MASTER'S THESIS | LUND UNIVERSITY 2016

DrAST - An attribute debugger for JastAdd

Joel Lindholm, Johan Thorsberg

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-10

DrAST - An attribute debugger for JastAdd

(A new solution for debugging AST-based compilers)

Joel Lindholm
joel.lindholm@gmail.com

Johan Thorsberg
johanthorsberg@gmail.com

April 1, 2016

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Görel Hedin, gorel.hedin@cs.lth.se

Examiner: Ulf Asklund, ulf.asklund@cs.lth.se

Abstract

Here we present a solution for debugging compilers that use abstract-syntax trees as their internal structure. The solution focuses on capturing one specific state of the compilation process, and should not be confused with the more known step-by-step debugging. The goal is to visualize the current state of the abstract-syntax tree and present its data to the user in an intuitive and interactive way. We believe that deeper understanding of an abstract-syntax tree, and bugs in its structure, can be achieved by visualization of the tree. Few such debuggers exist today however, but with this master thesis we aim to fill this gap.

The main feature of the developed tool DrAST is the ability to visualize the abstract-syntax tree. It is also possible to filter the tree, so that only nodes of interest are visualized, while the rest are gathered in what we call clusters. Further, DrAST can display attributes, draw references between nodes, calculate parameterized attributes and is built for further extension.

DrAST mainly debugs compilers created in the attribute-grammar-based system JastAdd. By the use of Java reflection and annotations from the JastAdd system, the debugger is able to extract the abstract-syntax tree from a compiler without knowing the specific grammar.

In short, DrAST provides a new solution in compiler debugging which can be of use for both students and professionals.

Keywords: debugger, compiler, reflection, Java, JastAdd, attribute grammar, visualization, interactive

Acknowledgements

We would like to thank our supervisor Görel Hedin, for supporting us in so many ways during the whole project. We would also like to direct our thanks to Niklas Fors for making important changes to the JastAdd system in order to make DrAST smarter.

A special thanks goes to Jennie Örjes, who has no previously experience of the computer sciences field but still helped us immensely with writing this report with insightful discussions and comments.

Lastly a special special thanks to the coffee machine in the building, who provided us with a total of 116 coffees, 103 tees and 33 hot chocolates.

Contents

Chapter 1
Introduction

This chapter will introduce and summarize different parts of this report, with a focus on the problem, method and result of the project.

1.1 Problem description

Today a lot of programming languages are being developed. These include not only the General-Purpose Languages (GPLs) like Java and python, but also a lot of domain-specific languages (DSLs). A certain DSL generally only work for a specific, narrowed down domain. Developing a DSL can be challenging and time consuming. This is where debugging tools can assist the developer.

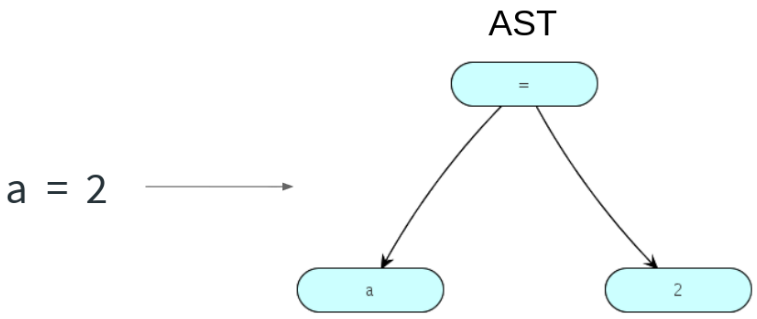


Figure 1.1: The code `a=2` can be described as an AST. The Binding (=) will point at the variable (*a*) and the value (*2*).

When compiling any kind of code, the code needs to be represented in a data structure. A widely used structure is the abstract syntax tree (AST). It is basically a tree structure where every node is a word or a token, parsed from some code being compiled. Figure 1.1 illustrates how the code `a=2` is represented in an AST. The Binding (=) has two children, the variable (*a*) and the value (*2*). Figure 1.2 shows an overview of how code that is compiled will result in an AST. This figure will be extended with more details throughout the report. The main subject of this report includes how to represent this type of structures in a meaningful way to help the developer.

During the compiling process of a program, there is a phase called the semantic analysis that controls if the constructed syntax structure follows the laws of the language and to see if the structure has any real meaning. This usually includes type checking and scope analysis, for example single declaration of a variable. Attribute grammars [16] is a method that is useful in this phase. Attribute grammars are a way of adding attributes to nodes in the AST, and describe how they are calculated. Adding attributes to the AST nodes also enables different kinds of analyses, in order to find both errors in and other interesting behaviours of a program. Examples of questions that can be answered with the help of attribute grammars are whether there are several nodes with the same ID and how many children a certain node type has.

However, to our knowledge, today there are few attribute grammar debuggers for AST implementations like the one described above. One could think of many features in such a debugger, but the main feature would be to visualize the AST and its current attribute values. However, developers are today usually forced to find their own solution to visualize how AST nodes and their attributes are connected. This could include writing on paper or writing print methods for the AST nodes, which can become cumbersome when a project grows. Also, to detect different bugs in the language structure the developer needs to write extensive test suites for the attributes. Another problem with the lack of user friendly interactive visualization tools is introducing someone new to a project with an extensive AST. This can be a bit overwhelming at first, where there might not be a quick way to get an overview of the project's AST structure.

Especially projects with large language structures can suffer from this. An example of this is the java compiler ExtendJ [9], which currently contains about 250 - 260 node types. A Java file, with 350 lines of code, compiled with ExtendJ can result in an AST with 650 nodes, or 61 000 if we include its library files.

All the above described issues would be less tiresome and time consuming for developers if there was a way to easily visualize the resulting AST, with attributes and their values of every node.

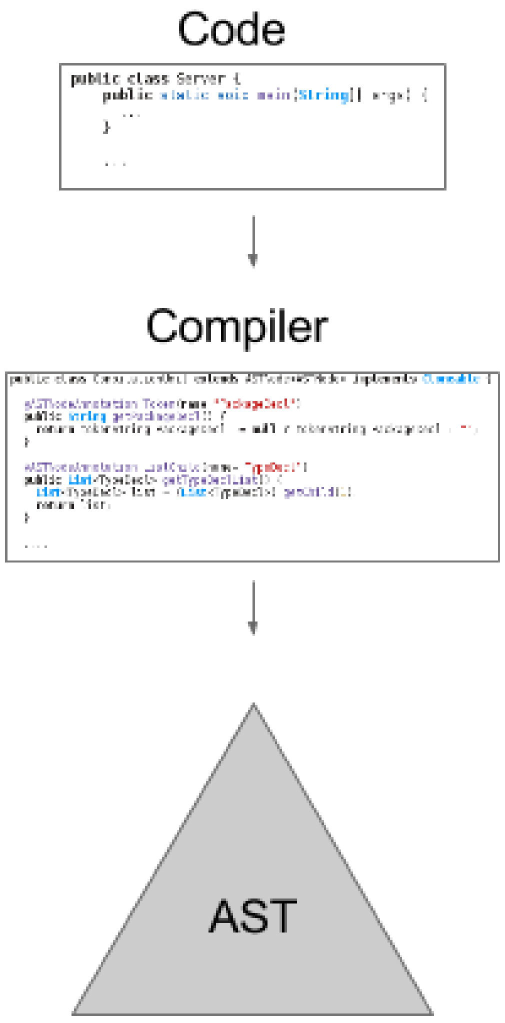
1.2 Project goals and challenges

Here we will present and explain the goals and challenges that we need to fulfil and overcome in this master thesis.

1.2.1 Goals

The goal of this master thesis was to find a solution for debugging compilers, proposedly by creating a new interactive-visual-attribute debugger for ASTs and analyse the benefits of such a tool. The proposed solution should be able to visualize ASTs from compilers generated with the attribute-grammar-based system JastAdd [11].

Usability The tool should have a high usability. The user base for the tool consists of new and inexperienced users in JastAdd like students in compiler courses, and more



```
public class Server {
    public static void main(String[] args) {
        ...
    }

    ...
```

```
public class CompilationUnit extends ASTNode<ASTNode> implements Cloneable {

  @ASTNodeAnnotation.Token(name="PackageDecl")
  public String getPackageDecl() {
    return tokenString_PackageDecl != null ? tokenString_PackageDecl : "";
  }

  @ASTNodeAnnotation.ListChild(name="TypeDecl")
  public List<TypeDecl> getTypeDeclList() {
    List<TypeDecl> list = (List<TypeDecl>) getChild(0);
    return list;
  }

  ....
```

Figure 1.2: An illustration of how a compiler creates a AST from some code.

advanced users such as researchers and companies. This makes the usability a vital feature.

Performance The tool needs to be stable and execute its computations within an acceptable time frame. This needs to be true for both small test projects and extensive research or industry projects

Scalability The tool needs a way to present the AST in an intuitive way. The AST can contain a great number of attributes and nodes, and the developer is not always interested in the whole graph. A way to highlight or show specific parts of the AST is therefore needed.

1.2.2 Challenges

The following challenges were recognized within the project, in order to create the desired debugger tool and fulfill the goals.

Features Identification of features that is important for the compiler developer.

Prioritizing the features Finding a way to prioritize the features that could be implemented in to the debugger is needed, due to that the scope of possible features is vast.

Platform Choosing the right platform and libraries to build this tool upon.

Method for implementation Finding the most suitable method for extracting the data needed for the debugger.

Evaluation of usability and performance Evaluating the resulting tool with the goals defined in section 1.2.1 in mind.

The details of these challenges are further explained and discussed throughout the rest of the report.

1.3 Method

During the process of this project a number of different methods and resources was used, ranging from literature studies and discussions to pair programming [3] and usability tests. This section will describe what we did and why we did it. The methods used have been chosen based on literature relevant to this type of work [18, 27], practical uses of said methods, and experience from the authors and their supervisor.

In the end, an iterative process was decided upon. There are a lot of different agile methods, Scrum and XP to name a few [10, 3]. What they have in common is that the development is split into smaller iterations, sprints. We did not choose one specific method; instead we mixed parts that we were comfortable with. Each sprint contained a number of steps like a backlog meeting, implementation and testing. By choosing the next step carefully and regularly, we believe that a more stable tool was developed. At the beginning of an iteration, the decision to strengthen the quality of existing features or adding new ones could be made based on time and need.

1.3.1 Meeting and backlog

We decided upon a fixed sprint length. Each sprint was one week long, except for some exceptions. In the beginning of an iteration, we had a meeting with the supervisor and discussed the status of the project. The questions usually were if we should add new features or improve the existing ones. The two main factors to consider when we decided to implement a feature were; how critical the feature was and how time consuming was the implementation. We also added time for research and writing.

1.3.2 Implementation

During the implementation phase different techniques was used. Most implementation was done individually after discussions, but for the more challenging or extensive features pair programming [3] was used. §1

In order to keep track of the status of the project, and to be able to merge new features developed on different machines, a Git repository was used located on BitBucket, a software hosting service.

1.3.3 Testing

To have fixed parts of a sprint dedicated to testing did not fit into this project. Although testing was needed in order to decide what needed to be implemented or improved. If performance was bad we needed to optimize, otherwise add features. Usually when a feature was done, it was tested by the developer themselves. Usability testing was conducted at the last half of the development, as at that stage the graphical user interface was starting to take shape and needed to be tested to be approved upon.

Performance and correctness The goal was to make a tool that help developers that uses JastAdd. It is important that the tool performs well for projects of any size, small and large. Therefore the tool was tested on a set of programs with ASTs of different size and different number of attributes in the nodes. These tests evaluated the correctness of the representation of the ASTs, and tool's process time and memory use.

User experience As mentioned DrAST is supposed to be a tool for every developer and therefore the user experience, or to be more precise the usability, is important. We had to create a visual representation of huge sets of different types of data. In order to battle our own blindness for faults in our user interface, we let users with different JastAdd experience use DrAST under the rules of Think aloud [18] usability testing.

1.4 Result

1.4.1 Visualization debugger tool: DrAST

A new visualization debugger tool for JastAdd systems have been implemented, called Display reflected AST (DrAST) and is implemented in Java. The tool shows a representation of the AST that we call the filtered tree. It is possible to see the attributes, and their values, for each node in the tree. Parameterized attributes can be computed with user input. References between nodes can be displayed. The main features of DrAST are the following:

Filter language A new domain-specific language has been developed to tell DrAST which nodes that should be visible and not. It is possible to filter on node classes, node position in the tree and attributes in each node. The language is created to be simple and powerful by itself, but the developer could combine it with JastAdd-implemented attributes, specifically added for filtering. See chapter 4 for more information.

Reflected tree The reflected tree is the 1-to-1 representation of the AST in DrAST, where each node in the AST are represented by a container node in the reflected tree. In short each node in the reflected tree has a reference to a node in the AST. See chapter 5 for more information how this structure is created.

Filtered tree The tree structure displayed in DrAST does not necessarily represent the complete AST, but instead a filtered version of it. Only nodes and attributes that are interesting to the user is shown, the rest is collapsed into what we call cluster nodes. In other words, DrAST can collapse parts of a tree, not just a whole subtree. This makes it easier for the user to find the requested information and gives a good performance boost when applied on big ASTs. See chapter 4 for more information about the tool itself.

1.4.2 Implementation

The implementation is based on a Model-View-controller pattern [17]. The architecture is split into one model and any number of view-controllers, that we just call views. The model contains the reflected tree and the filtered tree as well as other data connected to the AST. Views are then used to present the filtered tree to the user. The interactive graph view is such an example, described in chapter 4. Other examples are printing the filtered tree to an XML file or storing it as an image file.

The Model uses a combination of Java reflection [6] and Java annotations, generated by JastAdd, to extract the AST data. With reflection, DrAST is able to create the reflected tree, which then is used to create the filtered tree. See chapter 5 for more information about the structure and implementation decisions.

1.4.3 Evaluation

The usability tests were not performed in a way so that any statistical assessments could be done. The tests were too few, and performed on different versions of DrAST. However, the tests gave us a pointer of how understandable the user interface and documentation was. The participants of the tests always completed the given tasks within an acceptable time frame. Many also expressed that they liked the features and thought the tool was useful. More on this in chapter 6.

DrAST performs well, and creating the model should take under 2 seconds even for large trees of around 200 000 nodes, on everyday machines. The graph visualization is however not able to handle that amount of nodes without slowing down, but still works with around 5 frames per second (FPS). We do believe that the user is usually interested in viewing a smaller amount of nodes, and this is why we use the filter. The graph-view's performance is based on the filtered tree and not the original AST, and by shrinking the size of the filtered tree we will gain a boost in FPS. See chapter 6.2 to get more information about the performance.

