Università della Svizzera italiana

Faculty of Informatics

Bachelor Thesis

June 16, 2018

Byte code to source code mapper

Renzo Cotti
cottir@usi.ch

Abstract

This project is an extension for a Java compiler which allows for accurate mapping from bytecode to source code, with an intra-line granularity instead of inter-line.

Advisor
Prof. Matthias Hauswirth

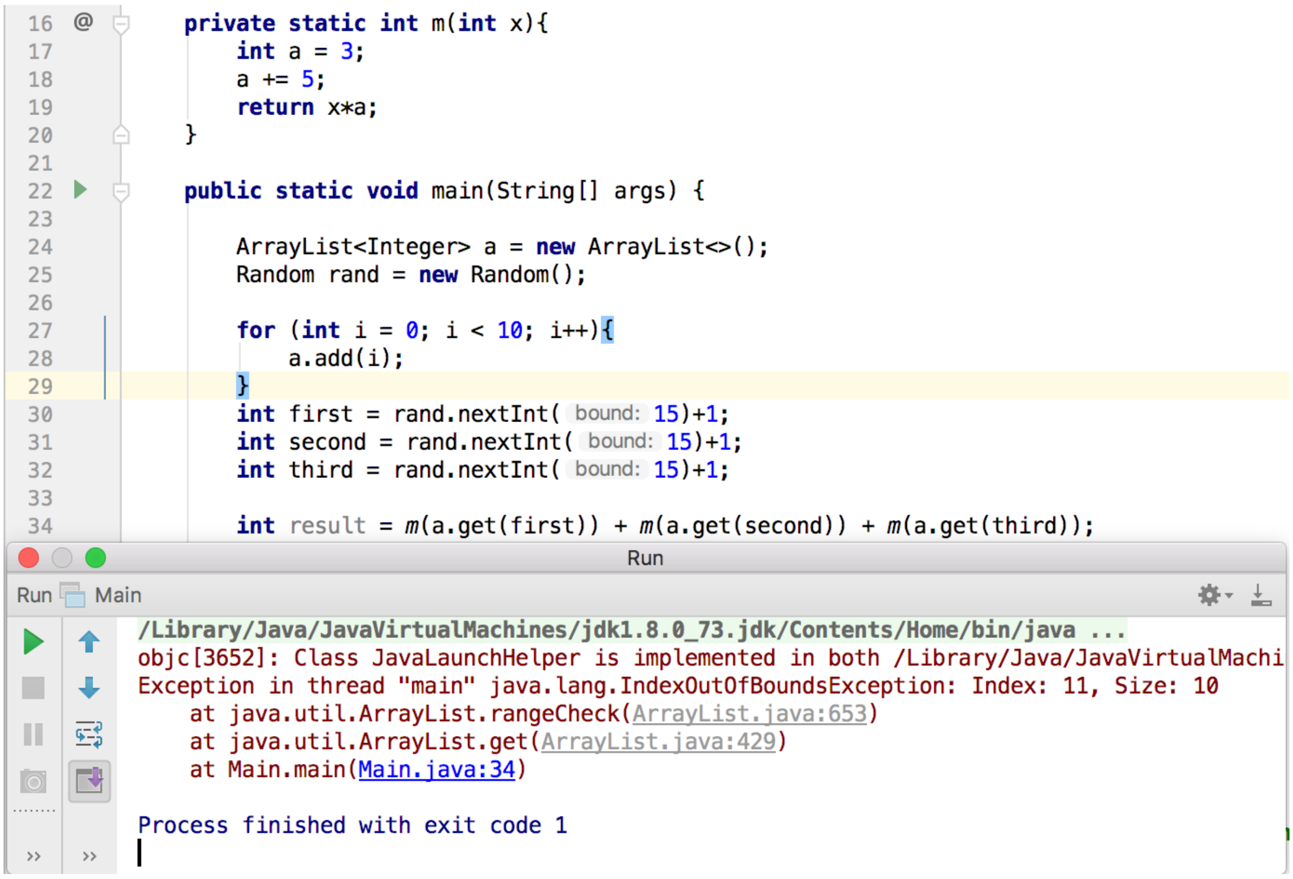
Advisor's approval (Prof. Matthias Hauswirth): Date:

Contents

1 Introduction

Debuggers nowadays are very useful. Whether it is for a logic problem in our programs, or a typo that makes the final outcome wrong, they can help us spot the error and get back on our tracks.

The granularity offered by many debuggers is unfortunately only inter-line – i.e. if an exception is thrown, it will generally state "exception on line x". This can be somewhat limiting.
For example, we can see a problem in this screenshot:



```
16      private static int m(int x){
17          int a = 3;
18          a += 5;
19          return x*a;
20      }
21
22      public static void main(String[] args) {
23
24          ArrayList<Integer> a = new ArrayList<>();
25          Random rand = new Random();
26
27          for (int i = 0; i < 10; i++){
28              a.add(i);
29          }
30          int first = rand.nextInt( bound: 15)+1;
31          int second = rand.nextInt( bound: 15)+1;
32          int third = rand.nextInt( bound: 15)+1;
33
34          int result = m(a.get(first)) + m(a.get(second)) + m(a.get(third));
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_73.jdk/Contents/Home/bin/java ...
objc[3652]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachi
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 11, Size: 10
    at java.util.ArrayList.rangeCheck(ArrayList.java:653)
    at java.util.ArrayList.get(ArrayList.java:429)
    at Main.main(Main.java:34)

Process finished with exit code 1
```

Figure 1. Example of limited usefulness of a line-based debugger

We declare an ArrayList in which we put 10 values, then we try to access 3 elements, some of which may be out of range.
We have no way of knowing precisely when the exception was thrown on line 34; was it the first call to m? Or maybe the second?
Unfortunately, the stack trace can't help us here, and we'd have to go through the code step by step in order to find the value of the variables which are randomized.

We can see another potential limitation by comparing the most famous IDEs for Java – IntelliJ, Netbeans, Eclipse – and see at which points a debugger stops when stepping.

We will use this function as our example:

```
public int m(int x) {
    int a = 3;
      a = a+5;
    return x * a;
}
```

All 3 IDEs executed the same pattern when debugging:

- Step 1: computed x = 3
- Step 2: computed a = 3
- Step 3: computed a = 8
- Step 4: return 24

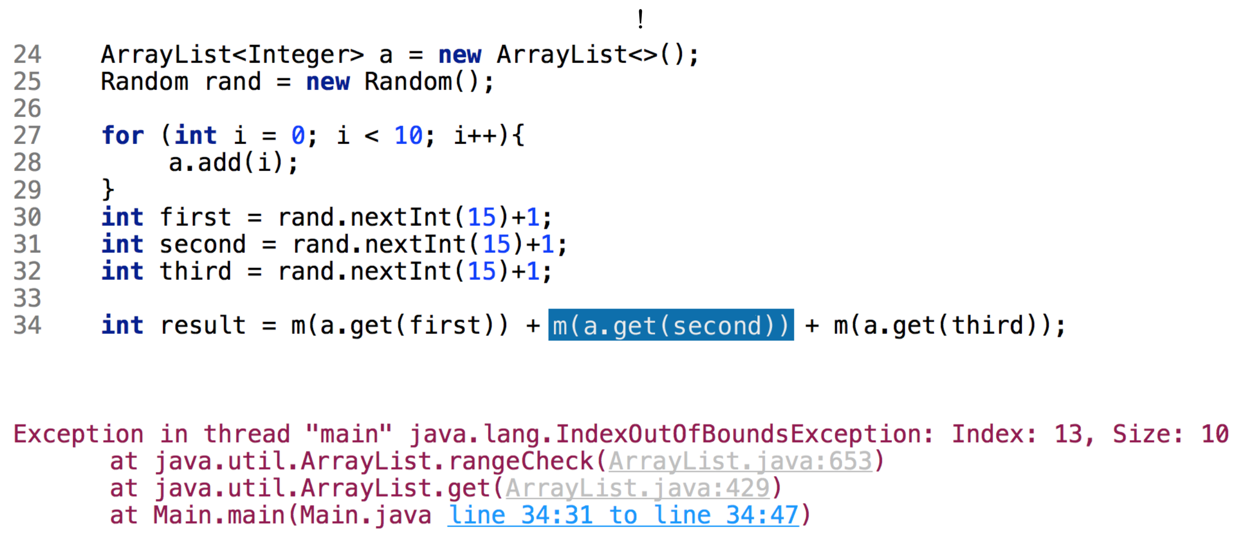
Now, if we break down the function into bytecode, we can see that it's composed of 10 instructions:

PC	Bytecode instruction
0	ICONST_3
1	ISTORE_3
2	ILOAD_3
3	ICONST_5
4	IADD
5	ISTORE_3
6	ILOAD_1
7	ILOAD_3
8	IMUL
9	IRETURN

This means that we're missing on 6 bytecode instructions when we're debugging. Some of them are not very interesting debugging-wise – such as ISTORE – however some do have value to them.
For example, in none of the IDEs used was there a way to see the previous value of an assignment in the current step – i.e. in a=a+5 we couldn't see a=3 → a=8, but just the end value of the operation, a=8.
Analysing the top of the stack and the ILOAD instruction for example could allow to know that missing piece of information: looking at the top of the stack before the ILOAD would show the value prior to the ILOAD, doing so after the ILOAD would show the value of the loaded variable and looking at it after the IADD would show the result of the addition.

