Design and Implementation of the JAVA-- Compiler
Part 1 - Frontend

Compilers - L.EIC026 - 2024/2025

Dr. João Bispo, Dr. Tiago Carvalho, Lázaro Costa,
Pedro Pinto, Susana Lima and Alexandre Abreu

University of Porto/FEUP
Department of Informatics Engineering

version 1.1, March 2025

Contents

Objectives

This programming project aims at exposing the students to the various aspects of programming language design and implementation by building a working compiler for a simple, but realistic high-level programming language. In this process, the students are expected to apply the knowledge acquired during the lectures and understand the various underlying algorithms and implementation trade-offs. The envisioned compiler will be able to handle a language based on the popular Java

programming language and generate valid Java Virtual Machine (JVM) instructions in the *jasmin* format, which are then translated into Java *bytecodes* by the jasmin assembler.

You will be given a fully working compiler, from parsing to jasmin generation, but for a very small portion of what you have to implement. This should be used as the basis for your project. This means that it is very important that you read and understand the code of this initial implementation! It is also expected that you write your own tests for the features that will be asked, this is also a very important part of compiler development.

1 The JAVA-- Language

The JAVA-- language is for the most part, a subset of the Java programming language, with some inclusions to make it a more interesting challenge, but also making JAVA-- code invalid Java code when those extra parts are used. Additionally, Java classes that have been compiled into bytecodes can be imported and used in JAVA-- code.

In this assignment we provide a grammar for JAVA--, but keep in mind that the grammar does not capture the correct JAVA-- language. Instead, it is deliberately lax, as some syntactic constructs are allowed but do not have a correct semantic meaning. For instance, the production Expression: Expression '[' Expression ']' allows for the (1+2)[0] input to be accepted, which is neither valid JAVA-- nor valid Java.

This aspect of the grammar definition illustrates an important point. Often, we can have a more relaxed grammar definition that will lead to a simpler parser implementation, at the cost of a more complex semantic analysis phase that will need to check more rules regarding what is and is not allowed in the language. The alternative is to have stricter grammar which would then reduce the burden of semantic checking. Our suggestion is to follow the former approach and make your semantic phase more elaborate.

1.1 JAVA-- Grammar

Figure 1 depicts the grammar of JAVA-- in EBNF (*Extended Backus–Naur Form*)[1]. The tokens are delimited by single quote signs ('...') and all the non-terminals have their corresponding production(s) specified in the grammar. The non-terminal "program" represents the starting rule of the grammar, and the elements of the grammar ***ID*** and ***INT*** are terminal symbols that follow the lexical rules of the Java programming language.

An ***ID*** is a sequence of letters and digits, the first of which must be a letter. The letters include uppercase and lowercase characters, the ASCII underscore (_) and the dollar sign ($). The digits include the digits 0-9. An ***INT*** is either the single digit 0, representing the integer zero, or consists of a digit from 1 to 9 optionally followed by one or more digits from 0 to 9.

The comments in JAVA-- also follow the Java rules regarding comments. There are two kinds of comments:

- `/* text */`, a multi-line comment where all the text from the character "/*" to the character "*/" is ignored;
- `// text`, an end-of-line comment: all the text from the characters "//" to the end of the line is ignored.

All the remaining symbols pertain to the EBNF rules, used to specify the grammar. A rule followed by a "?" is an optional rule; a rule followed by a "+" can be used one or more times; and a rule followed by a "*" can be used zero or more times.