1.5 Contributions

The work for this thesis was divided quite equally between us. We have been involved in most parts of the development, either directly or via discussions. In broad terms though, Joel has been more involved with the development of the model and the reflection part while Johan has been working more on GUI view with the JavaFX design and Jung optimizations.

1.6 Report structure

The structure of this report is as follows. The first chapter gives the reader a general overview of what this project is about. We then go deeper into the different parts of the process, in chapter 2 we introduce some basic knowledge about JastAdd and compilers. We then describe the problem by giving an motivating example in chapter 3. Next is chapter 4, DrAST, in which we describe our solution to the problems. After that comes chapter 5, Implementation, which goes deeper into how the tool is built, for example the underlying structure and libraries used to create DrAST. We then come to the last chapters of the report. Evaluation, chapter 6, presents how DrAST was evaluated as well as the result of these evaluations. In Related work, chapter 7, we discuss alternative solutions and other tools. And lastly in Concluding discussion, chapter 8, we summarize the topics of the report and discuss future work.

Chapter 2

Technical background

This chapter introduces some basic concepts in order to make the rest of the report more understandable. A short summary of the compiling process and a small introduction to JastAdd will be described.

2.1 The compiling process

Compiling code is the process of translating code into something the domain can understand. This process can be divided into two parts; analysis and synthesis [2]. However, synthesis is not always needed depending on the domain. In this project we are focused on the analysis part. The analysis consists of reading and analyzing the code to ensure that it follows the rules of the language. It can further be divided in to three steps: lexical, syntactic and semantic (Figure 2.1) [2].

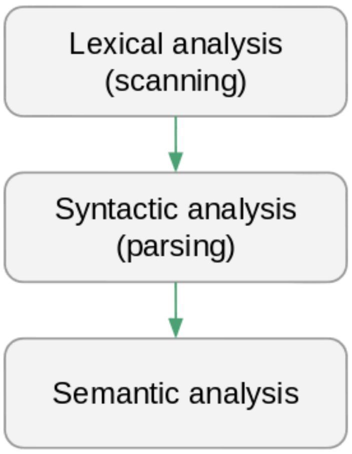


Figure 2.1: An overview of the analysis in the compiling process.

The lexical analysis converts the code (a char sequence) into a sequence of tokens. A token is defined by a specific sequence of characters, some examples could be `if` and `while`. This step is often called scanning.

The next step is the syntactic part of analysis. Here the compiler control that the tokens identified during scanning comes in the right order relative to each other. As an example the end of an expression might need to be followed by a semicolon token, or an equal token must be followed by a digit token. If the order does not make sense there is a syntactic error.

The last step of the analysis process is the semantic part, where the remaining rules of the language are checked. The language might require some type checking, or that it only accepts single declaration of variables.

Some compilers use an AST as an internal structure (ExtendJ [9] and CalcASM, described in chapter 3, to name a few) and this AST will be complete after the semantic analysis. DrAST, the tool described in this report, visualize this AST.

The synthesis part of the compiling process will not be described in detail. In short it can be mentioned that this is when machine code is generated and sometimes optimized. Generating machine code is though not always needed. Many programing languages are domain specific and the resulting data after the analysis (the AST as an example) is all that is needed.

2.2 JastAdd

JastAdd [11] is a system for generating compilers and different analytics tools, and is based on attribute grammars [16] and Java. As of writing this report JastAdd is developed and maintained by the JastAdd Team at the Faculty of Engineering at Lund University in Sweden. JastAdd uses an internal structure of an AST, where the nodes in the AST can hold a number of attributes. The idea with the system is that it should be easy to add attributes and nodes to the AST. The developer can simply "just add" attributes and nodes to the AST. This in turn makes it easy to extend languages and tools using JastAdd, in a modular way [11].

We will in chapter 3, Motivating example, see how the node types and attributes are defined by the developer in JastAdd. JastAdd generates Java classes, one class for each node type. The attributes become methods in their specific classes. When compiling some input code, any Java parser can be used together with the JastAdd-generated classes to define the AST for the code.

Chapter 3
Motivating example

This chapter is here to motivate and explain the complexity of an AST, and why a visual representation is needed. This chapter will also introduce abstract and attribute grammars in JastAdd and how this relates to the AST structure. For this we will use a small example language named CalcASM, and explain its abstract grammar and show how attributes can be added to the AST with JastAdd.

3.1 Overview of a JastAdd compiler

In this section we will illustrate an overview of how a JastAdd compiler is created by extending the Figure from chapter 1 (Figure 1.2) into Figure 3.1. The first thing the developer does is to define something called a abstract grammar, which are rules that define the structure of the AST. This grammar will then be read by the JastAdd system, which in turn generates Java classes which together makes up a compiler. Each type defined in the grammar will be represented as its own Java class, and child nodes and attributes are represented as methods in these classes. When code is compiled by the compiler, an AST will be created at some point during the compiler process.

3.2 CalcASM

CalcASM is a small example language that will help to illustrate how the abstract grammar and attributes are used during compilation to create an AST.

An example of a CalcASM program is shown in Figure 3.2. A CalcASM expression can either consist of one computation or one let-in-end block, like in the figure. In the example the variables `a` and `b` are defined in the let-block and are then used in the in-block.

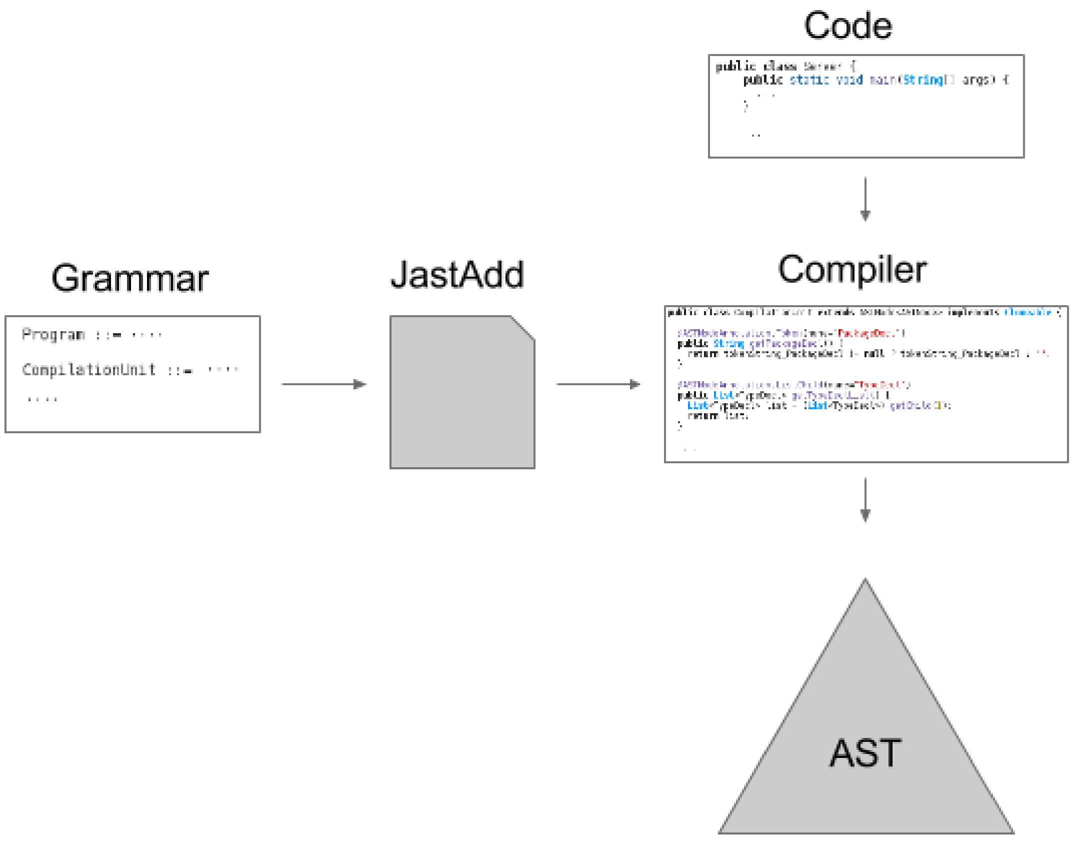


Figure 3.1: An overview of how a JastAdd compiler is generated and in turn how some code will result in an AST. The arrows indicate order.

3.3 Abstract grammar for CalcASM

To build an AST the abstract grammar for CalcASM first needs to be defined. The grammar describes the classes of the language constructs, and how they should fit together. This also describes the structure of the AST. Figure 3.3 shows the JastAdd abstract grammar for the CalcASM language. To the left of the `::=` is the name of a class and to right are its children.

A tree structure needs a root node, and in this case it is of the class `Program` and it has one child, an `Expr`. Similar to object-oriented languages one can define abstract classes, although in JastAdd these classes represent nodes. `Expr` is such a node class, and is defined as an abstract class in the grammar by the prefix `abstract`. Inheritance is marked by a `:` to the right of the class name. The following node classes all inherit the `Expr` class: `Mul`, `Numeral`, `IdUse`, `Let` and `Ask`. This means that the `Program` node can point to any of these classes as its child node.

In the grammar these child nodes all inherit the `Expr` class as explained above, but otherwise they differ. For example are`Mul` nodes defined to have two children, both of the type `Expr`. Another example of the grammar is that `Let` nodes are defined to have a list

```
let
    a = 4
    b = 2
in
    a / b
end
```

Figure 3.2: A small code example of the CalcASM language.

```
Program ::= Expr;
abstract Expr;
Mul : Expr ::= Expr Expr;
Numeral : Expr ::= <NUMERAL>
IdUse : Expr ::= <ID>;
Let : Expr ::= Binding* Expr;
Binding ::= IdDecl Expr;
IdDecl ::= <ID>;
```

Figure 3.3: The JastAdd abstract grammar for the CalcASM language.

of `Binding` nodes and one `Expr`. The list is indicated by the * character in Figure 3.3.

We can apply the abstract grammar from Figure 3.3 on the example code in Figure 3.2. In this case there is a `Let` first and will be the child of the `Program` top node. Looking at the grammar a `Let` node should consist of two children: one list of `Bindings` and one `Expr`. The list have two `Bindings` (`a=4, b=2`) and the `Expr` node is represented as the in-end block (`a/b`).

It should also be mentioned that the grammar in Figure 3.3 does not contain the *in* and *end* words from the code in Figure 3.2. This is because they are tokens, and should not be confused with node classes. A certain sequence of tokens will produce one node in the AST, in this case the words *let, in, end* creates a Let node.

To summarize, the abstract grammars defines the different building blocks of an AST, and their relation to one another.

3.4 Abstract syntax tree

An AST is the key component of a compiler, were the structure allows analyses to be performed program it represents (this will be described later). It is generated from input code (a program), the abstract grammars described in section 3.3 and attributes added through reference attributed grammars (RAGs) in section 3.5.

After the code in Figure 3.2 has been parsed and scanned its AST will be generated, but only if no parsing or syntactic errors are found. This is when the semantic analysis in the compilation cycle begins, as mentioned in chapter 2 section 2.1.

The generated AST can be seen in Figure 3.4. If we examine the structure of the AST we can see that it corresponds to the abstract grammar from Figure 3.3. The `Program` node points to an `Expr`, in this case a `Let` node. The `Let` node in turn has two children, a list with `Binding` nodes and an `Expr` node of the type `Div`.

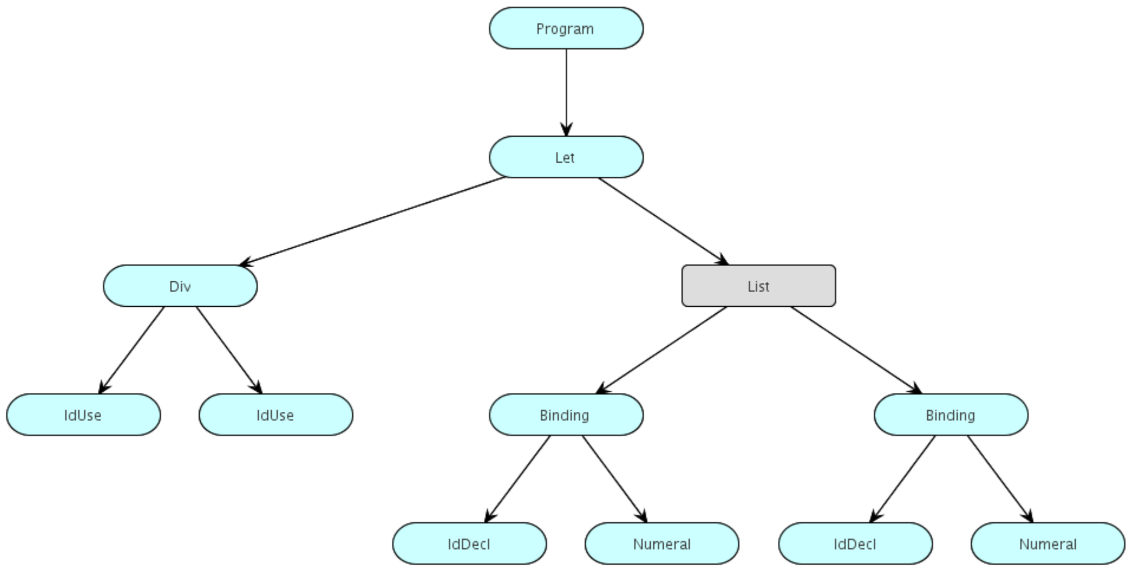


Figure 3.4: An representation of an AST generated by JastAdd.

3.5 Adding attributes to the AST

JastAdd comes with the possibility to alter the AST in Figure 3.4. The developer can add attributes to existing nodes or add new nodes, called nonterminal attributes (NTAs). It is also possible to rewrite or replace certain nodes. In JastAdd, these changes to the AST is defined through aspect files, `.jrag` and `.jadd`. Code written in these files will be, by JastAdd, translated into Java methods, which then are weaved into the appropriate node classes. The code in the aspects is similar to Java code; however the aspect files can also contain ordinary Java code.

Attributes in JastAdd are either defined as synthesized or as inherited, and they can be with or without parameters. Equations are used to define an attribute's value, which can consist of multiple equations. The equations are like a function, meaning that they always return the same value. An attribute can be declared in a parent class, and then let all classes inherit that parent define its equation.

When it comes to attribute grammars, the term inherited should not be confused with the inheritance in Object oriented programming. In JastAdd this for example means that if class `A` contains an inherited attribute, `x()`, all classes with a child class `A` in the attribute grammar must implement an equation for the `x()` attribute.