2 Goal

The goal is to provide a base for future production of debugging and program analysis tools in a convenient package by extending a modular compiler for Java written in Java, ExtendJ.
An example application could be a more informative debugger or a way of getting more useful information from runtime errors, as exemplified below.



```
                                         !
24    ArrayList<Integer> a = new ArrayList<>();
25    Random rand = new Random();
26
27    for (int i = 0; i < 10; i++){
28        a.add(i);
29    }
30    int first = rand.nextInt(15)+1;
31    int second = rand.nextInt(15)+1;
32    int third = rand.nextInt(15)+1;
33
34    int result = m(a.get(first)) + m(a.get(second)) + m(a.get(third));
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 13, Size: 10
      at java.util.ArrayList.rangeCheck(ArrayList.java:653)
      at java.util.ArrayList.get(ArrayList.java:429)
      at Main.main(Main.java line 34:31 to line 34:47)
```

Figure 2. Mockup of what the developed system could do

Here we can see that – even though we have exactly the same code as before – the Java Virtual Machine now knows which source code part is associated with the bytecode instruction that triggered the exception.

3 Project Description

The project consists in creating an extension for ExtendJ that allows to accurately map from bytecode back to source code lines, with an intra-line granularity.
We modified ExtendJ so that relevant positional information – about which fragment of source code emitted a certain bytecode – is saved inside the .class file.
All .java files compiled with this project produce a .class file that contains relevant information which previously had to be deduced at runtime, along with an external file containing information about the program structure.

4 Plan

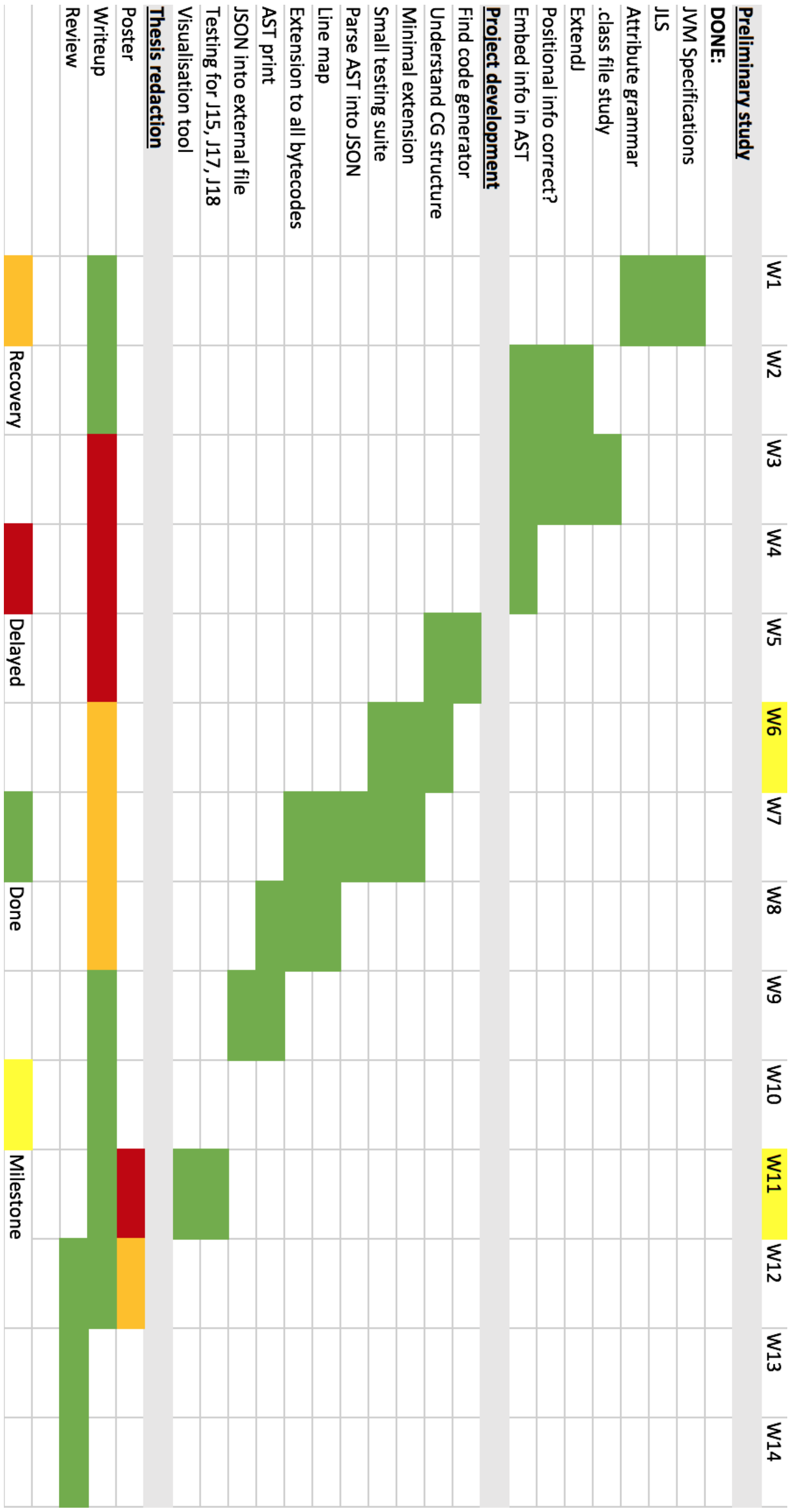


Figure 3. Gannt chart representing the planned work and the actual schedule

5 ExtendJ introduction

5.1 Abstract Syntax Tree

An Abstract Syntax Tree – AST in short – is a tree which stores in its nodes – ASTNodes – important information about the code structure.[1]

5.2 ExtendJ

ExtendJ is a compiler of Java written in Java.
In order to generate a .class file, it translates the source code provided into an AST, and after translates the AST into bytecode.

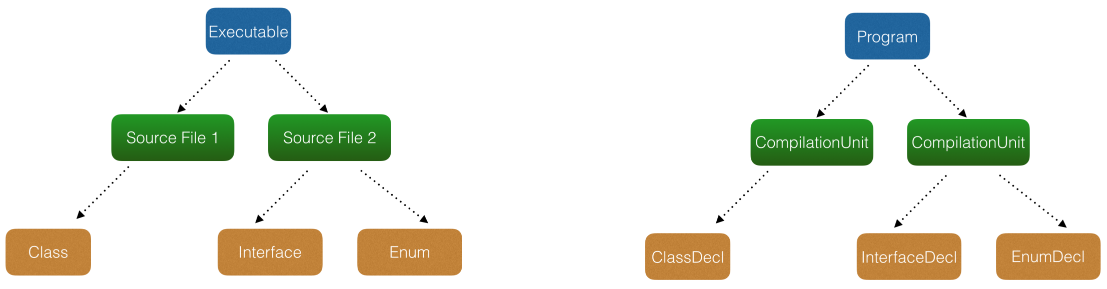


Figure 4. On the left we can see what an executable is made of. On the right, we can see how ExtendJ structures the corresponding AST.

This compiler allows for modules to be added to the core compiler, called *aspects* – Java files saved under the .jrag extension.[2]
They might change drastically how the compiler behaves, or they might simply be an static analysis tool.
In general, aspects add new behaviours to ASTNodes, like it's the case with the aspect that was created for this project, Position.jrag:

```
 aspect Position {
  syn int ASTNode.startLine() {
    return getLine(getStart());
  }
  [...]
}
```

The purpose of this extension is to ease access to the source code position information relative to each ASTNode, by creating 4 new attributes startLine, startColumn, endLine and endColumn.