[1] https://en.wikipedia.org/wiki/Extended_Backus-Naur_form

```
program           ::= (importDeclaration)* classDeclaration EOF
importDeclaration ::= 'import' ID ( '.' ID )* ';'
classDeclaration  ::= 'class' ID ( 'extends' ID )? '{' ( varDeclaration )* ( methodDeclaration )*
                      '}'
varDeclaration    ::= type ID ';'
methodDeclaration ::= ('public')? type ID '(' ( type ID ( ',' type ID )* )? ')' '{' ( varDeclaration
                      )* ( statement )* 'return' expression ';' '}'
                    | ('public')? 'static' 'void' 'main' '(' 'String' '[' ']' ID ')' '{' ( varDeclaration
                      )* ( statement )* '}'
type              ::= 'int' '[' ']'
                    | 'int' '...'
                    | 'boolean'
                    | 'int'
                    | ID
statement         ::= '{' ( statement )* '}'
                    | 'if' '(' expression ')' statement 'else' statement
                    | 'while' '(' expression ')' statement
                    | expression ';'
                    | ID '=' expression ';'
                    | ID '[' expression ']' '=' expression ';'
expression        ::= expression ('&&' | '<' | '+' | '-' | '*' | '/' ) expression
                    | expression '[' expression ']'
                    | expression '.' 'length'
                    | expression '.' ID '(' ( expression ( ',' expression )* )? ')'
                    | 'new' 'int' '[' expression ']'
                    | 'new' ID '(' ')'
                    | '!' expression
                    | '(' expression ')'
                    | '[' ( expression ( ',' expression )* )? ']'
                    | INT
                    | 'true'
                    | 'false'
                    | ID
                    | 'this'
```

Figure 1: EBNF Java-- Grammar.

1.2 The Import Declaration

The compiler will support the use of external classes via the import statement. To simplify the development of the compiler, you are not required to validate if the imported classes exist in the current classpath and simply assume that the classes do exist.

However, this imposes a restriction for the analysis of complex expressions. This restriction is due to information regarding types not being available for imported methods. For instance, if we import class `M` and use it in an expression such as `M.foo().bar()`, you will understand later that we can only invoke method "bar" if we know the return type of "foo". Since we do not know anything about class `M`, we do not have access to the return type of "foo". Therefore, and again to ease the development of the compiler, the use of imported classes can only be done in direct assignments (e.g., `a = M.foo();`, `a=m1.g();`) or as simple call statements, i.e., without assignment (e.g., `M.foo();` , `m1.g();`). With this simplification, you are expected to assume the return types of the methods of imported classes according to how they are used.

Consider the code in Figure 2. We can assume that "foo" is a static method, since it is being called directly from the class, and that returns an instance of "M", since "a" is of type `M`. We can also assume that the "bar" method is an instance method, since it is invoked from the object "a", that the return type is void, since there is no assignment, and that the method expects as arguments a single integer.

```
import M;

class Test {

    int foo(int param) {
        M a;

        a = M.foo();
        a.bar(param);

        return 0;
    }

}
```

Figure 2: Import example in Java--.

This simplification is only for methods of classes that are imported. For methods that are declared inside the current class you will have complete information about the method signature, which allows you to validate (and support) calls that appear in more complex, compound operations (e.g, `a = b * this.m(10,20)`, where "m" is a method declared inside the class).

1.3 Introducing Varargs

Variable arguments, also known as varargs or as variadic functions[2] are a concept that already exists in the Java language, and is represented by adding an ellipsis (i.e., `...`) after the type of a parameter in a method declaration (e.g., `void foo(int... a)`). During method calls, this annotation allows passing a variable number of arguments (all of the same type) to the method (e.g., `foo(10, 20, 30)`).

Varargs in Java can only be used in the last parameter of a method declaration, and in practice, it is equivalent to an array of the same type as the varargs (e.g., `int...` internally translates to `int[]`). Java-- will limit varargs to int, and will follow the same rules for parameters. The extra part in this year's Java-- language that is not part of Java is related to a new kind of expression, *array initializers* (e.g., `[10, 20, 30]`), that can only be assigned to a variable that is of type int array.

[2] https://en.wikipedia.org/wiki/Variadic_function

So, in short:

- as the last parameter: methods such as `void foo(int... a)` can have calls that have a variable number of arguments (e.g., `foo(10, 20)`), or receive a variable of type int array (e.g., `foo(anIntArray)`);
- as the return type: methods such as `int[] foo()` can return an array initializer (e.g., `return [10, 20]`) or a variable of type int array (e.g., `return anIntArray`);
- fields or variables: if they are an int array, they can be assigned an array initializer (e.g., `anIntArray = [10, 20]`);

1.4 An Example Java-- Program

For simplicity, your Java-- language specification, and consequently your program files, must contain just one class declaration. Figure 3 depicts an example of a simple Java-- program.