Attributes can also be declared as circular meaning that its equation can depend on itself, directly or indirectly. Also, there is collection attributes where nodes in the AST are so called contributors. A contributor adds some value to a collection attribute, so the value of a collection attribute will be a sum of the contributors' values. The collection attribute can for example be an arbitrary class which counts the number of occurrences of certain node type. Where each contributor increment the value of the collection attribute when they contribute to the attribute.

As mentioned, attributes are useful when performing analysis on the AST. With the help of the attributes found in Figure 3.5, we show how a namespace analysis can be performed to find if any variable is declared more than once. In the aspect `Errors`, a

```
aspect NameAnalysis {
    ...
    syn boolean IdDecl.isMultiplyDeclared() = ...
    ...
}
aspect Errors {
    ...
    coll Set<String> Program.errors() [new TreeSet<String>()] with add;
    inh Program ASTNode.program();
    eq Program.getChild().program() = this;
    ...
}
aspect ErrorContributions
    ...
    IdDecl contributes error("symbol: " + id() +
        " is already declared!")
    when isMultiplyDeclared()
    to Program.errors() for program();
    ...
}
```

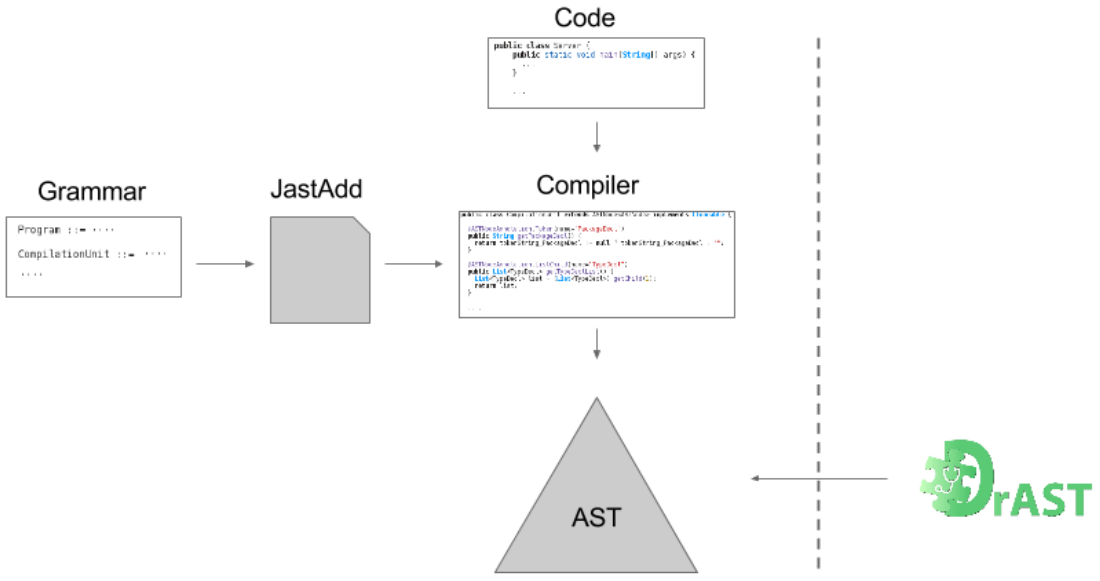
Figure 3.5: Example of JastAdd attributes and aspects.

collection attribute `errors()` is added to the `Program` node. With the inherited attribute `program()` on `ASTNode` we declare that all nodes will have a reference to the top node. The equation for this attribute is defined in the node `Program`. In the aspect `ErrorContributions` it is defined that each variable declaration, represented by `IdDecl` nodes, will contribute an error message to the `errors()` collection if they are declared more than once. The synthesized attribute `isMultiplyDeclared()` checks this, and is defined in the aspect `NameAnalysis`. Note that the equation for this has been omitted as it is quite large. In short it checks the AST for other declarations that define the same variable as itself, and returns true if it does and otherwise false. So if `isMultiplyDeclared` return true the `IdDecl` will contribute an error message to the collection `errors()` in the `Program` node, otherwise not.

The attributes can easily be reached through Java code, which is useful for any domain-specific language. One can reach the result of any analysis or structural information in a easy way and this is used by DrAST.

3.6 Looking at the AST and attributes

So how do we now inspect nodes and their attributes in an AST? How does one usually proceed? The basic way is to define attributes that print the attribute values to the console or to a text file, which often is quite time consuming and tedious. Therefore, to fulfil our goals mentioned in section 1.2.1, we have created a tool called DrAST that allows the developer to inspect the entire or parts of an AST, including all attributes defined for the different nodes. Figure 3.6 is yet another extension of our overview, and shows that DrAST runs after the AST has been created by the compiler. Chapter 4, DrAST, will describe how this tool can be used.



```
Program ::= ....
CompilationUnit ::= ....
....
```

```
public class Server {
    public static void main(String[] args) {
      ...
    }

    ...
```

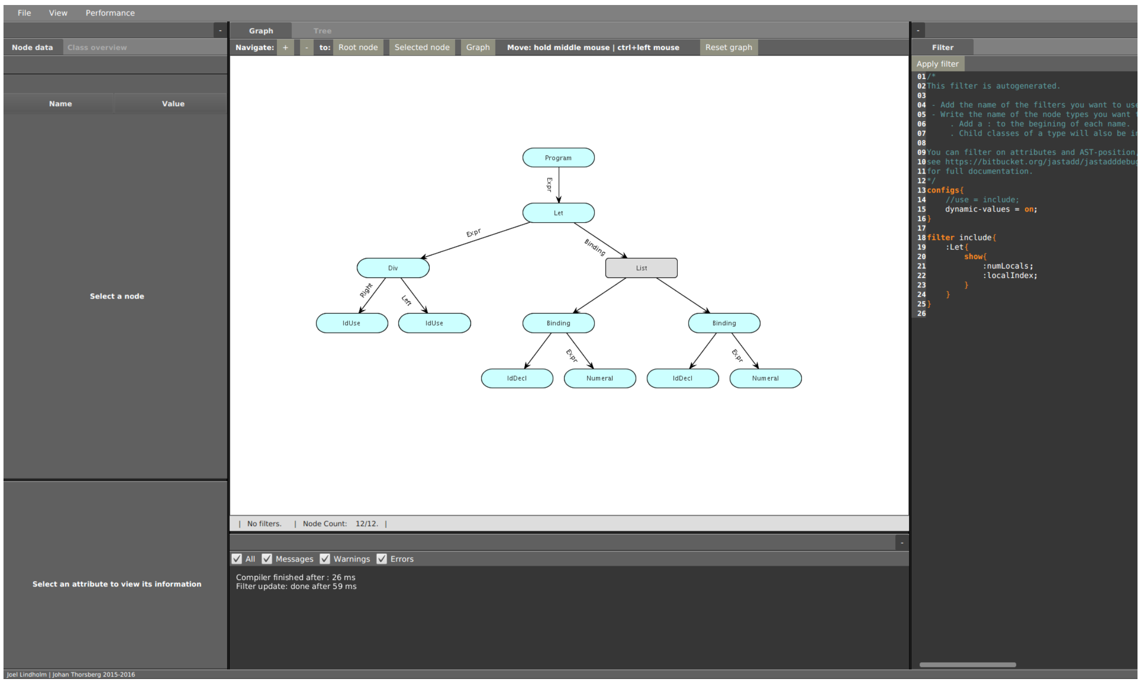
```
public class CompilationUnit extends ASTNode<ASTNode> implements Cloneable {

  @ASTNodeAnnotation.Token(name="PackageDecl")
  public String getPackageDecl() {
    return tokenString_PackageDecl != null ? tokenString_PackageDecl : "";
  }

  @ASTNodeAnnotation.ListChild(name="TypeDecl")
  public List<TypeDecl> getTypeDeclList() {
    List<TypeDecl> list = (List<TypeDecl>) getChild(1);
    return list;
  }

  ...
```

Figure 3.6: An overview of a JastAdd compiler and how DrAST interacts with it.

Chapter 4

DrAST

In this chapter our attribute debugger DrAST will be presented, which is the result of this report. In addition to a description of how to use the tool and some of its key functionalities, a number of design choices will be explained and defended.



```
/*
This filter is autogenerated.

 - Add the name of the filters you want to use
 - Write the name of the node types you want t
     . Add a : to the begining of each name.
     . Child classes of a type will also be in

You can filter on attributes and AST-position,
see https://bitbucket.org/jastadd/jastadddebug
for full documentation.
*/
configs{
    //use = include;
    dynamic-values = on;
}

filter include{
    :Let{
        show{
            :numLocals;
            :localIndex;
        }
    }
}

```

```
|  No filters.   |  Node Count:   12/12.  |
```

✓ All ✓ Messages ✓ Warnings ✓ Errors

```
Compiler finished after : 26 ms
Filter update: done after 59 ms
```

Figure 4.1: A screenshot of the graphical user interface of DrAST.

4.1 Visualizing the AST

To demonstrate DrAST the previously described language CalcASM (section 3.2) will be used. As mentioned, the code in Figure 3.2 will result in an AST. DrAST can be used to visualize this AST. It can start a compiler and extract the needed data. The screenshot in Figure 4.1 illustrates how DrAST looks right after running the compiler for the CalcASM language.

Currently DrAST has two ways to view the AST, Figure 4.2 illustrates them both side by side. The default view, to the left, is the *graph view* with nodes and edges. The second view, to the right, is a *text-tree view* where the indentation specify the relation between parent and its child nodes.

When it comes to visualizing large amounts of data to a user it is important to remember the basic principle *Overview first, zoom and filter, then details-on-demand* [23]. This has been one of the approaches we used when we designed DrAST's interface and functionality. With the *graph view* one get a good overview of the AST, and the attributes for a node can be seen on demand when selecting a node (explained in 4.2). The *text-tree view* on the other hand has better input delay performance, when displaying large sets of nodes, than the *graph view*.

DrAST can handle ASTs with children that are not set, in other words references to children that have the value *null*. If such a value is found an error is cast and a `null` child will be placed in the reflected tree, these nodes are displayed as red nodes in the filtered tree.

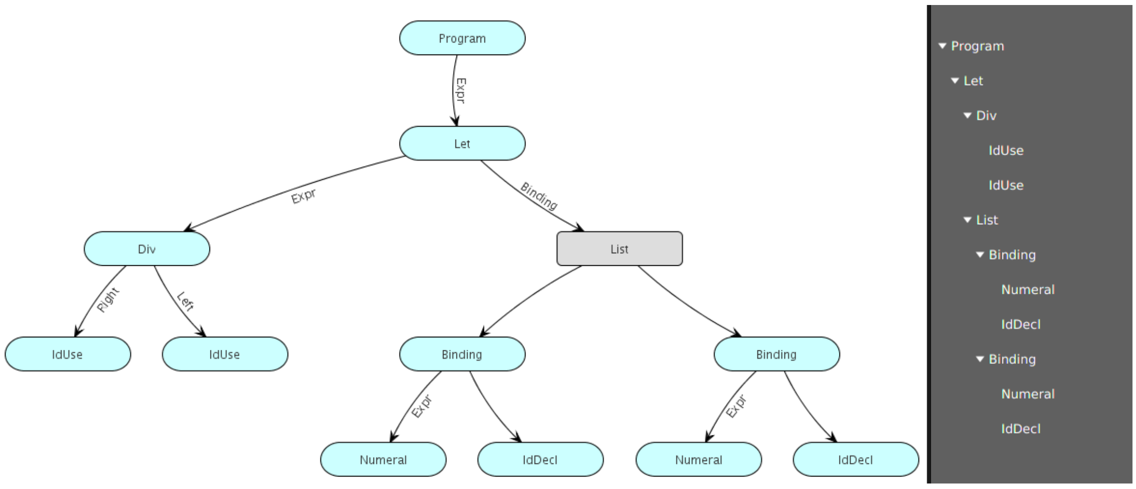
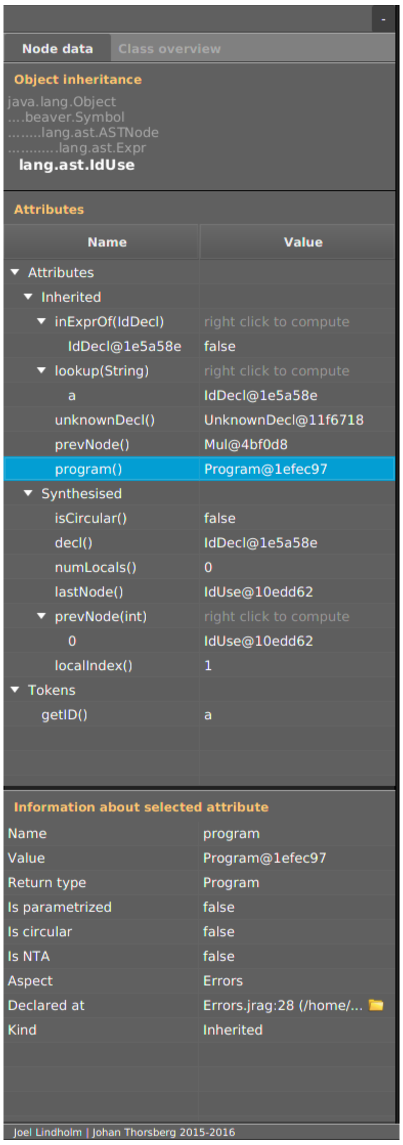


Figure 4.2: The AST can be shown as a *graph view* (left) or a *tree view* (right).

4.2 Attributes

The window to the left in Figure 4.3 is the *Node data* window, which display the attributes for a selected node. This view is split into two columns: *Name* and *Value*. The *Name* column will display the name of an attribute and the *Value* column will display its value, if there is a value. An attribute value can be anything from Java objects to primitive types. Some of the values may also be references to other nodes in the AST. If one of these reference attributes is clicked, an edge is displayed in the *graph view* between the selected and the referred node.



Object inheritance

java.lang.Object
....beaver.Symbol
........lang.ast.ASTNode
............lang.ast.Expr
lang.ast.IdUse

Attributes

Name	Value
▼ Attributes	
▼ Inherited	
▼ inExprOf(IdDecl)	right click to compute
IdDecl@1e5a58e	false
▼ lookup(String)	right click to compute
a	IdDecl@1e5a58e
unknownDecl()	UnknownDecl@11f6718
prevNode()	Mul@4bf0d8
program()	Program@1efec97
▼ Synthesised	
isCircular()	false
decl()	IdDecl@1e5a58e
numLocals()	0
lastNode()	IdUse@10edd62
▼ prevNode(int)	right click to compute
0	IdUse@10edd62
localIndex()	1
▼ Tokens	
getID()	a

Information about selected attribute

Name	program
Value	Program@1efec97
Return type	Program
Is parametrized	false
Is circular	false
Is NTA	false
Aspect	Errors
Declared at	Errors.jrag:28 (/home/...
Kind	Inherited

Joel Lindholm | Johan Thorsberg 2015-2016

Figure 4.3: The *Node data* window: here all attributes and their values can be inspected and computed on demand.