6 ExtendJ AST structure

Traditional AST use external data structures to keep track of information.
ExtendJ, however, produces AST with information embedded in each node, by declaring attributes that belong to each type of node.[3]

Over the next few pages, I will explain how ExtendJ translates the following example Java code into an AST – bigger examples would lead to bigger ASTs, thus the briefness of the example code:

```java
public class Test {
    private static String name;

    public Test(String n){
        this.name = name;
    }

    public int m(int x) {
        int a = 3;
        a = a+5;
        return x * a;
    }
}
```

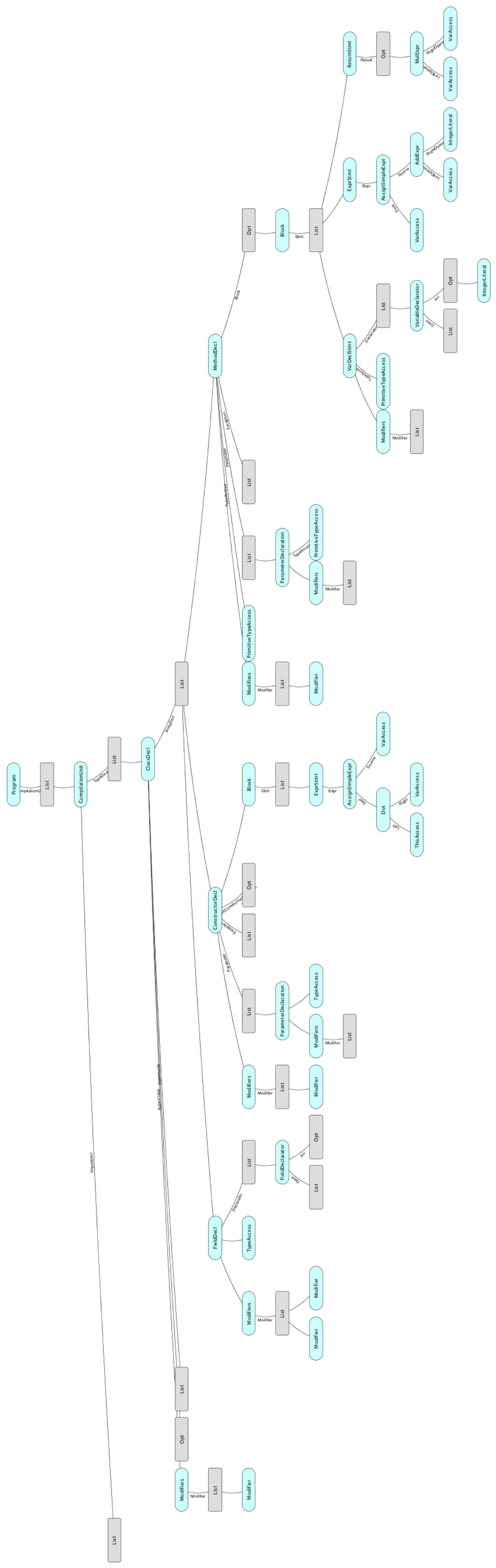


Figure 5. AST of a simple Java class

6.1 Program: the core node

The root node of the AST is the Program node, and its children are CompilationUnits, one for each source file; in this case, our program is Test.java.
Each source file compiled has a corresponding CompilationUnit; inside each, there is a List of Class declarations –ClassDecl–, or equivalent, like an Interface declaration.
In the following pages, we will talk about the main characteristics of a ClassDecl: Modifiers, Fields, Methods, Constructors.

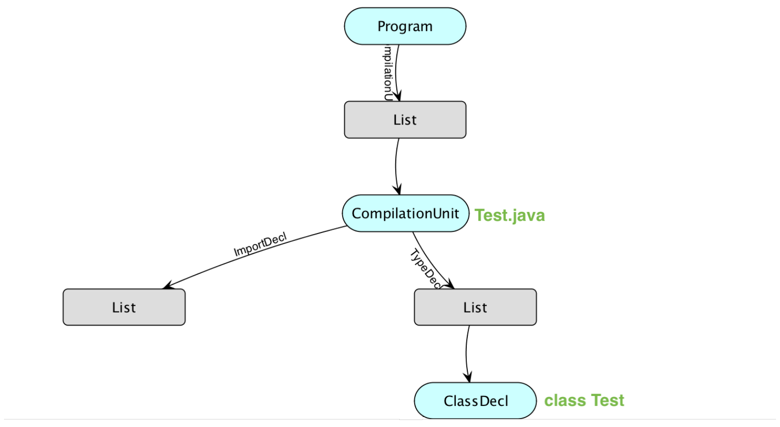


Figure 6. The top level of the AST

6.2 Modifiers

Here we can see the class modifier; it's declared as a List since a class can have multiple modifiers – e.g. a public static class.
This is a common trend for ExtendJ, oftentimes nodes declare Lists when it seems overly laborious to do so, but that is to allow multiple declarations; e.g. in the previous image, one can see that a CompilationUnit has declared a List of ClassDecl. That will be used for when we have two classes declared in a single file.

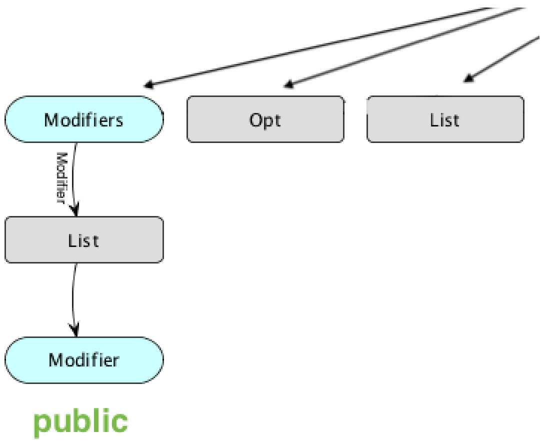


Figure 7. The class modifier

6.3 Fields

We can see in more detail how the code

```
private static String name;
```

gets translated in an AST; First of all, it's identified as a FieldDecl, a field declaration, with two modifiers, private and static.
Then, a TypeAccess (String) is detected, and we list each of the declaration in a List. As mentioned before, the list is because we could have multiple declarations on the same line, i.e.

```
private static String name, surname, address;
```

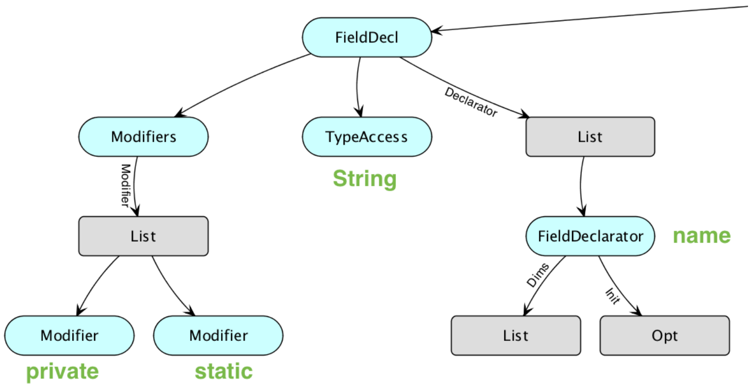


Figure 8. A field declaration

6.4 Methods

Here we can see the structure of a generic method, in this case:

```
    public int m(int x) {
        int a = 3;
        a = a+5;
        return x * a;
    }
```

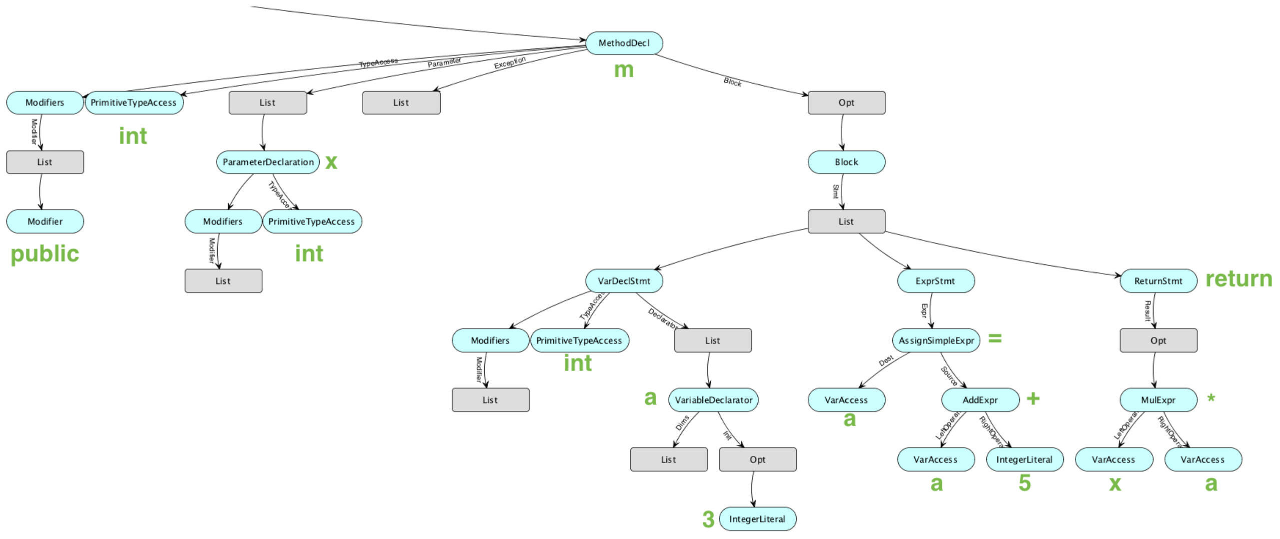


Figure 9. A method declaration

We can see that a method has, as usual a List of modifiers –in this case just public; furthermore, it has a List of Parameters –in this case an int named x–, and inside the Block node we can find all the statements of the method –the VarDeclStmt int a = 3, the AssignSimpleExpr a = a + 5, and the ReturnStmt x*a–. It also has a List dedicated to the Exceptions the method can throw, but for this specific case it's empty.