To exemplify the issue mentioned in the imports section, note the invocation to method "println" from the imported class "io", which is done as a simple call statement (`io.println(...);`). However, the argument of the "println" method has a compounded expression that includes two call expressions chained together (a `new` invocation, followed by a call to method "computeFactorial"). This behavior is allowed because it is being called from an instance of the class "Factorial", and we know that it contains a method "computeFactorial" that receives and integer, and returns an integer. If the method "computeFactorial" was not part of the class "Factorial", its invocation would have to be done outside and before the `io.println` call statement. This is an intended limitation that simplifies the semantic analysis that will be performed later by your compiler.

```
import io;

class Factorial {
    public int computeFactorial(int num){
        int num_aux ;
        if (num < 1)
            num_aux = 1;
        else
            num_aux = num * (this.computeFactorial(num-1));
        return num_aux;
    }
    public static void main(String[] args){
        io.println(new Factorial().computeFactorial(10));   //assuming the existence
                                                            // of the classfile io.class
    }
}
```

Figure 3: Simple Java-- program.

1.5 Lexical and Syntactic Analysis

These are the first stages of the compiler that you will implement, using ANTLR[3]. You need to adapt and extend the "Javamm.g4" file in your base project to follow the grammar and rules described in Section 1.1. The EBNF grammar has some unresolved issues that you need to solve during your implementation, including operator priority as described next.

The provided grammar has all the arithmetic operations in the Expression rule at the same level. You will have to reorganize these operations so that they have the same priority that is expected in Java[4], so that the generated AST already contains the arithmetic operations with a correct structure.

[3] https://www.antlr.org/

[4] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Operations with higher priority should be closer to the leaves in the AST, so they are evaluated earlier. On the other hand, lower priority should be closer to the "parent node" of the expression. In ANTLR, this is reflected by declaring high priority rules as the first choices of the production. For instance, between the add("+") and multiplication("*") binary operations, the multiplication has priority over the add operation. In ANTLR, this can be implemented by using a rule such as the one represented in Figure 4.

```
expr:
// ...
| expr "*" expr
| expr "+" expr
// ...
```

Figure 4: Example of rule precedence.

Furthermore, remember that the "negate" operator (!) has priority over any binary operation. For instance, in the expression `!a + b`, the negation is done over the variable a, not the "a+b" expression. This means that the negate rule also needs to be prioritized. Figure 5 represents the two previously mentioned formats in terms of AST, where the left one shows an incorrect AST structure and the right one is the expected structure.

```
unary [!]
   binOp [+]
      id [a]
      id [b]
```

(a) incorrect AST structure

```
binOp [+]
   unary [!]
         id [a]
   id [b]
```

(b) correct AST structure

Figure 5: Two different structures for the expression `!a + b`, which result from the order in which the grammar alternatives are specified.

1.6 AST Organization and Node Annotation

In this step you will proceed with the "clean up" of the AST by declaring and labeling the nodes you want to exist. For this, you will rename the nodes that are generated using the "#<name>" directive. For instance, in Figure 6, the left side shows a portion of the expression rules, where both the add and multiplication binary operations are named as "binaryOp", to easily identify them in the AST. At the right side of the example we can see the result when applied to the statement: `a = b + c;` (assuming the assignment statement as been named "assign"). The base grammar already names the subrules in the `stmt` and `expr` rules, you should also name the new rules you will introduce.

```
expr:
// ...
| expr "*" expr #binaryOp
| expr "+" expr #binaryOp
// ...
```

```
assign
   id
   binaryOp
      id
      id
```

Figure 6: Example of grammar and AST enhanced with node labels.

As you can see in the example, by just looking at the AST we do not have all the necessary information to understand the assignment. You should annotate the AST to add the missing data, which in the example above is for instance, to provide the actual value of the token "id" and the operation type inside the respective nodes, represented in square brackets below. This is done by adding an assignment in the rules for each terminal we intend to store, as exemplified on the left side of Figure 7. The right side shows and example of a annotated AST for the example above.

```
expr:
// ...
| expr op="*" expr #binaryOp
| expr op="+" expr #binaryOp
// ...
```

```
assign
   id [a]
   binaryOp [*]
      id [b]
      id [c]
```

Figure 7: Example of grammar and tree enhanced with attributes.