Some attributes are so called parameterized attributes, these have up to an infinite number of values depending on the parameters. In the *Node data* window, it is possible to compute these attributes with user input as parameters.

NTAs are also displayed in this window, but also in the graph as nodes. In theory, there could be a near infinite number of nodes under a NTA. We added the possibility to expand specific NTAs like any other subtree by right clicking in the *Node data* window. DrAST does not by default compute NTAs; however it is possible to configure DrAST to compute specific NTAs.

DrAST is a debugger and a visualisation tool, although it does not simply display one fixed state of a program. It is possible through DrAST to recalculate all attributes, even the ones that were not used during the compiling process. The developer does however have the option to configure DrAST to only show cached attributes, and by doing so show the static state of the debugged program. If an attribute throws errors they will be caught by DrAST, which itself will print the errors to the user.

4.3 Clusters

ASTs can rapidly grow large, with bigger programs and more complex compilers, but a developer is often only interested in some specific part of it. We state in our goals (section 1.2.1) that we must solve this problem with Scalability. Therefore, DrAST comes with the ability to filter unnecessary information instead of illustrating the entire AST. Nodes that do not pass the filter will be, what we call, clustered. This collapses part of a subtree, not necessary a whole subtree, into what we call a cluster node.

To demonstrate how this works we apply a filter on the AST in Figure 4.2 to only include the following node types: `Program`, `Let`, `IdUse` and `IdDecl`. The result of this can be seen in Figure 4.4, this representation is called the filtered tree and is what DrAST will display. Every node that is not one of the classes or subclasses of the types mentioned will be gathered into a cluster.

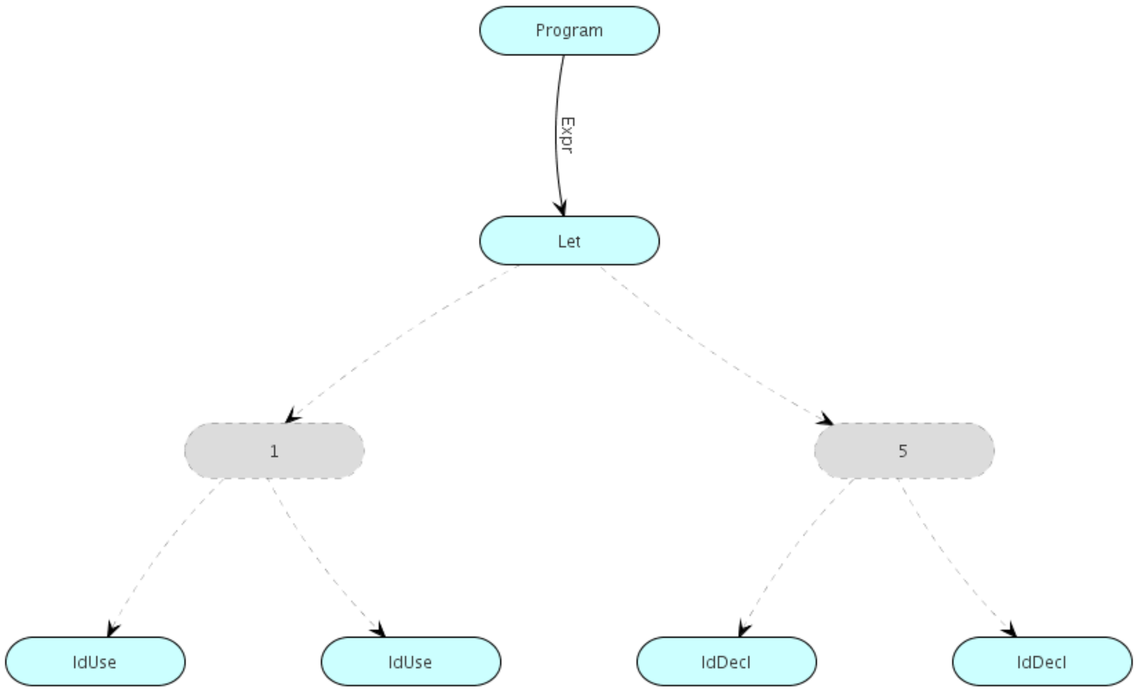
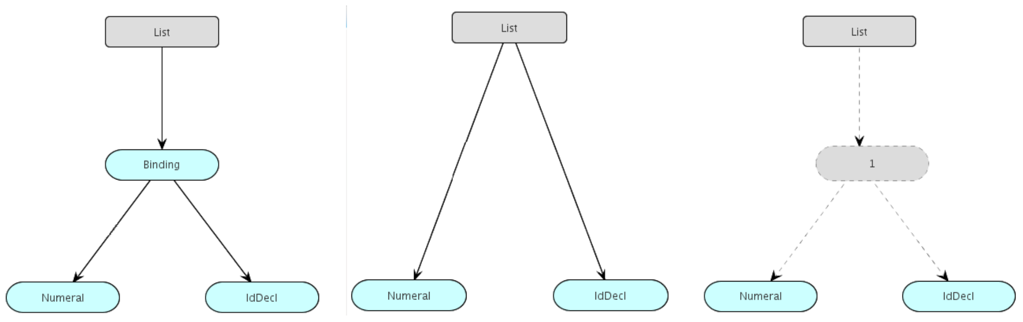


Figure 4.4: A filtered version of the AST in Figure 4.2, with the nodes not fulfilling requirements of the filter clustered into the two nodes (grey) following under the Let node.

The cluster nodes are in place to make the parent structure of the complete AST more apparent. We illustrate the advantages with the cluster node in Figure 4.5. Figure 4.5a is the original tree, and the other two are filtered with or without cluster nodes. We lose both structure and information about the tree in case 4.5b. By using the cluster, we can clearly see what nodes are direct children to a node, and we also get information about how many nodes that are hidden between two.



(a) Example graph **(b)** Removed parent **(c)** Clustered parent

Figure 4.5: Three different visualizations of an AST.

If a cluster node is selected in DrAST the *Node data* window will show a list of all node types in the cluster, and additionally the node count for each type. This helps the developer to see what has been filtered away, so the developer can more easily configure the filter include or exclude more nodes.

4.4 The Filter Configuration Language

To achieve a modular filter solution we implemented a new domain-specific language using JastAdd (Appendix A contains the abstract grammar). This language is simply called the Filter Configuration Language (FCL) [26].

With the FCL a user can define what should be included in the filtered tree. The FCL code is written in a `.fcl` file, which will be compiled by DrAST. This filter file is displayed in a text editor in the graphical user interface of DrAST, so that the user can configure the filter and see the changes to the filtered tree more easily. This editor window is the right window in Figure 4.1. In short FCL works in such a way that the developer writes one or more filters which all then are applied to DrAST's representation of the AST.

The filter describes the node types that should be included in the filtered tree, and all other types will be gathered into cluster nodes. The developer writes the simple-class name of a node type (an example of a simple-class name: `org.ast.Mul` has `Mul`). Some types like the `Expr` in the CalcASM language has a number of sub classes (`Mul`, `Numeral`, `IdUse` and `Let`). If `Expr` is added as a type that should pass the filter, all sub classes will pass too.

A filter could have been implemented in many ways. Different graphical elements consisting of buttons, check boxes and draggable objects could all be good solutions, but we chose to create FCL and a text editor. One problem was that we did not know what feature the filter should contain, and so old features was constantly changed or removed while new ones were implemented. To maintain a constantly changing graphical user interface would be too time consuming, and we wanted to focus on other parts of DrAST. In the end, we felt that a well-defined programming language with high usability would be sufficient and even preferred (by us) over a GUI. The implementation of the FCL also gave us the opportunity to test DrAST during the development of a new language. However, as future work one can implement a GUI for FCL into DrAST.

The FCL contains a number of features, and in the following sections we will explain those that manipulate the filtered tree.

4.4.1 Filter conditions

Filters can also be applied on a more detailed level than just different node types. It is possible to specify conditions that each node of a certain type needs to fulfil to be included in the filtered tree. There are two types of conditions that can be specified; firstly the conditions for attributes in the nodes and secondly the positions of the nodes in the AST. These conditions can be specified for each node type, see Figure 4.6, in a `when` block.

The code in Figure 4.6 contains two node types, namely `ASTNode` and `IdDecl`. Each of them has one condition specified in a `when` block. The `ASTNode` type is a superclass for all nodes in the CalcASM language. The condition for the `ASTNode` will apply for all nodes in the AST due to this (the conditions will propagate downward to all subtypes). This filter will thus only include nodes that are directly children of a node of the type `Binding` in the AST. `IdDecl` nodes will only be included if they fulfil the superclass conditions as well as their own conditions. `IdDecl` should have an attribute with the name `getID` and by the use of quotes in FCL, namely `"a"`, it is specified that the value of the attribute need to be of the type `java.lang.String` and a value equal to `"a"`.

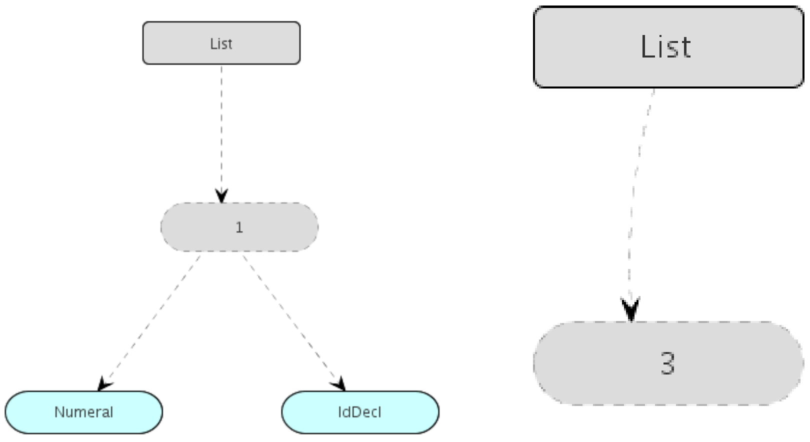
```
configs{
    use = f1;
}
filter f1{
    :ASTNode{
        when{
            child of :Binding;
        }
    }
    :IdDecl{
        when{
            :getID == "a";
        }
    }
}
```

Figure 4.6: Small example code of FCL, showing that the node types `ASTNode` and `IdDecl` are to be included in the AST (not filtered away and clustered) when certain conditions defined in the when blocks are fulfilled.

The conditions that currently are supported are quite basic. Attribute values are compared to some primitive value like *int* or *boolean*, and also *String*. This is usually enough. If more advanced filtering is needed the developer can add attributes with more advanced computations via the JastAdd system. These advanced attributes can in turn result in a simple value, say *boolean*, which then can be filtered on by the FCL. For example the `IdDecl` nodes can have a *boolean* attribute named `FilterMe`, that will return *true* if some analysis yield a correct value, then the developer can just filter on the `FilterMe` attribute. This is implemented this way so that FCL can use the existing attribute mechanisms present in the JastAdd system, without the need to implement a similar mechanisms into FCL.

4.4.2 Subtree conditions

In addition to clustering nodes with normal filter conditions one can collapse a whole subtree, which is useful for large ASTs. These conditions are very similar to the filter conditions, one specify conditions that the node type needs to fulfil to have its subtree included in the filtered tree. If one node fails a condition all its children, direct or indirect, will be collected in to a single cluster. See Figure 4.7 for a comparison between normal filter conditions and subtree conditions.



(a) Part of a tree collapsed, achieved by filter conditions.

(b) Graph with collapsed subtree, achieved by subtree conditions.

Figure 4.7: Illustration of the difference between normal filter conditions and subtree conditions.

4.4.3 Displaying attributes

With FCL one can also enable that attribute values should be displayed directly in the `graph view`. This can be applied for each AST node type, just like the filter conditions. Name and value of the attribute will be displayed directly inside its node in the graph. If the value is a reference to another node though, it will be represented as an edge between the nodes.

If the attribute that should be displayed is a NTA (an attribute that is or can dynamical create a new AST node), it and its subtree will be calculated added to the model. The filter will be applied to these nodes as well. This makes it possible to create an endless loop of

NTAs, if one NTA create another by the same type and so on. FCL has configurations that can prevent these kinds of infinite loops, by defining a recursive depth.

4.4.4 Styling

We also added the possibility to style nodes through the filter. Node colour, shape and the border around the node can all be defined, and will override the standard colours in the user interface.

4.4.5 Multiple filters

FCL supports multiple filters that can be enabled or disabled by the user. FCL will collect all conditions and other values for each of the AST node types in to collections during the compilation of the filter, but only from the enabled filters. This means that if a node type is defined in two filters, all AST nodes of that type need to fulfil the filter conditions from both filters to be included in the filtered tree.

An example of this use is that one could have a filter for deciding which nodes that should be shown, and a style-filter that only changes the styles of each node. This style can then be activated or not, without changing the filtering of nodes.

Chapter 5
Implementation

This chapter will describe the different methods and Java libraries used to develop DrAST. DrAST follows a Model-View pattern, where the model is separated into its own package and has no connections to the view. DrAST receives the root node of the AST, as a Java Object, and through Java reflection extracts the data and constructs its own representation of the AST to create the model. The model contains two versions of the tree: the reflected tree and the filtered tree.

The implemented graphical user interface is a view that visualizes the model. The GUI is built with JavaFX, a set of graphical packages for Java [5]. The *graph view* is created using the graph framework Jung [14].

5.1 The model-view architecture

The goal of the project was to create a tool for AST implementations, but the structure of the DrAST system is designed to potentially be expanded for more uses. Figure 5.1 show the architecture of the system. The architecture is divided into one model and any number of view-controllers, which we call views. As indicated by arrows, the model has no dependency on the view, and performs its computations without knowledge about the view (Figure 5.1). The model uses a Java object as a parameter that represents the root of the AST (top node). All necessary data is extracted from the object and creates the data in the model. Outside the model, views or other Java programs can use the data model in any way they want. The graphical user interface presented in this report is an example of a view implementation.

The views do not have direct access to the model, but rather through a long list of public java methods. These methods together could be called the API. If the view want to compute an attribute, change the filter or access the filtered tree it communicate this to the API, which in turn will alter or use the model to return the result. Figure 5.1 show some methods that can be used for communication.