6.5 Constructors

The structure of the constructor is similar to a normal method. Here for completeness.

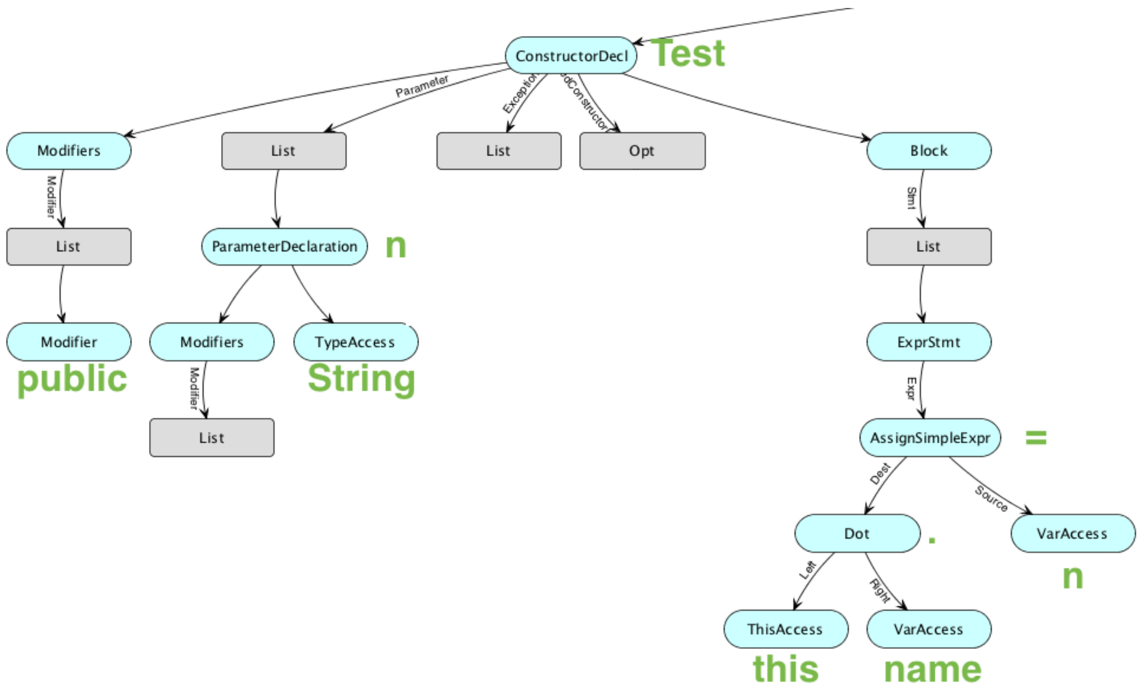


Figure 10. A constructor declaration

6.6 Design choices

Thanks to ExtendJ's modular structure, most of the changes I was able to apply through the creation of aspects in .jrag files.
However, some of them went to modify the core of the compiler –e.g. modifying the emit methods for bytecodes–, and it was unavoidable to modify the source code of the compiler itself; this is less than ideal –it breaks the Open/Close principle–, but it was unavoidable.

7 JastAdd grammar

JastAdd is an Attribute-based grammar on which ExtendJ is built.
During this project, I didn't have to deeply understand the system, although it certainly helped clearing up the hierarchy of the ASTNodes.
ExtendJ uses .ast files, which get compiled and end up generating .java files; in the specific case, all of the ASTNodes, many of which have a very similar structure, are written in the Java.ast file.
Incidentally, this list of generated ASTNodes turned out to be very useful when creating a testing suite for this project – see Evaluation section.

7.1 Syntax overview

Here is a brief overview of the syntax used in the .ast files[4]:

Syntax	Meaning
A	simple AST class
B: S	AST subclass (B subclass of S)
B ::= Y	Child component Y
X ::= C*	List of Cs
Y ::= [D]	optional component D
Z ::= <E>	token component E (default type is String);
Z ::= <F:Integer>	token component F, type Integer

7.2 JastAdd attributes

Inside aspects – extensions for ExtendJ – , we can declare attributes of ASTNodes; these can be of type syn – synthesized or inh – inherited.
syn attributes are used to propagate the information up the AST, towards the root node, whereas inh attributes propagate the information downwards.[5]
In other words, if a node has a syn attribute, in order to compute its value it will do some operation with its children; if it has a inh attribute, it will do some operation with its parent.
If we wanted to add attributes to class Example, we'd do it like this:

```
syn int Example.a() = 3;
//it's then accessed by
Example e = new Example();
e.a();    //returns 3
```

As we can see, attributes are basically functions we add to a given class. This is somewhat limiting since we can't set a new value to an attribute, i.e. e.a() = 3 isn't valid.

8 Bytecode generation

8.1 Java Virtual Machine

The Java Virtual Machine, or JVM for short, is a virtual machine in which all Java code gets run.
It's composed of a stack, on which we push values such as variables, and through which we perform operations – such as addition of the first two elements on the stack – ; it's composed also of a heap, used for example for object allocation.

8.2 Java .class file

When a .java file gets compiled, a .class file is the output. This file contains all the information the JVM needs to run the program.
There are 10 components to it[6]:

- The magic number 0xCAFEBABE, the first 8 bytes of the file stating that this is a java class file
- The version of the class file format
- The constant pool, storing strings and other constants
- The access flags, stating whether the class corresponding to this file is abstract, static, ...
- The super class, the possible class this class inherits from
- The interfaces, any interfaces this class might implement
- The fields, any fields this class might have
- The methods, any methods this class might have
- The attributes, any attribute the class might have, such as the name of the source file, the LineNumberTable,...

Relevant to this project are the constant pool – since we will write useful information in it –, the methods – since the mapping will be made for the methods. As a note, methods in the .class file are a series of bytecode instructions.
Attributes are also important: the LineNumberTable attribute, for example, stores information about which line of source code generated which bytecode.
We will add an attribute of our own to the class file, as we will explain later on.

8.3 Bytecode introduction

Bytecode is the JVM equivalent of assembly for normal binaries.
There are 255 different bytecode instructions – the name *byte*code comes from the fact that a single bytecode instruction is stored in a single byte.[7]
A few examples are:

Bytecode instruction	function
ISTORE	Stores an integer inside a variable
ILOAD	Loads an integer from a variable
IADD	Sums the first two operands on the stack
IRETURN	returns from a function call the value on top of the stack

8.4 Adding an attribute to a .class file

Declaring a new Attribute was fairly straightforward, I used as a guideline the attribute LineNumberTable[8] and worked from there, ending up with PositionTable.jrag, containing this:

```
aspect PositionTable{
    public class PositionTableAttribute extends Attribute {
        public PositionTableAttribute(CodeGeneration gen) {
          super(gen.constantPool(), "PositionTable");
          for (CodeGeneration.PositionEntry e : gen.positionTable) {
            u2(e.start_pc);
            u2(e.start_line);
            u2(e.start_column);
            u2(e.end_line);
            u2(e.end_column);
            u2(e.id);
          }
        }
    }
}
```

The call to super writes relevant information of this attribute into the constant pool – such as the name of the attribute; the u2 calls take the parameter, an integer, and push two bytes containing such information; this means that the range of lines ExtendJ can cover is 0-65536.
This aspect would generate an attribute storing information like this:

Program Counter	Start Line	Start Column	End Line	End Column	Node ID
0	1	0	4	5	42
...	...	...	...	...	...

Details about what each field stores can be found in the following section.

8.5 Design of the PositionTable

In order to maximise the amount of usable information inside a .class file, I decided to add an attribute directly into it, rather than writing all the relevant information to a separate file.
This was also done to ease the access of this information of a user of this system.
The PositionTable attribute was created following the same structure as the LineNumberTable attribute.
Initially, I just saved the Program Counter and the start and end column, since I assumed that a source code fragment that emitted a bytecode instruction started and ended on the same line.
Through testing, I found out that it wasn't always the case.
Towards the end, the ID field was added in order to allow for mapping between the JSON AST and the PositionTable.

9 Writing to the .class file

9.1 How bytecode instructions get written in the class file

After the AST gets generated, a call to the bytecodes method creates a CodeGeneration object, which starts writing information to the class file – such as the famous 0xCAFEBABE bytes at the beginning of each .class file, or the Constant Pool.
After this, it iterates over the AST, finds the ASTNodes that have to generate a bytecode instruction and calls an emit method for the specific bytecode – for clarity, I shall refer to this kind of methods as emitX methods.
An example of such emitX methods can be emitStore or emitLoad.
Each emitX method at one point calls an emit method – for sake of clarity let's call it plainEmit – which writes a certain bytecode – for example Bytecode.IRETURN – to the .class file.