1.7 Interfaces

1.7.1 Lexical and Syntatic Analysis Interfaces

The first compiler stage uses three main interfaces, namely *JmmParser*, *JmmNode*, and *JmmParserResult*. To implement this stage your compiler needs to extend the *JmmParser* interface. The base project already contains an implementation you can use, *JmmParserImpl*. This interface contains two methods: `getDefaultRule`, which returns a String with the name of the top rule of your grammar (`program`, in the base project); and `parse`, which receives a String with the Java-- code to parse, the name of the starting rule, and a *Map<String, String>* with the configuration, returning a *JmmParserResult* instance as result. As the name suggests, this method expects you to parse the input Java-- code, and provide the result of the compilation as a *JmmParserResult* instance, either with the resulting AST or an error if parsing failed.

A new *JmmParserResult* instance expects three input arguments: the root node of the AST, a list of reports, and the *Map<String, String>* with the configuration. The root node is the result of parsing the input code and generating an AST, and is of the type provided earlier (e.g., `program`). Every other node of the tree can be accessed through the root node. The nodes of the AST must be *JmmNode* instances, an interface also provided in the base project. The *JmmNode* interface represents a node of the AST we will be using during the project, and will be used for navigating the AST and accessing information about each node.

The list of reports collects all the logging, debugging, warning, and error information that might occur during compilation. For instance, if there is an error during compilation, it will return a *JmmParserResult* instance with at least one Report instance of type Error. It will use the *JmmParserResult* constructor and pass a null *JmmNode* if one could not be created, but it could have used the convenience static method `newError` for creating an instance of *JmmParserResult* with just an error report inside.

To help you with the task of developing your compiler, in addition to these interfaces and the base implementation of the project, we provide you with other interfaces and libraries. Among these, we can highlight the *JmmVisitor* interface and its several default implementations, which allow you to quickly traverse and add functionality to the tree nodes without changing them directly. Two other important libraries[5] are *SpecsCollections* and *SpecsIo*. The former provides you with methods to manipulate and filter Java collections, while the latter provides you with methods to interact with files and folders in an OS-agnostic way.

2 Semantic Analysis

This compiler stage expects you to analyse and validate the Abstract-Syntax Tree (AST) in terms of semantics. This includes, for instance, verification of the type of operations in invoked methods, if the methods for a call exists, if the types of the arguments match, etc.

Semantic analysis is used both to verify the validity of the input code (e.g. if a given variable that is used exists, if operands have the correct types) and to extract contextual information from the AST (e.g. calculate the return type of a given operation). You will be doing these two operations (more or less) at the same time, and to aid the analysis, you will use the symbol table.

For this project, the compiler will only need to verify the semantic rules in expressions, including the arguments in the invocation of methods that belong to the class being analysed (i.e., calls to

[5] https://github.com/specs-feup/specs-java-libs/tree/master/SpecsUtils

imported methods are assumed to be semantically correct). For a list of the main analyses you will need to perform please refer to the checklist at Section 3.3. This list already includes the main expected semantic checks.

If the analysis detects semantic errors, they must be reported, and the compiler will abort execution after completing the semantic analysis. Note that the sometimes it may make sense to stop the analysis immediately after certain errors, and other times you can just add the error to the "list of reports" and keep doing the semantic analysis. This will be left open for you to explore. We provide a base implementation (i.e., *JmmAnalysisImpl*) for the semantic analysis, that you must complete.

Tip: Use the analysis process to annotate the tree with extra information when you think it is relevant. For example, in the next checkpoint you will need to convert the AST to OO-based Low-Level Intermediate Representation (OLLIR) code, which requires all operands and operations to be annotated with the resulting type. For the expression "1 + 2", since "1" and "2" are integers, and the result of the add operator in this case is also an integer. Since for each expression node you will need to know its type, it can be beneficial to add this information to the AST while you are "analysing" those nodes (e.g. add a property "type"="int" in the node). Alternatively, you can write a (recursive) function that receives a node and calculates on-the-fly the type of the node.