Further, views can either create a model or receive one from an external source. The user can through the GUI export the graph as an XML file. The GUI view does not create this file itself, but calls another view (DrAST XML) and passes its own model as an argument. So a view can thereby use other views, and share its model.

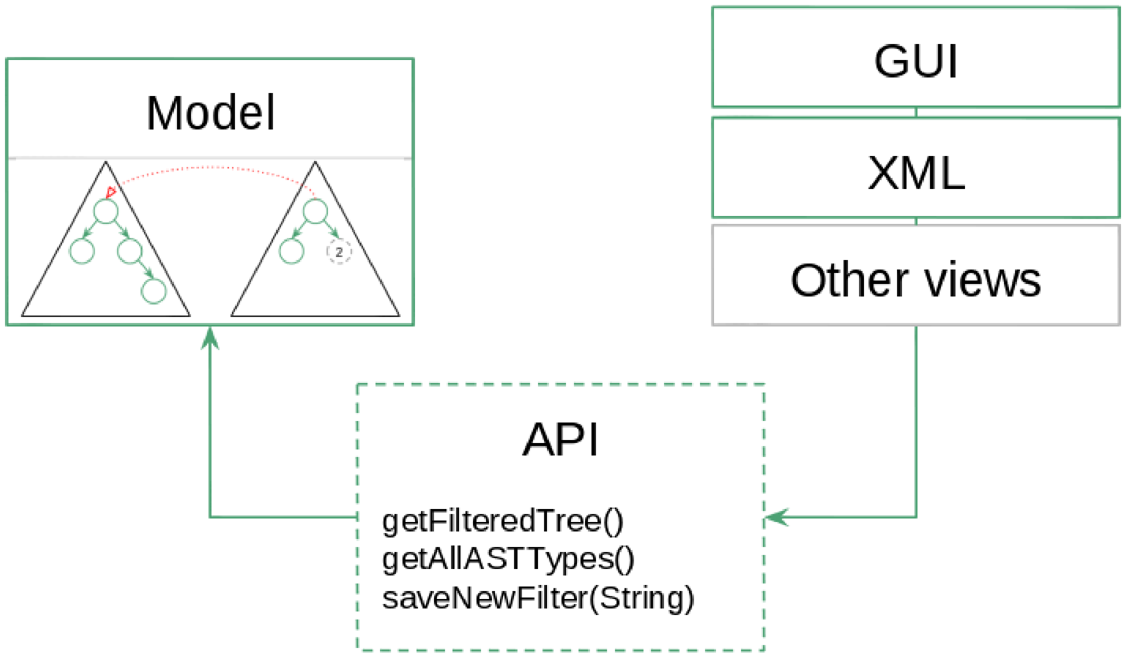


Figure 5.1: The model-view architecture of the DrAST implementation. The arrows indicate dependency. Views (GUI, XML) are dependent on the model.A view communicates with the API that in turn uses the model to return a results based on the request.

5.2 The model

The model has two representations of the original AST, which can be seen in Figure 5.2. The root object is passed into DrAST from the compiler. From this root node we traverse down to all the leaves with reflection (see section 5.4.1). Each node is placed in its own container class, and all these containers together become the reflected Tree. The reflected tree is thereby a one-to-one representation of the original AST, where each container node has a reference to the AST node it represents.

The last structure, the filtered tree, is created based on the reflected tree. First the model uses an internal compiler to compile the filter. The filter is then applied to each node in the reflected tree. If a reflected node passes the filter, it is placed in yet another container node that will be used by the filtered tree. If it does not pass the filter it is placed in a cluster node. A cluster node contains references to all reflected nodes it contain; no new container is created for these reflected nodes.

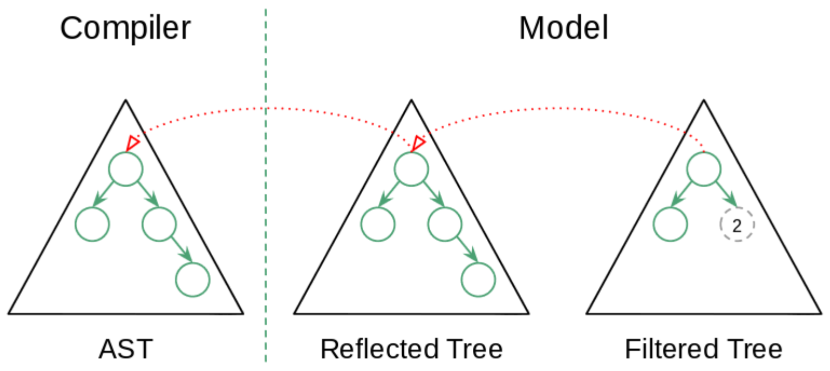


Figure 5.2: The reflected tree is DrAST's representation of the original compiler AST. From this tree, the filtered tree with cluster nodes is created. The reflected tree and the filtered tree together form the model. Each node has a reference to the node it represents in the previous tree (illustrated by the arrows).

5.3 Running DrAST

There are two ways DrAST can debug a compiler (Figure 5.3). It can either be done through a call to DrAST via the compiler (compiler call), or through an addition of a static field in the main class of the compiler, followed by letting DrAST run the compiler (DrAST starter).

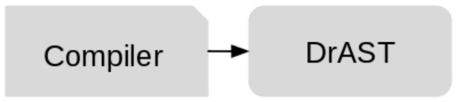
Both ways are valid, though the DrAST starter might often be the preferred method as the compiler does not need to import DrAST in order to run. DrAST have quite a number of library dependencies which the compiler unlikely need, and will make the compiler grow substantially. All one need to do in the second option is to define what node that will be used by DrAST by assigning it to the static field.

5.3.1 Compiler call

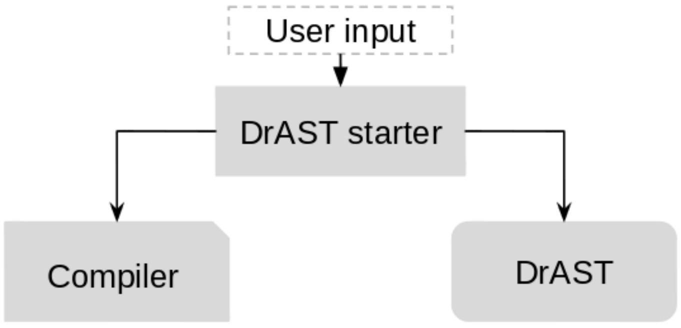
The compiler call works as following, and is illustrated by Figure 5.3a. The compiler imports the DrAST library and calls it itself. It can either create the model or a DrAST view (for example the GUI), with the root node as a parameter. Figure 5.4 show some example code of a compiler calling the GUI view. When the GUI is called (or any other view), it will in turn create the model. The filtered tree is created and can be reached by any view, for example the GUI.

5.3.2 DrAST starter

The second alternative (Figure 5.3b) is by adding a static field in the compiler, called *DrAST_root_node*. Figure 5.5 show some example code of this. The object in the static field of the compiler will act as the root in DrAST. DrAST starter first calls the main function of the compiler and then fetches the *DrAST_root_node* field. User input is required to get the arguments needed to run the compiler. The Object from the field is then, like when using the compiler call, passed to a view or to the model.



(a) Compiler call: the compiler starts DrAST



(b) DrAST starter: first call compiler and then DrAST

Figure 5.3: The two ways to start DrAST, compiler call and DrAST starter.

```
LangScanner scanner = ...;
LangParser parser = ...;
// Compile some code with the scanner and parser.
CompiledProgram rootNode = (CompiledProgram) parser.parse(scanner);
if (program.errors().isEmpty()) {
    // start DrAST gui tool
    DrASTGUI gui = new DrASTGUI(rootNode);
    gui.run();
} else {
    ...
}
```

Figure 5.4: Example of code for DrAST compiler call where the compiler starts DrAST from the compiler by calling the method *run()*.

When the debugged compiler's classes are loaded at runtime there could be problem with DrAST's own filter compiler created with the JastAdd system, since Java cannot distinguish between two classes with the same name and package [19]. To avoid method calls on the wrong classes we therefore created our own custom class loader, instead of letting Java handle it with its own class loaders. In Java, class loaders are connected in a hierarchic order [19, 7]. There are several loaders that each contains a number of class paths and its parent loader. When a program is searching for a certain class in a loader, the loader first asks its parent to search for the corresponding class. Only if the parent class loader does not find the class the loader it will conduct a search. The above described problem, two classes with the same name from the two different compilers (debugged and filter compiler), will result in calls on the wrong class. Our custom class loader is created to avoid this. It contains all possible class paths in the debugged compiler, and is placed at the bottom of the loader hierarchy for the debugged compiler, thereby being the first loader activated. Our loader will first search itself for the wanted class, and only if it is not found, calls for a search by its parent. The filter compiler and the debugged compiler can therefore use their own class versions, and still have any number of classes or libraries with the same name without any problems.

```
public static Object DrAST_root_node;
...
public void runCompiler(){
    LangScanner scanner = ...;
    LangParser parser = ...;
    // Compile some code with the scanner and parser.
    CompiledProgram rootNode = (CompiledProgram) parser.parse(scanner);
    if (program.errors().isEmpty()) {
        DrAST_root_node = rootNode;
    } else {
        ...
    }
}
```

Figure 5.5: Example of code for DrAST starter where the compiler sets the *DrAST_root_node* field to the root node of the AST.

5.4 Building the reflected tree

As explained DrAST creates its own representation of the compiled AST that is called the reflected tree (Figure 5.2). In order to make DrAST run regardless of what JastAdd-generated language the compiler is built upon, we had to ensure that DrAST has no dependencies on any specific grammar. In other words, DrAST should function without knowing what Java classes the compiler is built on. This was achieved by using the Java Reflection API [6] and Java annotations produced by the JastAdd system. With this we can extract the necessary information needed to represent an AST of any JastAdd compiler. This section will explain in more detail what methods were used to achieve this.

5.4.1 Reflection

In order to extract data DrAST uses Java reflection, which enables modification and examination of the runtime behaviour of an application that is running on a Java Virtual Machine (JVM) [6]. With reflection one can perform operations on an application that otherwise would not be possible, for example change the access level modifier of a method or class. It is also possible with reflection to invoke methods of an object without knowing its type. An example can be a class `A` with the method `foo`, which generate its value `bar`. If we have a reference to a generic Java object which we assume is of the type `A`, we can compute the method `foo` without type casting and thereby retrieve its value `bar`. With this it is possible to, without knowing the type of an object at compile time, retrieve the value of a method [6], parameterized or not. See Figure 5.6 to see an illustration of the example.

The return type, access modifier, annotations, parameter types and more are examples of other information that can be extracted with Java reflection [6]. With Java 8, it is also

```
//Normal Java code to extract the value bar from the method foo()
A a = new A(); //A is a known type
//This is the standard way of computing methods
int bar = a.foo();
...
//Reflection code which the value bar from the method foo()
Object a = //Given a value some where, which we assume has a method foo()
//Using reflection to find and compute the method foo()
int bar = a.getClass().getMethod("foo").invoke(a);
```

Figure 5.6: Example code of Reflection code compared with normal Java code, where the goal is to retrieve the value `bar`.

possible to extract the names of the parameters for any method or constructor, but this requires a compiler flag, namely `-parameters` [6].

Java reflection is usually used for special cases, as there are a number of drawbacks [6]. With reflection it is possible to invoke methods, but there is no way to see what they actually do. Invocation of a method can produce unexpected side-effects, especially if it involves methods not normally reachable, for example methods declared private. This could change things within the object that normally should not change, and the user would not understand why.

However we still need something that tell us what information that is connected to the AST, for example which methods that represents attributes. This problem is solved with generated Java annotations, which are explained in section 5.4.2 below.

5.4.2 AST annotations

The JastAdd system produces a number of annotations, in the generated classes, for the methods that represent the attributes, tokens and children references in the AST. In short the annotations help us to find the correct methods in the classes so we can extract the correct information and understand what they represent.

The JastAdd annotations are currently in the form of *@ASTNodeAnnotation.value*. The *value* can be either *Attribute*, *Token*, *Child*, *ListChild* or *OptChild* in the latest released JastAdd version 2.2.0. The *Attribute* and *Token* value correspond to their type, an attribute or a token. The other three, namely *Child*, *ListChild* and *OptChild* represent references to the children of a certain node in the AST.

To create the reflected tree we traverse the entire AST provided by the compiler, by invoking the methods annotated with *Child*, *ListChild* or *OptChild*. We presume that the objects returned from these methods represent nodes in the AST and they are put in a container class. This process continues with the the child nodes, until we cannot find more of these annotated methods. During this traversal we store basic information about the nodes, for example if the node is a *List* or an *Opt* node.

During the development of DrAST new annotations were added to the JastAdd system, to help us extract more data about the AST (Figure 5.7). It follows the same pattern as before but the *value* part can now also be *Source*. Also, a number of value fields were added to the annotations. An example is that the *Source* annotation now contains value fields such as *aspect* and *declaredAt*. The *aspect* field contains the name of the aspect in which the attribute was declared in, and the *declaredAt* field contains the file path to the aspect. The attribute annotations now also contain a *kind* field that specifies the kind of an attribute; *synthesised*, *inherited* or *collection*.

```
...
  @ASTNodeAnnotation.Attribute(kind=ASTNodeAnnotation.Kind.SYN)
  @ASTNodeAnnotation.Source(aspect="MyAspect",
                             declaredAt="somePath/MyAspect.jrag:7")
  public abstract String getID(){
    ...
  }
...
```

Figure 5.7: An example of the new JastAdd annotations for a synthesized method called getID.

DrAST do not know that the objects it extracts are actually from an AST, it blindly invokes the annotated methods. This creates the possibility to use DrAST on any Java program, of any kind. If a developer is interested in viewing some structure in their code (tree or linked list are two examples), it is possible to just add these annotations to visualize the structure in DrAST.

5.4.3 Cached values

The JastAdd system caches the resulting values from computations of the attributes, so that these values are available without need for recomputation each time they are needed, increasing overall speed [25]. The generated classes have, for each attribute, a private field where cached values are stored [25]. For parameterized values the storing can occur in a Java Map.

With reflection we are able to find these private fields for each AST node.

In the current implementation cached values are found by making assumptions on how the name of the field is structured for an attribute, and then finding a field with the same name. This approach is not entirely safe and prone to errors, but it is the only one currently available. For example, if the JastAdd system is updated and the way private fields are named changes, this can mean that we search for the wrong names - so that present fields cannot found. This can be avoided by adding annotations to specify the names of the fields.