9.2 Refactoring choice

As a reminder, the call tree done to generate bytecode is:

1. a method calls emitX
2. emitX calls plainEmit
3. plainEmit writes the bytecode in the .class file

Since we want to make this project useful to somebody in the future, the mapping has to be as thorough as possible, meaning that as close as possible to every bytecode instruction has to be mapped.
In order to do the mapping, we need to work with the AST – which contains the positions and the bytecode index; the Abstract Syntax Tree is available to the method calling emitX, and we want to pass it to either emitX or plainEmit.
Ideally one would have to pass it to the plainEmit method – that way, we're guaranteed that every bytecode emitted is mapped. By searching in the project files for uses of that method, we see that there are 675.
Unfortunately, we would need to pass the AST from the method calling emitX, to emitX and finally to the plainEmit methods; looking in the project, we find just short of a thousand uses of emitX methods.
Thus, we decided to stick to editing the emitX methods only in order to minimise the amount of refactoring necessary.

9.3 JSON tree

To complement the information given by the mapping, we decided to add some information about the program structure.
Having access to the AST, we decided to parse it into a JSON object, by keeping the information we found most relevant.
The JSON structure of the parsed AST is the following:

1. ID
2. class of the node
3. JSON object containing the position information
4. list of children

The class of the node is the dynamic type of the current ASTNode, e.g. ClassDecl, VarDeclStmt,...
The position object contains start_line, start_column, end_line, end_ column just like the PositionTable does.
The list of children does not have a bound, and if it is empty the current node is on the fringe of the tree.
The original idea was to store this into the class file directly, however there were complications – detailed in the design choices.

9.4 JSON AST Design choice

The idea behind making available the AST to the user is so that he/she can have access to the program structure so to gain a clearer understanding of the code.
The choice of encoding were either Java serialisation – however that'd require access to the whole ExtendJ project and classes in order to deserialise it – or using the JSON format.
The choice was the latter, so that it would be easier to access, parse and reuse the information.
The idea behind encoding as much as possible inside the .class file was to make it easy for somebody using this system to use this information – everything in one place.
That proved to be successful when it came to adding an attribute and adding entries to it.
Unfortunately, the JSON AST was impossible to encode in the Constant Pool, because the DataOutput-Stream that was used has a limit of 32k bytes written at a time.
It would theoretically be possible to add the AST, but we'd have to split it up in several chunks, thus also forcing somebody that wants to use the information to have to join the pieces before using them; writing it to an external file was the cleaner option.
The file name we write the JSON to is dependent on the .java file compiled: if the name is file.java, the resulting file will be fileAST.txt, so to make it easy to find out which file produced what.
On top of that, each .class file produced contains in the Constant Pool an entry which tells the ID of the ClassDecl/InterfaceDecl/... that produced the file.
For example, the last entry in the Constant Pool might be "This class file was generated by node 131"; if one were to look for node with ID 131 in the JSON AST, he'd find the node corresponding to the main Class of the compiled file.

10 Evaluation

10.1 Testing file: J14.java

During my project, I needed a file to test the amount of coverage my mapping was doing.
Wanting to build something small that worked and extend from there, I started by just considering java 1.4 features only, planning to extend it to java 1.5 and further, if time allowed.
This file was generated by using all the standard features of the language – for loops, while loops, variables, sums, divisions,...; once most of it was done, I researched on the web the language version specific features – such as the assert statement.
It has to be noted that classes like ArrayList, HashMap,... – classes requiring an import statement – were excluded, since those aren't of the core java 1.4 language, rather belonging to external libraries.
Finally, wanting to complete the coverage as much as possible, I looked at the ASTNodes subclasses that ExtendJ generated from the .ast files mentioned before: those gave me a list to extend it as much as possible.
The list of features tested can be summarised as follows:

- operators: pre-post increment/decrement, sum, division,...
- loops: while, do while, for, ...
- bitwise operations: &, |, ^ , ...
- conditionals: if else, switch, ? notation, ...
- classes: interfaces, classes, private/public, fields,...
- others: assertions, exceptions,...

10.2 List of ASTNodes used for testing

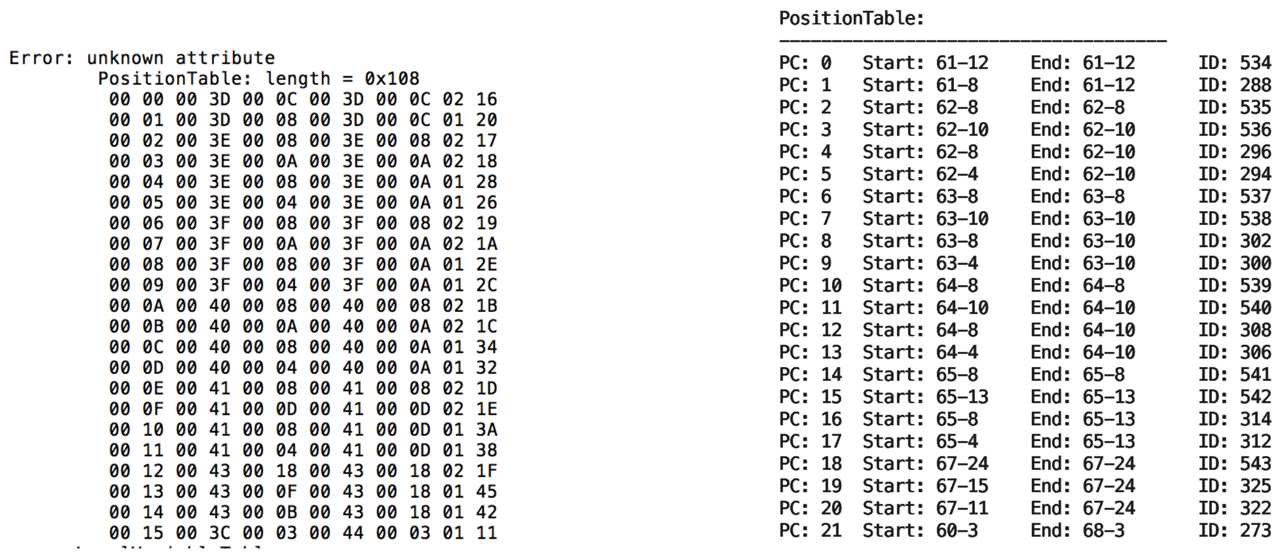
In order to maximise coverage for the testing, I looked into the list of ASTNodes files generated by ExtendJ and added a statement for each relevant node – e.g. seeing a PreIncExpr node made me add ++i to the testing file.
The resulting file might still miss some operations, since the list included unfortunately also nodes that didn't get implemented – e.g. the ASTNode class –, and some other nodes that I couldn't understand what statement they were referring to – IntegerType, for example.
The Addendum contains the full list that was used.

10.3 Decompilation

In order to see any results at all of my work, I needed to compile the compiler, compile the testing file into a .class file using this compiler and decompile the .class file.
The tool used for decompilation was the javap command, since it was readily available.
However, choosing this tool meant that my PositionTable attribute didn't get recognised, meaning that the information encoded in it was visualised as raw bytes – see the example tool that was developed.
This problem could be solved by writing an extension for javap or using another tool for decompilation.