2.1 Interfaces

This stage expands on the use of the *JmmAnalysis* interface, using the *JmmAnalysisImpl* class that you can find in your project, in package `pt.up.fe.comp2025.analysis`. You can always use the IntelliJ shortcut to find the class (i.e., `Ctrl+N` or double tap shift and write `JmmAnalysisImpl` to quickly find the class). Please consider the methods in this class that have the following signature:

JmmSemanticsResult buildSymbolTable(JmmParserResult parserResult)

JmmSemanticsResult semanticAnalysis(JmmSemanticsResult semanticsResult)

You can see that this stage is split in a sequence of two steps, building the symbol table, and applying the analysis passes. It now expects as input the result of the previous stage: the *JmmParserResult*. If you have not done it yet, please take a look at the class *JmmParserImpl*, in package `pt.up.fe.comp2025.parser`. Regardless, the output of the previous stage, *JmmParserResult*, will contain the root *JmmNode* of the parsed code, and the input configuration.

2.2 Symbol Table

The symbol table is a data structure that stores information related to the variables and other symbols in the source code. For instance, which variables are declared inside a given method, or if the class extends another class. We provide a *SymbolTable* interface and a base implementation (i.e., `JmmSymbolTable`) and builder (i.e., `JmmSymbolTableBuilder`), that you must complete.

You do not need to support method overloading (i.e. when two or more methods have the same name but different parameters), so using the name of the method as its signature is sufficient in this case. If you want to support method overloading as an extra, consider that your method signature must have information about the name of the method and its parameters.

The symbol table should include information regarding:

- Imported classes
- Declared class (and its superclass)
- Fields inside the declared class
- Methods inside the declared class
- Parameters and return type for each method
- Local variables for each method

Each item corresponds to a method in the `SymbolTable` interface, so please refer to code to see what each method should return. To build the symbol table you will need to get the information from the AST. You can either extend and complete the approach in the base project, which creates a class that uses the Visitor pattern for each analysis (e.g. extend *AJmmVisitor*), or manually visit the AST (e.g., for each node, apply all analyses).

2.3 Analysis Passes

In this step you will receive an instance of *JmmSemanticsResult*, created by the method `buildSymbolTable()`. This instance is built based on the previous *JmmParserResult* (*JmmNode* root, reports and configuration), and additionally contains a *SymbolTable* instance, representing the symbol table. The objective will be to return an instance of *JmmSemanticsResult* with additional reports, if needed.

This analysis is composed by doing a set of passes throughout the AST (starting from the root node), where each pass tries to provide one or more semantic validations. Therefore, the idea of each pass is to analyse each node of the AST and verify if they are semantically correct.

To help in the development of this phase, the `JmmAnalysisImpl` class already provides logic to execute a list of analysis passes. Looking at this class you can see that it contains a field to store the analysis passes to perform in a list of `AnalysisVisitor` instances. An `AnalysisVisitor` extends a `PreorderJmmVisitor` and implements `AnalysisPass`, an interface that contains a single method:

List<Report> analyze(JmmNode root, SymbolTable table);

The method *analyze* receives a node of the AST and the symbol table, and returns a list of Report instances. If any problem is detected, there should be at least a Report with ReportType ERROR in the list.

An implementation of this interface can either just analyse the given node, or analyse the given node and its descendants (i.e., what `AnalysisVisitor` does). Although the provided code does the latter, you can choose if you want to implement the former approach. In this case, change the return type of the method `buildPasses()` from `AnalysisVisitor` to `AnalysisPass`.

You can have many implementations of *AnalysisPass*/*AnalysisVisitor*, each for a single semantic analysis, or very few implementations where each implementation performs several analyses. If you add instances of your *AnalysisPass* implementations to the list *analysisPasses*/*AnalysisVisitor* inside the `buildPasses()` method, they will be automatically called with the current code you have. Note that the current implementation will call all *AnalysisPass*/*AnalysisVisitor* instances before stopping the analysis, if you want to stop early, you will have to modify the code.