5.4.4 Modularity

DrAST, in its current state, uses Java reflection to extract the data necessary for creating the model. In chapter 7, we discuss alternative methods. We have isolated the current reflection solution of the model so it is not dependent on this particular solution, but instead on an interface that should contain all data. This allows for alternative methods to be implemented and replacing the current one.

5.5 Frameworks and libraries

In this section the frameworks and libraries DrAST utilize are presented and described.

5.5.1 JavaFX

JavaFX [5] is a software platform for Java, and is used to create the GUI of DrAST. JavaFX is intended to be the new way of creating GUI based applications in Java, and is embedded into JRE/JDK package in Java 8. JavaFX uses an XML based language to create the GUI components called FXML, and these components can be styled through CSS or code.

JavaFX was used because it is the new official version of GUI handling in Java, hopefully this will mean that the maintenance for this framework will be longer than for other libraries. Also, our previous experience with XML and CSS made JavaFX a good option to work with.

The graph library Jung [14] (section 5.5.3) is built on Java Swing [4]. This created some challenges. JavaFX have the possibility of embedding Swing applications, which helped greatly. However this disrupted the normal JavaFX structure flow a bit, because the Swing components and the JavaFX components run on different threads. Some handling of the multiple threads was therefore needed to be implemented in the appropriate classes.

As mentioned, the JavaFX library is, at the time of writing this report, part of the latest JRE/JDK package. However, it is not included in the path of some Linux distributions, why it has to be manually added as an external library on these machines.

5.5.2 JUnit

We used JUnit tests suites to make sure things work as they should after adding new features or changing old code. These also give future developers helping hints so that they can avoid breaking any existing functionality with their additions to DrAST. Three different test suites have been created: Input from file, input from code and GUI tests. The two former tests the model of DrAST, while the latter suite only tests the GUI. The three types of test suites are further described below.

Input file We used input file tests for the structure of the AST representations in the model and for the structure of the filter language. Each test has its own folder and contains three files: a filter written in the filter language, an input file and an XML file with the expected output.

Two programming languages were used when performing the input file tests: CalcASM 3.2 and FCL (4.4). We used CalcASM because we needed a stable language that was simple and with an AST with that had some variation. Tests with CalcASM were implemented to make sure the model's AST representations were correct. The FCL was used to check that the structure of the filters stays the same, even though a developer changes or adds to parts of the code. Also, it works as a reminder for updates of FCL's documentation after structural changes.

A third case was added to check the performance of the model, by creating the model on a large AST. This case can be used as an indicator for when it is time to optimise the model.

Code generated input Manually created ASTs are supported by tests of this type. This does for example simplify creation of trees with null children.

GUI testing To perform automatized GUI testing we used an open source library called TestFX. TestFX can emulate mouse and keyboard input events on JavaFX components, so with TestFX one can control a components' position and status. However, TestFX does currently not support embedded Java Swing components.

The GUI tests perform action calls on buttons, test keyboard shortcuts and so on. With these tests we also controlled that the *text-tree view* succeeded in its creation.

5.5.3 Jung

The Java Universal Network/Graph Framework (Jung) is a Java Swing based library for modelling, analysing and visualising data [14]. In Jung data can be represented in a number of different ways, for example tree graphs or networks. DrAST uses version 2 of this framework to visualize the AST.

Input events such as mouse and keyboard are handled by Jung internally. The developer simply chooses what events that is interesting and Jung will handle them. Selection of nodes, moving nodes, panning the camera and zooming are some examples of such events.

To change the look and feel of a Jung graph something called Transformers are used. Each individual transformer handles either an edge or a node and transforms its appearance. Examples of transformers are the shape of a node, or the colour of it.

The built in API for basic mouse and keyboard events saves a lot of time, because it makes it relatively easy to extract the clicked node and perform UI changes upon such events. The transformers collect their information about the nodes and edges from the model, something that makes it easy to change the design of the graph by simply changing the value in the model.

Jung performs quite well for smaller ASTs, but when the number of nodes starts to rise, above 10000 nodes, the performance starts to drop. The Jung library renders the graph every frame, and with a large set of nodes, edges and complex transformers the performance will be drop significantly. To boost the performance we made a simple optimization: if the filtered tree contains over a certain number of nodes, the edges and labels will not be rendered during any navigation event.

Problems with Jung

Although Jung is a robust framework with a high extendibility it has some problems when one want to add a specific behaviour to a layout or transformer. Most of these can be solved by extending the class that performs the operation and override a conflicting method.

The problem with this is that methods overridden usually contain a lot of specialized code, for example the computations for scaling the graph, and when overriding these methods one can lose the version handling of the framework, so one's own code will not be effected by future updates. This can be error prone and troublesome when using the Jung framework.

Chapter 6
Evaluation

So far we discussed the functionality of DrAST, but not how well it performs in different areas. DrAST has been evaluated for its performance (time, memory and lag), and for its usability. The methods for the evaluations and their results will be described in this chapter.

6.1 Usability

The potential users of DrAST could be both professionals working with compilers development with the JastAdd system, like The JastAdd Team that is supervising this project, or students that are learning the basics in compiler technology. One of our goals (section 1.2.1) was to create a tool that is understandable and non-restrictive for new and experienced users.

The usability tests focused on finding faults in two cases. Firstly, we wanted to control if the GUI hinders the developer by not being intuitive enough. Secondly, the usability can be insufficient regarding the documentation. We therefore also wanted to see how different users read FCL's documentation, in order to improve its language and structure.

6.1.1 Think aloud

We decided to use the Think aloud [18] method when conducting the usability tests. First we thought of doing more complex tasks, where the user would find and solve bugs in a language grammar using DrAST. While a test like that could demonstrate how powerful the tool is, it is not a suitable method to find problems in usability. Instead we decided that small, easy tasks in an already complete language better would meet our needs ([18]). If the user could figure out the given tasks without being familiar with DrAST, we deemed usability to be of good quality.

The same method was used for every participant: we read a number of tasks aloud which the participant had to solve. At the same time, the participant had to continuously give oral feedback about their view and experience of DrAST. The task were read aloud to encourage a dialog and making the process of giving feedback aloud more natural. The feedback was documented and conversations between us and the participants were kept at a minimum. We only answered questions when we deemed it would have no impact on the outcome of the test, since the participant was supposed to complete the tasks without our help.

Test groups

As mentioned, the users can be both professionals and beginners in JastAdd development. The participants were divided into two groups. We started with only two professionals, since a small test group of 1-4 persons is enough to find major usability problems, according to Lauesen ([18]). These two participants used the same version of DrAST, so that we could see whether or not they both got stuck on the same problems. After this first test group, we made changes based on their feedback and added new features. The next group consisted of four people with varied experience in JastAdd development. As a common knowledge base all had completed the same compiler course at Lund's University. In contrast to the first test group, we made changes to DrAST after each test and the next participant tried the new version.

Tasks

Each participant had to solve in total 9 tasks (for all see Appendix B), some examples are:

- The root of the AST has an attribute nodeCount. What is the value of this attribute?
- Create a filter that hides all nodes except Program, Let, IdDecl.
- IdUse have an attribute decl that points to an IdDecl somewhere in the three. Draw, for all IdUse:es, these reference edges directly in the graph.

The tasks are designed to see if a participant, without any background knowledge of DrAST, can figure out how to solve them by just having access to DrAST and FCL's documentation [26]. We gave as little information as we could about DrAST in the task description, to stay away from so called hidden help [18]. To summarize, the tasks hinted on features in DrAST but not how to use them. By observing how the users worked with these tasks we could identify potential faults in both the GUI and the documentation, which decreased the overall usability of DrAST

The tasks that required reading the FCL's documentation [26] allowed us to see how users interacted with the documentation. At the time of writing this thesis, the documentation is available on a single Wiki page, and therefore only scrolling is needed to find information. We did not only observe the time it took the users to find the information, but also which search techniques they used. The documentation was displayed in a web browser, which let participants use word search functions. Not only did the participants read the documentation the traditional top-to-bottom way or reading titles but also searched with key words like "child" and "reference". It was important that these words were at relevant position in the text.

6.1.2 Results of usability testing

As we described above, we did changes between the user tests, based on the feedback. We did not have any measurable data on usability. However, we found a lot of usability problems, bugs and got ideas for new features. Here we will list some of the more interesting findings.

Usability problems

Usability problems are what we called problems that had an direct effect on DrAST's usability. For example buttons that the user did not see because of their colour are such a problem.

We noticed that it was difficult to understand the use of the class overview window, which explains how the classes in the graph are connected (for example a *binding* node has *IdDecl* and *Value* as its children). Most participants thought the window displayed the grammar, until they looked closer. We did not improve this window, because we decided that this window should be used for something else. However, in the end we added a text at the top of the window explaining its use was.

There was also lack of feedback, given by the GUI, at a number of places (waiting for the compiler and calculating an attribute value are two examples). To solve these problems we changed the position of information, implemented loading cursors and loading dialogs.

Features

Ideas for new feature that could enhance the usability came to light through the tests. We noticed that some users tried to use the keyboard instead of clicking in the GUI, so we added a number of short commands to DrAST. Saving the filter, rerunning the compiler and hiding all windows except for the graph are some of the implemented commands.

Another feature was the ability to open the source code for attributes. If, for example, a user notice that an attributes value is faulty, it is possible to open the file that define the attribute (the `.jrag` or `.jadd` file) in the default program for the machine.

Bugs

Most bugs or problem with GUI occurred when the user did not do "as they should". Mostly it was short keys that did not work (pressing enter to press a highlighted button), windows placing themselves over each other or that the graph positioned itself wrongfully on the screen.

User survey

At the end of each test session the participant was given a small survey consisting of three questions with the option to answer either yes or no. Again, this has no statistical significance but helps as an indication. The questions were the following:

Do you think this tool would have been useful during the compiler course?

Do you think this tool would be useful today, if you worked with JastAdd?

Do you think you could recommend the tool to other people?

In the end, all participants answered yes on all three questions.

6.1.3 Conclusions from usability tests

The tests conducted were not designed to provide data for statistical analyses. The test groups were too small and the changes made between tests make a comparison among the feedback from participants difficult. Also, such a comparison would not have been relevant, as the participants' prior experience of working with JastAdd varied a lot. In order get sufficient data for statistical analyses, the usability tests need to be further developed. Regardless of this, the results of our tests enabled identification of the worst usability problems and indicated the suitable direction of future developmental work.

In the end all participants completed the tasks within a short time span. The total time for each test session ranged from 20 minutes to over an hour, depending on how much feedback the participant gave.

To summarize - from this we cannot statistically quantify and prove the level of usability in DrAST. However, we can still conclude that users of a varying degree of JastAdd experience could manage to do the tasks they were assigned, and that they all had a positive attitude towards DrAST. This implies that our goal about usability (section 1.2.1) is reached.

6.2 Performance

Performance tests were conducted to see how well DrAST performs on ASTs of different sizes. The results from these tests hinted on the machine requirements for DrAST, and also indicated if and where optimizations were needed. Time, memory and input lag were the main metrics and the model (Section 5.2) the main focus.

In order to test the performance of DrAST we used two different compilers: CalcASM and ExtendJ [9]. CalcASM is the compiler used as an example in this report, and is a small compiler with only 9 node classes and with few attributes in each class. ExtendJ is a Java compiler that is considerably larger, with around 250 - 260 node classes, each of them usually containing a large number of attributes (some classes have around 30 - 60 different attributes).

The machine used for these tests had an Intel Core i7 870 CPU with a clock frequency of 2.9 GHz and 16 GB ram.

6.2.1 Time

The time it took to create the model was measured in two ways. Firstly, we used the time needed to create the full model - both the reflected and the filtered tree. The second measure we used was the time required for building only the filtered tree.

For each of the compilers a set of programs were run. For ExtendJ 16 different Java programs of varying sizes were chosen. As the CalcASM language is very small and large programs do not exist (and are not feasible to create manually), we created a script that automatically generates 16 CalcASM programs with AST sizes corresponding to the Java programs'. The largest programs had about 170 000 nodes; for CalcASM this corresponds to 64 200 lines of code and for Java around 2000 lines of code. The reason Java programs can have such extensive ASTs with so few lines of code, is that they use a large amount of libraries represented in ASTs as NTAs (section 3.5). A Java program with 2 lines of code could produce an AST of approximately 2700 nodes, while a CalcASM program (not using so many libraries) would need 1000 lines of code to generate an AST of the same size.

Time to build the entire model

Here we present the total time it took to build the entire model. The 32 programs, 16 for each compiler, were executed 10 times each. A single filter was used for all programs. The filter in question was configured to include all node types, without exception. Figure 6.1 show the result of the performance tests, where the bars represent the average time it took to create the model for each program, including 95% confidence intervals. Note that the x-axis is not linear; the values (number of nodes generated) are only following size order.

In Figure 6.1 we see that the confidence intervals are small, which means that the time it takes to create a model of a certain size is relatively constant. If one creates the model for certain a program multiple times the time for each iteration will be approximately the same, this is indicated by the confidence intervals. The number of nodes seems to directly affect the total time, so that time increases linearly with the number of nodes in an AST.

The difference in time between ExtendJ and CalcASM can be explained by the difference in number of attributes included in the two languages. Each node in ExtendJ has a lot more attributes than the nodes in CalcASM. As mentioned in 5.4.2 each method in a node class can represent an attribute, token or edge in the AST, meaning that more attributes results in more methods. DrAST must, for every node class in the tree, iterate through all methods and read the Java annotations to be able to build the reflected tree. The methods are cached for each node class so the full iteration will only be done once every class. This caching thereby makes the number of attributes less important for the total time it takes to generate the entire model. Without the caching, we would expect bigger differences between the two compilers in mean total time.

We deem the total time required for creating even the larger trees of 170 000 nodes acceptable, with a mean of 1.2 seconds when using ExtendJ.

Time to build the filtered tree

We believe it is acceptable that DrAST does not start immediately, but after that point it should run smoothly. After the models first creation one can apply a new filter which will create a new filtered tree, and it is important that the new model is ready after a short time. When a new filter is applied, one iteration is done on the reflected tree and every node that passes the new filter is added to the filtered tree. We wanted to see the impact of this iteration.