10.4 Testing suite

After having managed to insert entries into the PositionTable, I then developed a tool to test the coverage, a script that compiles the ExtendJ project and produces a .jar compiler, takes j14.java, compiles it with the .jar file and decompiles the .class file using javap.
At this point the information in the .class dump is parsed through a java program that retrieves and structures the information – the PositionTable entries are visualised by javap as raw bytes, since the attribute isn't recognised as a "standard" attribute that one would find in a .class file.



```
Error: unknown attribute
        PositionTable: length = 0x108
         00 00 00 3D 00 0C 00 3D 00 0C 02 16
         00 01 00 3D 00 08 00 3D 00 0C 01 20
         00 02 00 3E 00 08 00 3E 00 08 02 17
         00 03 00 3E 00 0A 00 3E 00 0A 02 18
         00 04 00 3E 00 08 00 3E 00 0A 01 28
         00 05 00 3E 00 04 00 3E 00 0A 01 26
         00 06 00 3F 00 08 00 3F 00 08 02 19
         00 07 00 3F 00 0A 00 3F 00 0A 02 1A
         00 08 00 3F 00 08 00 3F 00 0A 01 2E
         00 09 00 3F 00 04 00 3F 00 0A 01 2C
         00 0A 00 40 00 08 00 40 00 08 02 1B
         00 0B 00 40 00 0A 00 40 00 0A 02 1C
         00 0C 00 40 00 08 00 40 00 0A 01 34
         00 0D 00 40 00 04 00 40 00 0A 01 32
         00 0E 00 41 00 08 00 41 00 08 02 1D
         00 0F 00 41 00 0D 00 41 00 0D 02 1E
         00 10 00 41 00 08 00 41 00 0D 01 3A
         00 11 00 41 00 04 00 41 00 0D 01 38
         00 12 00 43 00 18 00 43 00 18 02 1F
         00 13 00 43 00 0F 00 43 00 18 01 45
         00 14 00 43 00 0B 00 43 00 18 01 42
         00 15 00 3C 00 03 00 44 00 03 01 11
```

```
PositionTable:
-----------------------------------------------
PC: 0    Start: 61-12    End: 61-12    ID: 534
PC: 1    Start: 61-8     End: 61-12    ID: 288
PC: 2    Start: 62-8     End: 62-8     ID: 535
PC: 3    Start: 62-10    End: 62-10    ID: 536
PC: 4    Start: 62-8     End: 62-10    ID: 296
PC: 5    Start: 62-4     End: 62-10    ID: 294
PC: 6    Start: 63-8     End: 63-8     ID: 537
PC: 7    Start: 63-10    End: 63-10    ID: 538
PC: 8    Start: 63-8     End: 63-10    ID: 302
PC: 9    Start: 63-4     End: 63-10    ID: 300
PC: 10   Start: 64-8     End: 64-8     ID: 539
PC: 11   Start: 64-10    End: 64-10    ID: 540
PC: 12   Start: 64-8     End: 64-10    ID: 308
PC: 13   Start: 64-4     End: 64-10    ID: 306
PC: 14   Start: 65-8     End: 65-8     ID: 541
PC: 15   Start: 65-13    End: 65-13    ID: 542
PC: 16   Start: 65-8     End: 65-13    ID: 314
PC: 17   Start: 65-4     End: 65-13    ID: 312
PC: 18   Start: 67-24    End: 67-24    ID: 543
PC: 19   Start: 67-15    End: 67-24    ID: 325
PC: 20   Start: 67-11    End: 67-24    ID: 322
PC: 21   Start: 60-3     End: 68-3     ID: 273
```

Figure 11. On the left, the raw bytes javap produces. On the right, the structured information the testing tool produces

The tool also allows the user to print it in an orderly fashion, see the percentage of coverage or see which bytecode instructions are missing from the mapping.
The percentage of coverage is calculated simply by taking the number of bytecode instructions mapped and dividing it by the total number of bytecode instructions.

10.5 J15, J17, J18

Having done this, I tried to check if the framework I extended covered other Java versions as well, so I created several files with features whose mapping I wanted to check:

Version	Features added
Java 1.5	enums, enhanced for loops, generics, autoboxing
Java 1.7	diamond notation, binary numbers, underscore numbers, switch statements with strings
Java 1.8	lambda expressions

10.6 Coverage

To my delight, the mapping seemed to hold even for new features:

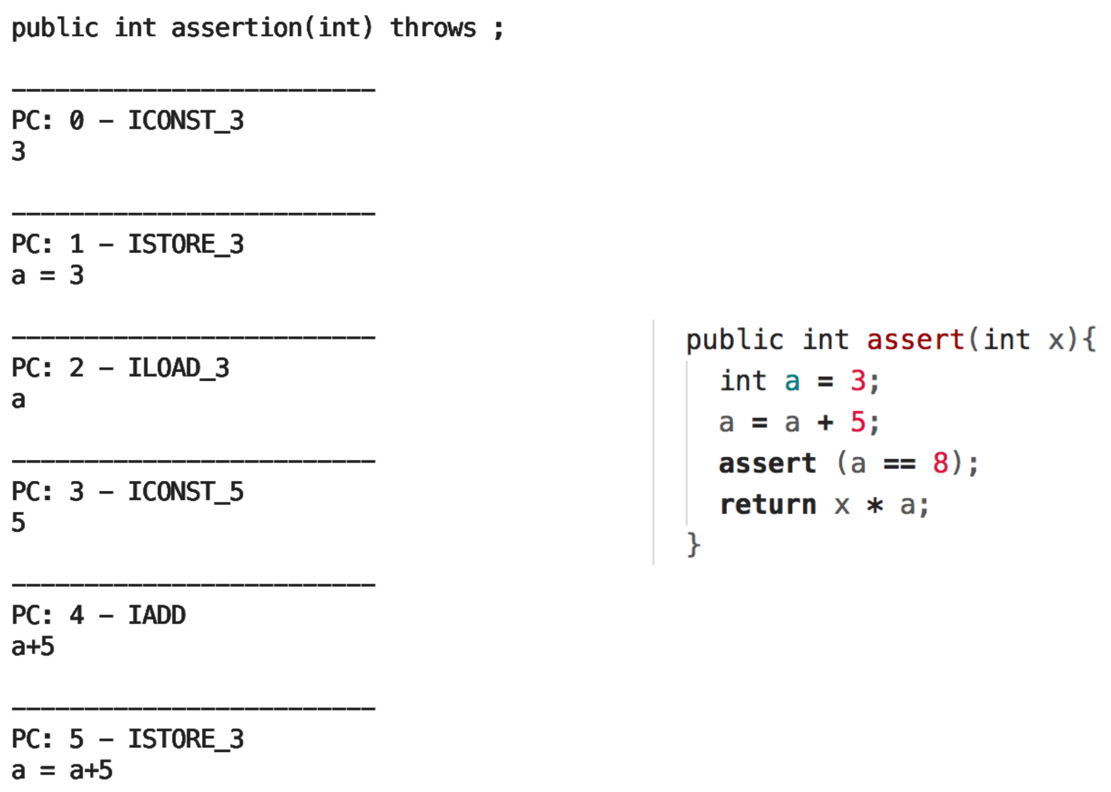
Version	% of coverage	Bytecodes mapped/total
Java 1.4	91.19	145/159
Java 1.5	91.32	169/185
Java 1.7	92.66	202/218
Java 1.8	90.22	203/225

Roughly, the number of instructions that remains unmapped remains constant.

That is because there is one corner case given by the assert keyword; ExtendJ takes it and translates it into source code, bypassing the emit methods.
That accounts for 6 bytecode instructions that are missing.
Another 6 are missing due to the static initializer, meaning a piece of code that gets run the first time the class gets loaded – for example, in order to enable assertions.

10.7 Simple step-by-step code displayer

Thanks to this project, I was able to create a simple command line application that allows for bytecode-per-bytecode displaying of the relevant chunk of source code.



```
public int assertion(int) throws ;

___________________________
PC: 0 - ICONST_3
3

___________________________
PC: 1 - ISTORE_3
a = 3

___________________________
PC: 2 - ILOAD_3
a

___________________________
PC: 3 - ICONST_5
5

___________________________
PC: 4 - IADD
a+5

___________________________
PC: 5 - ISTORE_3
a = a+5
```

```
public int assert(int x){
  int a = 3;
  a = a + 5;
  assert (a == 8);
  return x * a;
}
```

Figure 12. Left: the stepper built. Right: the corresponding source code

11 Conclusion

Through this project I gained a deeper understanding of how bytecode generation works and of the underlying structure of Java.
While the end result doesn't produce a mapping for every bytecode emitted, it comes pretty close to doing so; I believe this project could be used as a basis to develop debugging tools or software analysis tools.
The final project can be found on GitHub – see Links section.

12 Links

Libraries	IDEs	Other
ANT 1.9.4	Eclipse Oxygen.3a (4.7.3a) Release	ExtendJ (general version)
Jastadd 1.10.4	Netbeans IDE 8.2	ExtendJ (2017-04-06 version)
Gradle 3.4.1	IntelliJ IDEA 2018.1.4	Final version of the project

13 Related works

- Decompilation of binary programs[9]: the authors start from a binary program and find a way to reconstruct a high-level source code that, when compiled, functionally does the same thing as the binary.
- Decompiling Boolean Expressions from Java Bytecode[10]: the authors try to find a way starting from bytecode to obtain the original structure of a boolean expression, since the compilation obfuscates it.

Both works start from bytecode and try to obtain some information of the source code, which is quite similar to what we did.

References

[1] Various contributors. Abstract Syntax Tree (Wikipedia). `https://en.wikipedia.org/wiki/Abstract_syntax_tree`, 2018. [Online; accessed 21-May-2018].

[2] Erik Hogeman Torbjörn Ekman Jesper Öqvist, Emma Söderberg. ExtendJ. `https://extendj.org/index.html`, 2018. [Online; accessed 09-May-2018].

[3] Erik Hogeman Torbjörn Ekman Jesper Öqvist, Emma Söderberg. Splash 2015 tutorial slides. `https://bitbucket.org/extendj/extendj/downloads/2015%20SPLASH%20Tutorial%20ExtendJ.pdf`, 2015. [Online; accessed 09-May-2018].