You can implement an *AnalysisPass* without using the Visitor pattern, but for certain analyses it can help. Please check the class *UndeclaredVariable* for an example of how you can extend *AnalysisVisitor* to implement an *AnalysisPass*.

Tip: You can use these passes to add extra information to the AST, such as calculating and adding the type of each expression. This might help other passes and aid in the code generation phase of the next part of the project, and avoid traversing the AST multiple times to obtain the same information again and again. In the other hand, do not be afraid to do multiple visits over the AST, if that helps with the readability and organization of your code.

3 Checklist

The following checklist applies to groups with 3 elements. If your group has 2 elements, you will not need to implement language features related to **fields**, **arrays** and **varargs**. This means that during the evaluation of your work, we will not apply tests that contain code with those elements.

3.1 The Java-- Language

By the end of **checkpoint 1**, it is expected that you output an annotated AST and/or a list of reports (errors, warnings, debug messages, or any other relevant reports).

The following is a checklist of work regarding the grammar of the language:

- ☐ Complete the JAVA-- grammar in ANTLR format
  - Import declarations
  - Class declaration (structure, fields and methods)
  - Statements (assignments, if-else, while, etc.)
  - Expressions (binary expressions, literals, method calls, etc.)
- ☐ Setup node names for the AST (e.g. "binaryOp" instead of "expr" for binary expressions)
- ☐ Annotate nodes in the AST with relevant information (e.g. id, values, etc.)
- ☐ Used interfaces: *JmmParser*, *JmmNode* and *JmmParserResult*

3.2 Symbol Table

By the end of **checkpoint 1** it is expected the symbol table to be populated with the information regarding all the symbols in the input code. The following is a checklist to aid the development of the symbol table:

- ☐ Imported classes
- ☐ Declared class
- ☐ Fields inside the declared class
- ☐ Methods inside the declared class
- ☐ Parameters and return type for each method
- ☐ Local variables for each method
- ☐ Include type in each symbol (e.g. a local variable "a" is of type `X`. Also, is "a" array?)
- ☐ Used interfaces: *SymbolTable*, *AJmmVisitor* (the latter is optional)

3.3 Semantic Analysis

By the end of **checkpoint 1**, it is expected that you perform a set of analyses that will test the correctness of the code and that report an error if the implemented rules are not respected.

3.3.1 Types and Declarations Verification

- ☐ Verify if identifiers used in the code have a corresponding declaration, either as a local variable, a method parameter, a field of the class or an imported class
- ☐ Operands of an operation must have types compatible with the operation (e.g. int + boolean is an error because + expects two integers.)
- ☐ Array cannot be used in arithmetic operations (e.g. array1 + array2 is an error)
- ☐ Array access is done over an array
- ☐ Array access index is an expression of type integer
- ☐ Type of the assignee must be compatible with the assigned (an_int = a_bool is an error)
- ☐ Expressions in conditions must return a boolean (`if(2+3)` is an error)
- ☐ "this" expression cannot be used in a static method

☐ "this" can be used as an "object" (e.g. `A a; a = this;` is correct if the declared class is A or the declared class extends A)

☐ A vararg type when used, must always be the type of the last parameter in a method declaration. Also, only one parameter can be vararg, but the method can have several parameters

☐ Variable declarations, field declarations and method returns cannot be vararg

☐ Array initializer (e.g., `[1, 2, 3]`) can be used in all places (i.e., expressions) that can accept an array of integers

3.3.2 Method Verification

☐ When calling methods of the class declared in the code, verify if the types of arguments of the call are compatible with the types in the method declaration

☐ If the calling method accepts varargs, it can accept both a variable number of arguments of the same type as an array, or directly an array

☐ In case the method does not exist, verify if the class extends an imported class and report an error if it does not.

  – If the class extends another class, assume the method exists in one of the super classes, and that is being correctly called

☐ When calling methods that belong to other classes other than the class declared in the code, verify if the classes are being imported.

  – As explained in Section 1.2, if a class is being imported, assume the types of the expression where it is used are correct. For instance, for the code `bool a; a = M.foo();`, if `M` is an imported class, then assume it has a method named foo without parameters that returns a boolean.