The same sets of programs from the last section (6.2.1) were executed for both compilers and their models were created. We recreated the filtered tree 10 times after this by

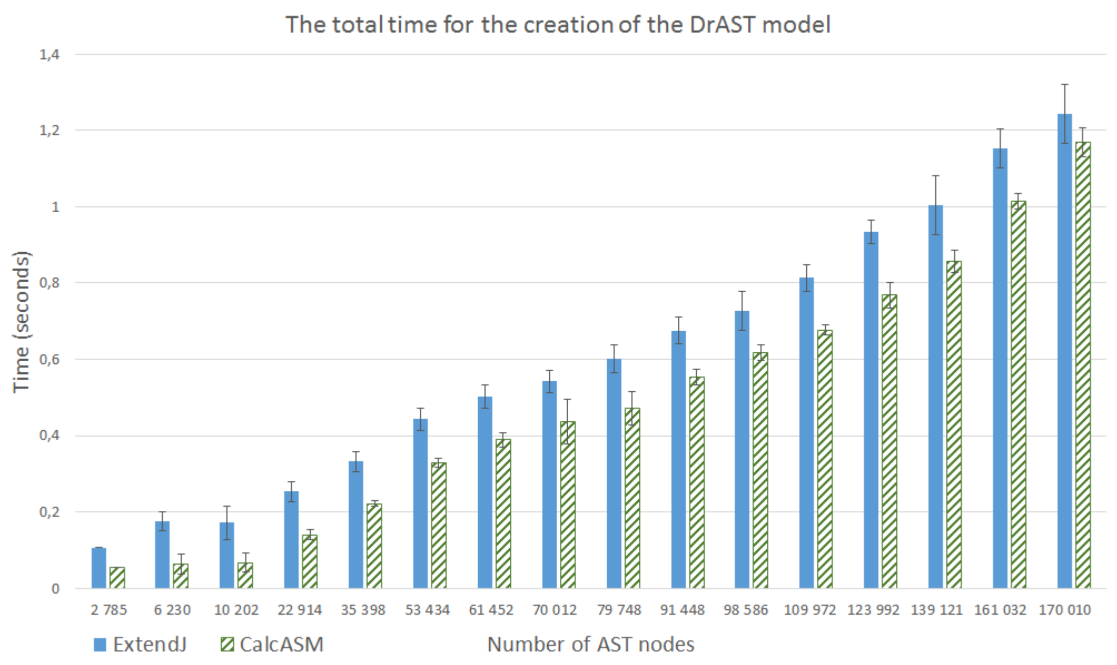


Figure 6.1: Average time to create the model, for trees with different numbers of nodes, with two different compilers (ExtendJ and CalcASM).

applying a new filter over and over. The result of this can be seen in Figure 6.2. The graph is similar to Figure 6.1. Each bar represent the average time for the filtered tree to be created. The confidence interval is marked at the top of each bar as a line, here as well we used a confidence interval of `95%`.

Again, the confidence interval is small, indicating that the number of nodes seem to be the biggest factor when creating the tree. According to J. Nielsen [20] a time below 0.1 second gives the impression that a system reacts instantaneosly. Above that up to 1 second will be noticed, but the *user's flow of thought* will stay uninterrupted. We see that for the program with 170 000 nodes it takes 0.35 seconds, which is well below the 1 second limit. Note that this time only represents recreation of the filtered tree, if a view like the GUI also is used, this will add additional time.

6.2.2 Memory

The compiler holds the original AST, and within the DrAST model we have the reflected tree and the filtered tree. The GUI adds two additional representations with the graph and text view. To this we also have a large number of different caches and the whole GUI with buttons, windows and so on. We are interested in how much memory that DrAST with the GUI use when running, especially on large ASTs.

We generated 5 different programs for the CalcASM compiler, with different sizes ranging from around 2 700 to 170 000 nodes (around 1 000 to 64 200 lines of code). We ran the compiler that in turn started the DrAST GUI. We printed the memory use (in megabytes) at three set points, running garbage collector right before each print. The

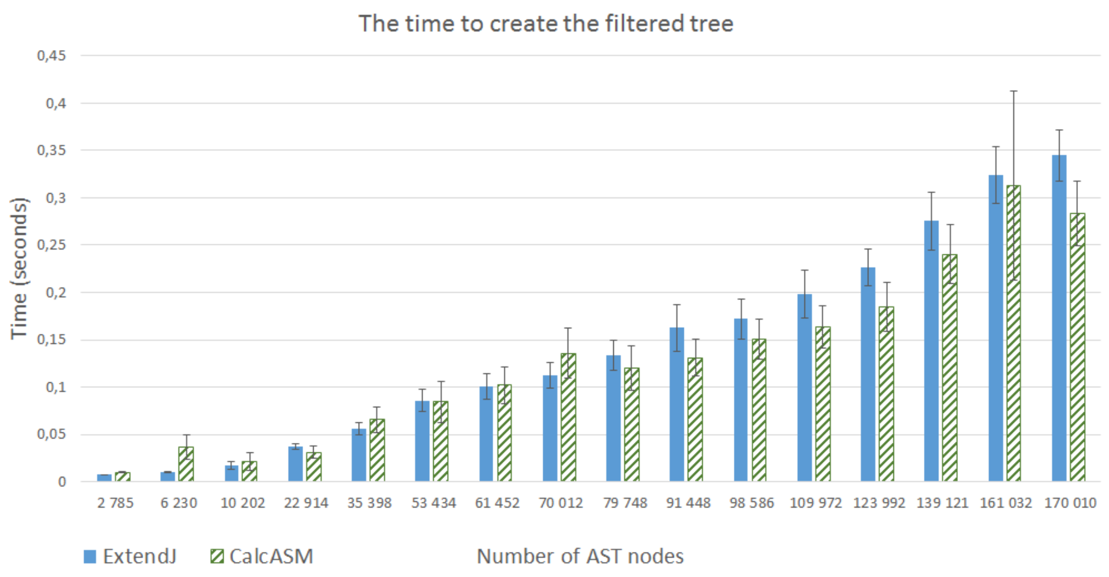


Figure 6.2: Average time to rebuild a filtered tree, for trees with different numbers of nodes, with two different compilers (ExtendJ and CalcASM).

memory-use data was printed when the compiler was done, when the model was done and when the GUI was running. We did this two times for each program with different filters. First with a filter that passed all nodes and then only passing around 900 nodes. Table 6.1.

The largest memory use of 544 megabytes was when the AST contained around 170 000 nodes and the GUI was running. This should not be a problem for machines of today. By filtering this large tree to only 900 nodes, we lowered the total memory use to 157 megabytes. Hence, applying the filter language can save memory.

Table 6.1: The memory usage (megabytes) in different stages of running programs with different graph sizes in DrAST.

AST size (number of nodes)	**2 784**	**9 610**	**50 101**	**109 971**	**170 012**
Compiler (MB)	3	5	18	39	59
Model (MB)	9	19	84	180	275
Filtered model (MB)	9	16	45	94	140
GUI (MB)	33	53	178	364	544
Filtered model, GUI (MB)	31	38	66	114	157

Figure 6.3 uses the data from Table 6.1 to exemplify the memory usage of the different parts of DrAST. From top to bottom we see memory usage of the GUI, the filtered tree, the reflected tree and lastly the compiler. No filtering was applied for the case in the figure, in other words all nodes were displayed. We see that the reflected tree is almost the same size as the compiler, the difference are some caches that use some extra memory. The filtered tree section consists of both the reflected tree and the AST from the FCL compiler.

In this case all nodes pass the filter, resulting in that the filtered tree's memory usage is approximately the same as the reflected tree. The GUI uses a lot of memory compared to the other parts. Some memory section of the GUI is based on the filtered tree and by adding a more restrictive filter one can lower the memory impact of both the GUI and the filtered tree.

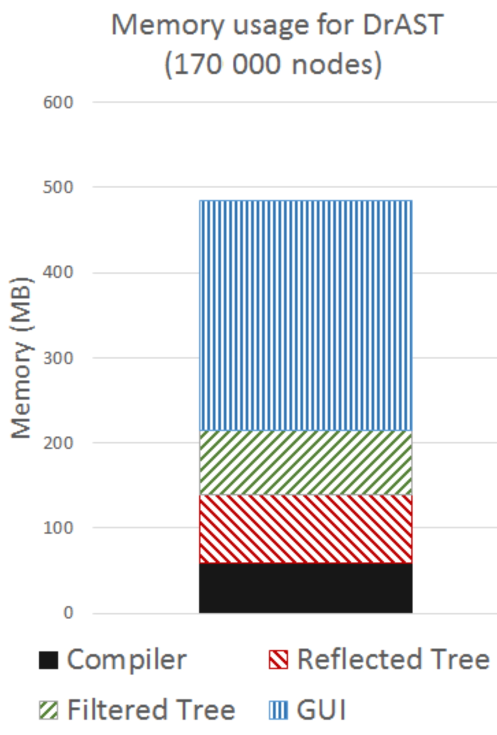


Figure 6.3: The memory usage in different parts of DrAST running with an AST of 170 000 nodes.

6.2.3 Input lag

The GUI must appear to be responsive and not suffer from input lag, otherwise the overall impression of DrAST will suffer. The heavy part of the GUI is the graph view that must render all nodes, edges and labels onto the screen. The GUI needs to render the graph every time the user wants to zoom, pan or highlight a node. The graph view is however on its own thread so even if it is lagging behind, the rest of the program is still responsive. To evaluate DrAST regarding lag performance we tested the number of times the tool can render a graph per second, also known as frames per second (FPS).

We created a script that rendered graphs as many times as it could during 10 seconds. Rendering of nodes was only done with edges and nodes remaining hidden, like they are while a user navigates the graph view in DrAST. This was done for 13 programs with different AST sizes, ranging from around 2 700 to 170 000 nodes. The result is presented in Figure 6.4, where the bars represent the average FPS for each program. Note that the values on the x-axis only are in size order and that the scale is not linear.

J. Nielsen (1993) showed that a program appears to react instantaneously when it has an update rate above 10 FPS [20]. As seen in Figure 6.4, the GUI of DrAST performs at this level for programs with less than 30 000 nodes. When the FPS is between 10 and 1 a user will notice a lag but will not lose its interest [20]. In our tests the GUI performance approached the lower limit of 1 FPS for graphs of 170 000 nodes, but never dropped below it (Figure 6.4). We also want to highlight that by using the filter mechanism the lag can be decreased as only nodes the user actually is interested in are included.

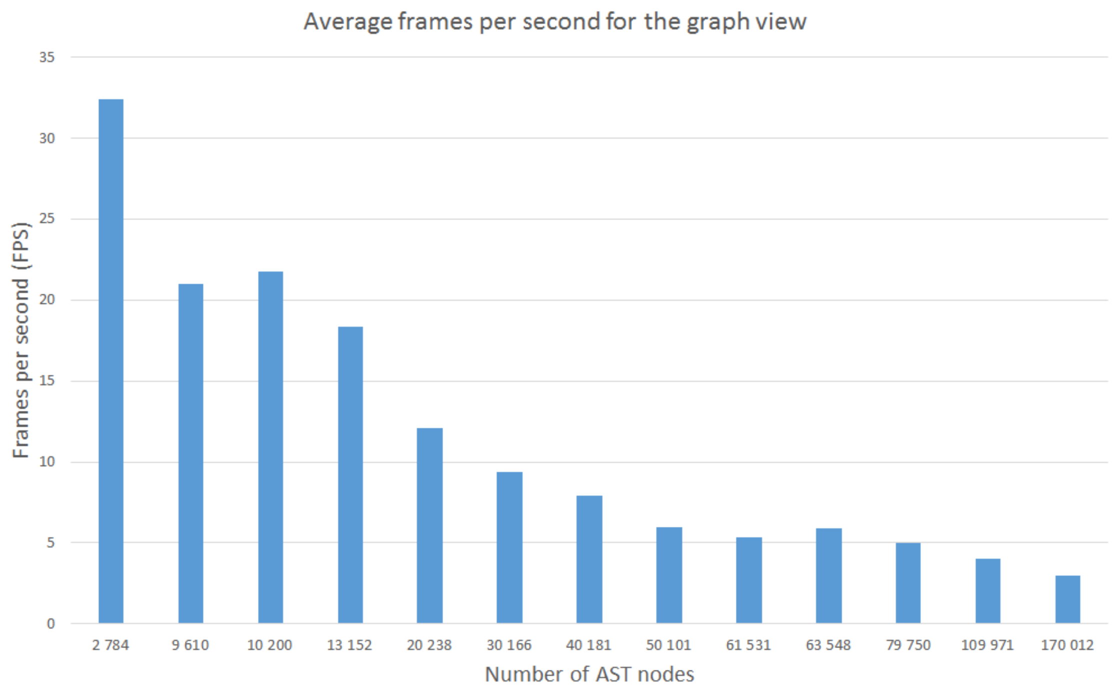


Figure 6.4: The average frames per second of several renderings of the *graph view* of 13 programs of differing AST sizes (number of nodes).

If a node has a large amount of children, we noticed a great loss of performance. This is connected to the edges that must calculate the length and angle between two nodes. The children of a node will all be placed next to each other in a row, with the parent placed above in the middle of the row and this result in a lot of computations with large numbers for the children farthest away from the parent. This problem could make a more restrictive filter perform worse, because a cluster node can get a lot of children. Our quick solution to this is to deactivate edges in the GUI, so they will not be rendered.

6.2.4 Conclusions from performance tests

The performance goal (1.2.1) was to create a stable tool that performs its computations within an acceptable time frame, for both small and large projects. We can conclude from our studies that this goal has been achieved for ASTs with up to 170 000 nodes. In the worst case DrAST is able to create its model in 1.2 seconds, and can reapply a filter in about 0.35 seconds. DrAST's GUI is able to visualize large ASTs. However, the FPS drops when displaying a large amount of nodes but with FCL it is possible to remove the vast majority of nodes, down to a number that is manageable for the user.

Chapter 7
Related work

Today there exist a number of different tools used for debugging, and in this chapter we will compare DrAST to some of these solutions. Here we will discuss different methods for debugging attribute grammars and methods for extracting the AST information.

7.1 Attribute debuggers

An attribute grammar debugger can have many features, although the main feature usually is to inspect the current values of attributes. DrAST debugs a program by visualizing the AST at some state. With DrAST, one can inspect attributes, nodes, edges and references, as well as invoke parameterized attributes. There are some things DrAST cannot do. Some examples of missing features are analysing of the grammar, see if the AST structure follows the grammar or if attribute values correct or not. This is partially solved by the internal structure of DrAST that allows the developer to add its own analyse modules. Currently no such analyses are implemented in DrAST though.

In this section we will broaden our perspective on attribute debugging, by also looking at other tools and algorithms.