[4] Görel Hedin. Abstract syntax. `http://jastadd.org/old/manual/abstract-syntax.php`, 2010. [Online; accessed 09-May-2018].

[5] Jesper Öqvist Emma Söderberg Görel Hedin, Niklas Fors. Attributes. `http://jastadd.org/old/manual/attributes.php`, 2010. [Online; accessed 09-May-2018].

[6] Various contributors. Java .class file (Wikipedia). `https://en.wikipedia.org/wiki/Java_class_file`, 2018. [Online; accessed 16-June-2018].

[7] Various contributors. Bytecode (Wikipedia). `https://en.wikipedia.org/wiki/Java_bytecode`, 2018. [Online; accessed 21-May-2018].

[8] Alex Buckley Tim Lindholm Frank Yellin, Gilad Bracha. Java Language Specifications. `https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.12`, 2015. [Online; accessed 09-May-2018].

[9] K. John Gough Cristina Cifuentes. Decompilation of Binary Programs. 1995.

[10] Mangala Gowri Nanda Mangala Gowri Nanda. Decompiling Boolean Expressions from Java Bytecode. 2016.

14 Addendum: Research notes

14.1 Research phase

In this phase, the aim was to get an initial understanding of ExtendJ and JastAdd, producing as an end result a minimal mapping between ASTNodes and source code lines.
I started by playing around with ExtendJ and tried to store information in the ASTNodes, specifically encoding start and end column of a source line.
I eventually found out that the information was already implicitly present in the nodes, since an ASTNode is a Symbol, and Symbols have a getStart() and getEnd() method which retrieve the start and end positions, respectively; those contain both line and column number together, so if one wishes to access for example the starting line, it would be getLine(getStart()), or for the column getColumn(getStart()).
At this point, I created a jrag aspect (an extension for ExtendJ), in which I retrieved those positions.
I also found out that empty Opt blocks or empty lists are marked as Line 0 Column 0 for both start and end, making it easy to identify them.
Having done this, I quickly realised there were some inconsistencies in the boundaries (e.g. a Node starting at 5 and ending at 2); that is because some nodes start and end at different line. I went on to modify the jrag file accordingly, ending up with start_line, start_column, end_line, end_column.

14.2 Code generation phase

In this phase, the aim was to extend minimally the code generation so to include inside a .class file the position information of the ASTNode that emitted a certain bytecode.
So, I went on and studied how the bytecode generation was done. I found a pre-existing attribute of a class file serving a similar purpose to what I planned to implement, LineNumberTable.
It maps a source code line to a specific bytecode instruction (i.e. it maps from PC 0 to PC 4 to source code line 1, from PC 5 to PC 9 to source code line 2,...).
I added a PositionTableAttribute in PositionTable.jrag, planning to store PC, start_line, start_column, end_line, end_column information inside it.
ASTNode.codeAttributes had to be modified as well (CodeGeneration.jrag), since I had to add the new attribute inside the list.
CodeGeneration.java was modified so to include an ArrayList of PositionEntry (each one being a container for the 5 aforementioned attributes), and a addPositionEntryAtCurrentPC method.
Once that was done, it was time to find where to place the calls to this method so to get maximal coverage for the mapping.
There were two choices: either to place them in the CreateBCode methods, or to place them in the emit methods.
I decided to go for the latter, since it's the methods in which bytecode instructions actually get emitted, thus pretty much guaranteeing maximal coverage.
After having created a test .java file (J14.java, described later), I proceeded to map the ISTORE, ILOAD, IRETURN, ICONST bytecode instructions through the method described above.
This resulted in a coverage of around 54% of all bytecodes emitted for J14.java.
I then found out that Mohammad, a PhD student developing a similar tool as mine, managed to already tweak all methods and get a very high percentage of coverage, 92%.
Not wanting to spend the rest of my time on the rather repetitive task of looking at stack traces, identifying which method get called and modifying them to add the mapping, I decided to use his version of the code, adapting it to my needs, so to be able to focus on more challenging parts of the project.

14.3 AST to JSON

The next step was to take the AST, parse it into an ordered JSON structure, and label each node, so to be able to map each entry in a position table to a specific node in the AST.
The initial idea was to write the JSON directly into the constant pool, however that proved to be a problem due to limitations of the library used (the maximum amount writable would be 32k bytes).
Thus, I resorted to writing to an external file.

14.4 Final stretch

With the time remaining, I extended the tests from Java 1.4 all the way to Java 1.8.
While I was at it, I developed a small terminal program to visualise what this mapping could actually produce.

15 Addendum: Testing files

The following code is what I used to check the coverage.

15.1 J14.java

```
import java.util.*;

interface Language {
  public float getVersion();
}

abstract class Java implements Language{
  protected float version;

  public float getVersion() {
    return version;
  }
}

class J14 extends Java{
    private static String name;

    public J14(String name){
        this.name = name;
    }

    public int assertion(int x) {
        int a = 3;
        a = a+5;
        assert ( a == 8 );
        return x * a;
    }

    private class InnerClass {
      private int value;
      public InnerClass(int x){
        this.value = x;
      }
    }

    public int loops() {
      int counter = 0;
      for (int i = 0; i <= 10; i++){
        ++counter;
      }

      while (counter > 5){
        counter--;
        continue;
      }

      do {
        counter++;
      } while (counter < 10);
      --counter;

      counter+=1;
      counter-=1;
      counter*=1;
      counter/=1;
```

```
        return counter;
      }

      public void bitwise(){
        int a = 5;
        a = a^3;
        a = a|3;
        a = a&3;
        a = a << 3;
        double b = (double) a;
      }


      public boolean ifStatements(){

        char a = 'a';

        switch(a){
          case 'b':
            a = 'c';
            break;
          default:
            break;
        }

        InnerClass ic = null;
        ic = new InnerClass(3);

        if(false && false && ic != null){
          return true? false : true;
        } else if(true || false && a >= 3){
          return true;
        } else {
          return false;
        }
      }


      public void exceptionThrower() throws Exception{
        int [] a = new int[2];
        int [] b = {1, 2, 3};
        int [] c;
        try {
          a[4] = 4;
        } catch(IndexOutOfBoundsException e){
          throw new Exception("A chained Exception has been thrown.");
        }
      }
  }
```

15.2 J15.java

```
import java.util.*;

interface Language {
  public float getVersion();
}

abstract class Java implements Language{
  protected float version;

  public float getVersion() {
    return version;
  }
}

enum TypeOfLanguage
{
  FUNCTIONAL, IMPERATIVE
}

class J15 extends Java{
    private static String name;
    private TypeOfLanguage type;

    public J15(String name){
        this.name = name;
        this.type = TypeOfLanguage.IMPERATIVE;
    }

    public int assertion(int x) {
        int a = 3;
        a = a+5;
        assert ( a == 8 );
        return x * a;
    }

    private class InnerClass {
      private int value;
      public InnerClass(int x){
        this.value = x;
      }
    }

    public int loops(int... items) {
      int counter = 0;
      for (int i = 0; i <= 10; i++){
        ++counter;
      }

      while (counter > 5){
        counter--;
        continue;
      }

      do {
        counter++;
      } while (counter < 10);
      --counter;

      counter+=1;
      counter-=1;
      counter*=1;
      counter/=1;

      ArrayList<Integer> m = new ArrayList<Integer>();

      for (int x : items){
        m.add(x);
      }
      return counter;
    }

    public void bitwise(){
      int a = 5;
      a = a^3;
      a = a|3;
      a = a&3;
      a = a << 3;
      double b = (double) a;
    }


    public boolean ifStatements(){
      char a = 'a';

      switch(a){
        case 'b':
          a = 'c';
          break;
        default:
          break;
      }

      InnerClass ic = null;
      ic = new InnerClass(3);

      if(false && false && ic != null){
        return true? false : true;
      } else if(true || false && a >= 3){
        return true;
      } else {
        return false;
      }
    }

    public void exceptionThrower() throws Exception{
      int [] a = new int[2];
      int [] b = {1, 2, 3};
      int [] c;
      try {
        a[4] = 4;
      } catch(IndexOutOfBoundsException e){
        throw new Exception("A chained Exception has been thrown.");
      }
    }
}
```

