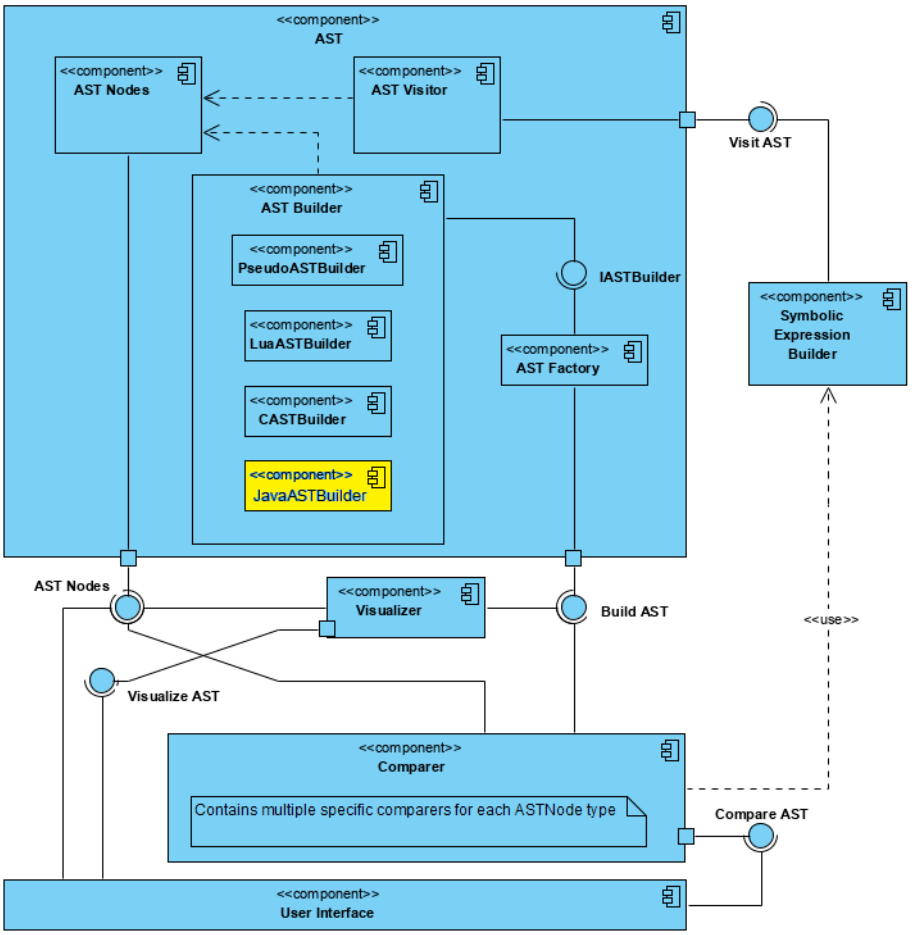

Figure 1: UML diagram of LINVAST implementation with new component added (Java builder)

Builder should extend `ANTLR_GENERATED_BASE_VISITOR<ASTNode>` in order to implement methods for recursive visit of ANTLR parse tree (a visitor method for every ANTLR tree node type). These methods return language invariant AST nodes, and when whole parse tree is visited, the whole AST is created and returned.

In Figure 2 there is an example of difficulty we faced. There is a discrepancy between Java parse tree and the AST. First two images of the figure show that Java uses two node types to represent variable declaration: modifier (e.g. public, private, static, final, etc.) and fieldDeclaration (e.g. <type> <var> = <expression>, int x = 3). On the other hand, the AST separates variable declaration bit differently: Declaration Specifier Node type stores both modifiers and type name, and declarator stores the rest (e.g. x = 3). Hence there is no trivial mapping between Java parse tree node types and AST node types.

```
classBodyDeclaration
    : ';'
    | STATIC? block
    | modifier* memberDeclaration
    ;

memberDeclaration
    : methodDeclaration
    | genericMethodDeclaration
    | fieldDeclaration
    | constructorDeclaration
    | genericConstructorDeclaration
    | interfaceDeclaration
    | annotationTypeDeclaration
    | classDeclaration
    | enumDeclaration
    ;
```

```
fieldDeclaration
    :  typeType variableDeclarators ';'
    ;

variableDeclarators
    : variableDeclarator (',' variableDeclarator)*
    ;

variableDeclarator
    : variableDeclaratorId ('=' variableInitializer)?
    ;
```

```
extern static const int x = 3, arr[] = {1, 2, 3};

public static final int x = 3;
public static final int[] arr = new int[] {1, 2, 3};

public static readonly int x = 3;
public static readonly int[] arr = new[] {1, 2, 3};
```

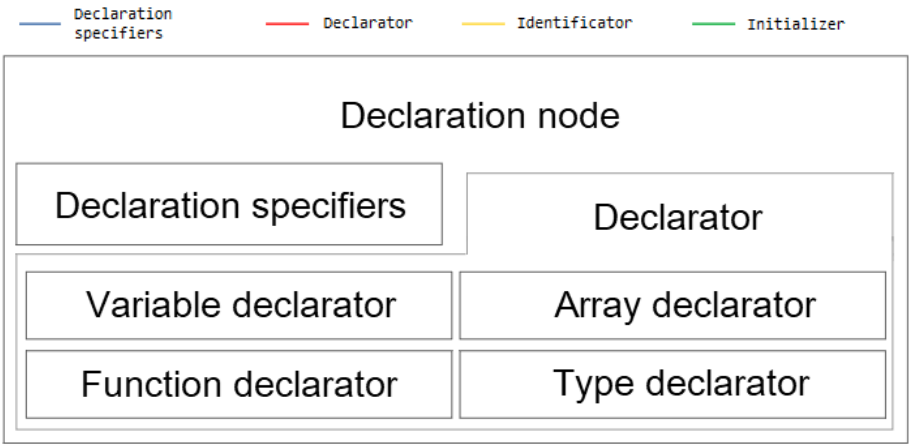

Figure 2: Example of discrepancy between ANTLR parse tree and the language invariant AST

3 Examples of Language-invariant AST generated from Java source

Since not all visitor methods have been implemented yet, the examples are small, e.g. for now we can only show examples with empty block statements.

```
1 Java code will be added here
```

Image of AST will be added here.

Figure 3: Example 1

```
1 Java code will be added here
```

Image of AST will be added here.

Figure 4: Example 2