15.3 J17.java

```java
import java.util.*;

interface Language {
  public float getVersion();
}

abstract class Java implements Language{
  protected float version;

  public float getVersion() {
    return version;
  }
}

enum TypeOfLanguage
{
  FUNCTIONAL, IMPERATIVE
}

class J17 extends Java{
    private static String name;
    private TypeOfLanguage type;

    public J17(String name){
        this.name = name;
        this.type = TypeOfLanguage.IMPERATIVE;
    }

    public int assertion(int x) {
        int a = 3;
        a = a+5;
        assert ( a == 8 );
        return x * a;
    }

    private class InnerClass {
      private int value;
      public InnerClass(int x){
        this.value = x;
      }
    }

    public int loops(int... items) {
      int counter = 0;
      for (int i = 0; i <= 10; i++){
        ++counter;
      }

      while (counter > 5){
        counter--;
        continue;
      }

      do {
        counter++;
      } while (counter < 10);
      --counter;

      counter+=1;
      counter-=1;
      counter*=1;
      counter/=1;

      ArrayList<Integer> m = new ArrayList<>();

      for (int x : items){
        m.add(x);
      }

      return counter;
    }

    public void bitwise(){
      int a = 5_5;
      a = a^3;
      a = a|3;
      a = a&3;
      a = a << 3;
      a = 0b0010001;

      double b = (double) a;
    }


    public boolean ifStatements(){
      String s = "hello";

      switch(s){
        case "hello":
          s = "world";
          break;
        default:
          break;
      }

      Integer a = 3;
      InnerClass ic = null;
      ic = new InnerClass(3);

      if(false && false && ic != null){
        return true? false : true;
      } else if(true || false && a >= 3){
        return true;
      } else {
        return false;
      }
    }

    public void exceptionThrower() throws Exception{
      int [] a = new int[2];
      int [] b = {1, 2, 3};
      int [] c;
      try {
        a[4] = 4;
      } catch(IndexOutOfBoundsException e){
        throw new Exception("A chained Exception has been thrown.");
      } catch (Exception e){
        a[0] = 1;
      }
      finally {
        a[0] = 2;
      }
    }
}
```

15.4 J18.java

```
import java.util.*;

interface Language {
  public float getVersion();
}

abstract class Java implements Language{
  protected float version;

  public float getVersion() {
    return version;
  }
  public static void doNothing(){
    return;
  }
}

enum TypeOfLanguage
{
  FUNCTIONAL, IMPERATIVE
}

class J18 extends Java{
    private static String name;
    private TypeOfLanguage type;

    public J18(String name){
        this.name = name;
        this.type = TypeOfLanguage.IMPERATIVE;
    }

    public int assertion(int x) {
        int a = 3;
        a = a+5;
        assert ( a == 8 );
        return x * a;
    }

    private class InnerClass {
      private int value;
      public InnerClass(int x){
        this.value = x;
      }
    }

    public int loops(int... items) {
      int counter = 0;
      for (int i = 0; i <= 10; i++){
        ++counter;
      }

      while (counter > 5){
        counter--;
        continue;
      }
      do {
        counter++;
      } while (counter < 10);
      --counter;
      counter+=1;
      counter-=1;
      counter*=1;
      counter/=1;

      ArrayList<Integer> m = new ArrayList<>();

      for (int x : items){
        m.add(x);
      }
      m.forEach(x -> System.out.println(x));

      return counter;
    }

    public void bitwise(){
      int a = 5_5;
      a = a^3;
      a = a|3;
      a = a&3;
      a = a << 3;
      a = 0b0010001;
      double b = (double) a;
    }


    public boolean ifStatements(){
      String s = "hello";

      switch(s){
        case "hello":
          s = "world";
          break;
        default:
          break;
      }

      InnerClass ic = null;
      ic = new InnerClass(3);
      Integer a = 3;

      if(false && false && ic != null){
        return true? false : true;
      } else if(true || false && a >= 3){
        return true;
      } else {
        return false;
      }
    }

    public void exceptionThrower() throws Exception{
      int [] a = new int[2];
      int [] b = {1, 2, 3};
      int [] c;
      try {
        a[4] = 4;
      } catch(IndexOutOfBoundsException e){
        throw new Exception("A chained Exception has been thrown.");
      } catch (Exception e){
        a[0] = 1;
      } finally {
        a[0] = 3;
      }
    }
}
```

16 Addendum: List of AST Nodes

ASTNode	ASTNode.State	ASTNodeAnnotation
AbstractDot	AbstractWildcard	AbstractWildcardType
AddExpr	AdditiveExpr	AmbiguousMethodReference
AndLogicalExpr	AnnotatedCompilationUnit	Annotation
AnnotationMethodDecl	AnonymousDecl	ArithmeticExpr
ArrayCreationExpr	ArrayDecl	ArrayInit
ArrayTypeAccess	ArrayTypeWithSizeAccess	AssertStmt
AssignAndExpr	AssignBitwiseExpr	AssignDivExpr
AssignLShiftExpr	AssignMinusExpr	AssignModExpr
AssignMultiplicativeExpr	AssignOrExpr	AssignPlusExpr
AssignShiftExpr	AssignSimpleExpr	AssignURShiftExpr
Attributes	BasicCatch	Binary
BitwiseExpr	Block	BlockLambdaBody
BodyDeclList	BooleanLiteral	BooleanType
BoundMethodAccess	BoundTypeAccess	BranchTargetStmt
BridgeMethodDecl	ByteType	BytecodeClassSource
BytecodeReader	BytecodeTypeAccess	CONSTANTClassInfo
CONSTANTFieldrefInfo	CONSTANTFloatInfo	CONSTANTInfo
CONSTANTInterfaceMethodrefInfo	CONSTANTLongInfo	CONSTANTMethodrefInfo
CONSTANTStringInfo	CONSTANTUtf8Info	Case
CatchClause	CatchParameterDeclaration	CharType
ClassAccess	ClassDecl	ClassDeclSubstituted
ClassInstanceExpr	ClassPath	ClassReference
CompilationUnit	ConditionalExpr	ConstCase
Constraints	ConstructorAccess	ConstructorDecl
ConstructorReference	ConstructorReferenceAccess	ContinueStmt
DefaultCase	DiamondAccess	Dims
DoStmt	Dot	DoubleLiteral
EQExpr	ElementAnnotationValue	ElementArrayValue
ElementValue	ElementValuePair	EmptyStmt
EnhancedForStmt	EnumConstant	EnumDecl
EqualityExpr	Expr	ExprLambdaBody
ExprStmt	FieldDecl	FieldDeclaration
FieldDescriptor	FieldInfo	FileBytecodeClassSource
FinallyHost	Flags	FloatType
FloatingPointType	FolderPath	ForStmt
FunctionDescriptor	GEExpr	GLBType
GTExpr	GenericClassDecl	GenericClassDeclSubstituted
GenericInterfaceDecl	GenericInterfaceDeclSubstituted	GenericMethodDecl
IdUse	IfStmt	IllegalLiteral
InferredLambdaParameters	InferredParameterDeclaration	InstanceInitializer
IntType	IntegerLiteral	IntegralType
InterfaceDeclSubstituted	IntersectionCastExpr	JarClassSource
JavaParser	LEExpr	LShiftExpr
LUBType	LabeledStmt	LambdaAnonymousDecl
LambdaExpr	LambdaParameters	LazyFinallyIterator
Literal	LocalClassDeclStmt	LogNotExpr
LongLiteral	LongType	MemberClassDecl

MemberInterfaceDecl	MemberSubstitutor	MemberTypeDecl
MethodDecl	MethodDeclSubstituted	MethodDescriptor
MethodReference	MethodReferenceAccess	MinusExpr
Modifier	Modifiers	MonitorExit
MultiCatch	MultiplicativeExpr	NEExpr
NameType	NullLiteral	NullType
NumericType	Opt	Options
OrLogicalExpr	PackageAccess	ParClassDecl
ParConstructorAccess	ParConstructorDecl	ParConstructorReferenceAccess
ParInterfaceDecl	ParMethodAccess	ParMethodDecl
ParSuperConstructorAccess	ParTypeAccess	ParTypeDecl
ParameterDeclarationSubstituted	Parameterization	ParseName
PlusExpr	PostDecExpr	PostIncExpr
PreDecExpr	PreIncExpr	PrimaryExpr
PrimitiveType	PrimitiveTypeAccess	Problem
RShiftExpr	RawClassDecl	RawConstructorDecl
RawMethodDecl	ReferenceType	RelationalExpr
ResourceModifiers	ReturnStmt	ShiftExpr
Signatures	SimpleParameterization	SimpleSet
SingleTypeImportDecl	SourceFilePath	SourceFolderPath
StaticImportDecl	StaticImportOnDemandDecl	StaticInitializer
StringLiteral	SubExpr	SuperAccess
SwitchStmt	SynchronizedStmt	SyntheticTypeAccess
ThrowStmt	TryStmt	TryWithResources
TypeDecl	TypeDescriptor	TypeImportOnDemandDecl
TypeVariable	URShiftExpr	Unary
VarAccess	VarDeclStmt	Variable
VariableArityParameterDeclarationSubstituted	VariableDecl	VariableDeclaration
VoidType	WhileStmt	Wildcard
WildcardExtendsType	WildcardSuper	WildcardSuperType
WildcardsCompilationUnit	XorBitwiseExpr	