7.1.1 Visualization

The visualization of an AST structure can be done in many ways, and today there are a number of different ways to visualize graphs. For example exporting the AST to a file (DOT, GraphML or JSON format) and use a program that can read these files. VAST is an example of this [1]. The idea here is that the developer generates one output file from their parser and then opens this file in VAST. VAST displays to the user an interactive visualization of the AST, a disadvantage in this approach is that this structure is not connected to the compiler. However VAST is good for cases when the visualization of the current AST is all you want, where the output file can be stored and used later. Although adding analysing tools to these files could prove hard. For example computing attributes with parameters on demand is not possible. This is one of DrAST's strengths when compared to external file solutions, as it is integrated on a deeper level and can also be extended more easily (Although only for JastAdd based systems).

The Eclipse plugin AST View [24] is another example of an AST visualizer. It shows the AST for a Java file in a view similar to the *text-tree view* in DrAST. However, AST View does not support attributes and thus it is not possible to compute these on demand, which is a disadvantage compare to DrAST. Also, it can only display the AST for a single Java file as where DrAST can visualize the complete ASTs from programs with different compilers.

Aki [13] is another example of a visualization tool but in contrast to VAST, is built into the compiler and has full access to grammar and attribute-definitions. Aki is closely connected to the specific task, even more than DrAST, and must has access to a lot of information about the syntax to function. Aki has some features that DrAST is lacking though, which will be explained in the next section.

One strength that DrAST also have compared to these three is its filtering capabilities. VAST shows everything that is stored in a file, AST View shows the AST for a single Java file, and AKI shows the complete AST of the compiled program. For large trees or files this will become a problem regarding both performance and usability, where it will be hard for the user to navigate and find the point of interest.

7.1.2 Grammar debugging

For debugging attribute grammars one can also use algorithmic debugging [13]. The general idea of these algorithms is to help the user localize errors in their program. The algorithms are based on user input. The user get to follow the evaluation of one or more attributes through the AST, and see its value in each node up to its final value. The idea is that the user tells the program if a value is correct or not. If the Value is correct then we know that the problem should not be in this part of the AST, and we can look somewhere else. These methods require that the user knows what the value of solution should be and that it know the values in between. AKI mentioned in the last section has this feature. DrAST has no way to follow the evaluation of the attributes in the grammars, due to that reflection can only invoke methods but cannot access the internal computations.

7.2 Heap snapshot

Compared to many other debuggers DrAST work on a high level, and visualize only data of one state compare too many other step-by-step debuggers. We do not want to debug the java code in itself, but rather one structure in the program. In other words, only the objects representing nodes and methods representing edges, attributes and tokens. To use DrAST we force the user to alter their code to make certain information visible, DrAST then uses Java reflection to automatically extract the needed information.

Another solution to extract the AST from a program could be by following the theory used in the article Visualizing Memory Graphs [28]. The authors talk about debugging memory and visualizing it as a graph. Pointers and references in the stack or heap memory could be represented as edges and objects as nodes. This theory has been used in a number of different tools [12, 15, 22] and is intuitive with object oriented languages where relationships between objects often are represented as graphs. One method for creating this kind of memory graphs is called Heap snapshot. A snapshot of the heap could be used to extract the AST objects and we will discuss methods for this in this section.

7.2.1 Examples

Some examples of tools using heap snapshots is Heapviz [15] and Fox [22]. For any state of the program these tools are able to take a snapshot of the debugged program's heap. This data is used to produce the object graph for this state. In the heap one can access all objects surrounding a java program, from external libraries and java libraries to the source code itself. The heap data is therefore huge, even for small programs, and grows quickly with bigger programs. The graphs get cluttered quickly so some kind of filtering is needed. Especially for DrAST who are only interested in one particular subset of objects, the ones in the AST.

Heapviz apply a summarization algorithm [15] to group objects of the same type into something similar to our clusters. So a list of objects becomes one node in their graph. This method could be used by DrAST to extract the AST from the snapshot, by finding the set of nodes representing the AST. Although it still might prove difficult, due to that there is no simple way to specify the information or nodes DrAST need. Fox on the other hand use a two-step filtering [22]. The first step is to only let objects fulfilling certain criteria (e.g. depth in graph or class type) be part of the snapshot. The second step is user based. The user constructs SQL-like queries to fetch the data from the constructed snapshot. Something like this would probably work better for DrAST, compared to Heapviz, with the purpose to extract the AST information.

7.2.2 Getting the snapshot

Normally when you take a heap snapshot in Java you take a heap dump. This takes the current heap and dumps it to a file. In this process you lose your references to all objects in the JVM, and will not be able to invoke the methods that represent attributes and tokens. Both Heapviz and Fox are tools for viewing the current status in the heap, and need no references to the Java Objects. DrAST however support additional features, and are not only interested in viewing current information that can be derived from a snapshot file. DrAST is also able to compute attributes with or without parameters on demand. Without the references a heap snapshot is not enough for these on-demand computations. One approach to solve this is to calculate all attributes before the heap snapshot is taken. The main issue with this is that it would obviously slow things down and make the heap grow substantially, especially for large ASTs with a great number of attributes in the nodes. To use the snapshot approach, we would need a way to directly connect to the JVM and take a snapshot and also save all real Java objects during runtime. We are not quite sure if this is feasible, so further studies are required.

7.2.3 Comparison

Using a heap snapshot method would make the integration with the debugger easier, compared to the current solution. There would be no need to alter source code, because all data is fetched from the heap memory. We would also be able to receive extensive data about the compiler, not just the AST objects and at any point during runtime. Currently DrAST have no clue what is happening inside method/attribute calls, and cannot break the calculation without altering the source code. In theory one can compare two snapshots to see the evaluation of some attribute and changes in different nodes, however this would not be an efficient solution.

One problem that would arise with the use of heap snapshots compared to reflection is that we would not know which objects that represent the AST. DrAST should work for any JastAdd grammar, and therefore we do not have any information about which classes that are the nodes in the AST beforehand. We do not know if ArrayList and String are nodes of the AST. With our solution we get the root object, and use JastAdd annotations to separate the AST classes from other classes. With heap snapshots DrAST would need to have more information to know what classes are AST nodes, some information about a parent class that all AST nodes have in common, or a package name.

The heap memory consists of large amounts of data and could be used by many different applications, but we are interested in a specific subset. The majority of the heap data could therefore be considered junk, and would be discarded. Focusing on only the AST Objects directly, like our solution, probably gives a better performance in time and memory compared to heap snapshots.

7.3 Java remote debugging

JAVIS [21] is an example of a tool that is quite similar to DrAST, but uses a different method than reflection and annotations to gather its data. According to their paper, JAVIS was created to help new students to understand the structure of Java programs, where it visualizes how a program’s different objects refer to each other. As an example it can show how a linked list has a reference to the first and last element of the list, and each element points to the next. JAVIS uses the Java Debug Interface (JDI) to extract the data, the JDI is part of the Java Platform Debugger Architecture (JPDA) [8]. This architecture lets one application run and debug another application.

7.3.1 Comparison

Using Java remote debugging to debug would hold some similar pros and cons as heap snapshot. We would not need to change the source code and we could step through the code like a normal debugger. This method would most probably perform better than reading the heap. As with the heap snapshot, this method would give us access to all classes in the debugged program and not just the ones we are interested in. We would need a way to know which class objects that are part of the AST.

Compared to our current solution JDI does not let us access the real objects of the tree, rather just a proxy called ObjectReference [8]. However it would still be able to reach the values of fields and invoke methods. DrAST as in the moment of this writing would not have a problem with this restriction. Future features in DrAST could however include changing the AST and this would need the real AST-node objects.

Chapter 8
Concluding discussion

This chapter concludes this report. A summary of the key results of the project is presented and some ideas for future work of DrAST.

8.1 Summary

This paper presents a solution to debugging compilers that use ASTs as internal structures, as such solutions to our knowledge previously have been lacking. The solution has been implemented in a tool called DrAST. The main feature of DrAST is visualizing a representation of the AST to the user in an interactive way. We call the representation the filtered tree, due to that the user can hide nodes with the help of a filter. The nodes that do not pass the filter are gathered in nodes we call clusters, which can contain any number of filtered nodes. The purpose of the cluster nodes is to hint on how the original structure of the AST looks and they help the user by removing unnecessary information, i.e nodes that did not pass the filter.

DrAST was implemented in Java for the JastAdd system. JastAdd is an attribute grammar based system that can be used to generate compilers and analysing tools. It generates Java classes and each method can represent attributes or edges in an AST. The methods are also annotated with Java annotations. DrAST uses Java reflection on the root-node object of the AST and the JastAdd annotations to create its own representation of the AST that can be filtered and displayed to the user.

The tool has been tested for its usability by 6 participants of varying experience in JastAdd development. Their common background is a compiler course at the faculty of engineering at Lund's university. Their feedback was used to improve and add features to DrAST in order to make the user experience better. Each participant gave a positive response, and thought the tool would be useful for both the compiler course and more advanced JastAdd projects.

A number of different performance tests were conducted on DrAST. The primary metrics for the tests were the following; Process time, memory usage and input lag (measured in FPS). The FPS and process time is dependent on the machines, although it is possible to create a filtered tree with 200 000 nodes under 3 seconds on everyday machines. Navigation in the graph works well, up to around 40 000 nodes. The idea is though that the user should filter the tree to something smaller than this. For normal usage, we expect the user to have between 20 to 50 visible nodes, and as seen in Figure 6.4 it takes less than 0.03 seconds to draw the filtered tree with roughly 2 800 nodes present in the graph. This is well below the limit of 0.1 seconds recommended by user interface experts [20].

In the end we believe that our goals, usability, performance and scalability, have been reached.

8.2 Future work

The DrAST system was implemented with support for continued development in mind. The model-view architecture makes it easy to create new views that can visualize or analyse the reflected or the filtered tree. An analyser interface can be extended to easily add analysis during or after the construction of the reflected or filtered tree. The reflection part of the model that is used to fetch attributes and edges of each node is its own separate module, and could be replaced by something else without making changes to the rest of the model. The overall structure of the GUI is based on abstract classes and interfaces, to help developers add their own buttons, dialogs or windows. In this section we will present some ideas we have, that can be extensions to DrAST.

8.2.1 Save the model to a file

The model is separated from the views so that new views can be added. We have a view that creates a simple XML file that only store the nodes but no attributes. One could think of many other views that do a similar task but more complex, and to different formats. DOT, GraphML or JSON could be some formats that would be useful for different uses. By extending this feature, the DrAST system could be used to export specific information about an AST to any other software.

8.2.2 Java remote debugging API

The current solution requires the developer to change the compiler source code. Instead, one could use a normal debugging interface (example Java remote debugging), that would allow developers to set break points, and from there start DrAST. The Java remote debugging API could then also replace the reflection part of the model. By doing this, the objects used to create the reflected and filtered tree would instead be extracted from the reflection API. The Java remote debugging API could also be used to add support for the evaluation of attributes, to see intermediate values and nodes that create the final value.

8.2.3 Optimizations

One optimization could be to change the filtering to only affect relevant nodes, not the whole tree. The idea is to inspect the difference between two filters and only reevaluate the affected nodes. Presently the model discards the whole filtered tree and creates a new one when applying a new filter. Instead one could keep the current filtered tree and alter it according to the new filter, which could improve the time it takes for the model to create the filtered tree.

8.2.4 Analyzes

Analyser is an interface in DrAST that executes a method during or after the creation of the reflected or the filtered tree. The current state of the trees or their nodes is passed into an analyser object, in which DrAST can perform whatever analysis the developer needs.

8.2.5 Graphical interface for FCL

Currently in DrAST's GUI the filter language is handled through a text editor. One could later on replace this with a GUI of some sort to improve DrAST's usability, for example with edit boxes, spinners and buttons.

Compare tree to grammar

DrAST does not know about the grammar of the AST that it performs reflection on. Adding an analyser to the project that knows the grammar would be a solution to this. This analyser could be called during the creation of the filtered tree, and for each node see if it has the parent of the correct class (because the children have yet to be created in the filtered tree). This could of course be done after the filtering, but this would mean another iteration of the whole tree. Note that this analysis should be done on the reflected tree, which is a direct representation of the AST.

JastAdd annotations would make it possible to add a general grammar analyser for all JastAdd compilers. Every class generated by JastAdd could be annotated with the defining grammar. This way our tool could rebuild the grammar, or at least the part of the grammar that are currently in use by the AST represented.

Find equations for inherited attributes

Our solution does not help the user to see how an attribute was calculated, because Java reflection only allows us to invoke methods. We cannot see the internal structure of a method with reflection. However, with the information we get from JastAdd annotations, we can see if an attribute is inherited or synthesized. DrAST could, for inherited values, search upward in the tree until it finds a synthesized attribute with the same name. This node could then be marked in a way so the user easily finds it. Of course, we still do not know what the attribute does internally. A synthesized attribute that use other attributes to compute its value will still be hidden from us.

Bibliography

[1] Urguiza-Fuentes J. Velazquez-Iturbide J.A. Almeida-Martinez, F.J. Vast: Visualization of abstractsyntax trees within language processors courses. *Proceedings of the 4th ACM Symposium on Software Visualization, SoftVis '08*, pages 209–210, 2008.

[2] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in Java.* New York : Cambridge University Press, 2002, 2002.

[3] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70 – 77, 1999.

[4] Oracle Corporation. The java tutorials,lesson: Getting started with swing. Online; accessed October-2015. `https://docs.oracle.com/javase/tutorial/uiswing/start/index.html`.

[5] Oracle Corporation. Javafx: Getting started with javafx, 2014. Online; accessed October-2015. `http://docs.oracle.com/javase/8/javafx/get-started-tutorial/title.htm`.

[6] Oracle Corporation. The java tutorials, trail: The reflection api, 2015. Online; accessed October-2015. `https://docs.oracle.com/javase/tutorial/reflect/`.

[7] Oracle Corporation. The java tutorials, understanding extension class loading, 2015. Online; accessed januari-2016. `http://docs.oracle.com/javase/tutorial/ext/basics/load.html`.

[8] Oracle Corporation. Java™ platform debugger architecture (jpda), 2015. Online; accessed december-2015. `http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/`.

[9] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[10] Martin Fowler and Jim Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[11] Görel Hedin, LTH Institutioner vid LTH Datavetenskap Publisher Lunds universitet, Lunds tekniska högskola, and LTH at Lund University Departments at LTH Computer Science Publisher Lund University, Faculty of Engineering. An introductory tutorial on jastadd attribute grammars. *Generative and Transformational Techniques in Software Engineering III / Lecture notes in computer science*, page 166, 2011.

[12] T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. *Proceedings 37th International Conference on Technology of Object-Oriented Languages |& Systems TOOLS-Pacific 2000*, page 202, 2000.

[13] Yohei Ikezoe, Akira Sasaki, Yoshiki Ohshima, Ken Wakita, and Masataka Sassa. Systematic debugging of attribute grammars. 2000.

[14] Jung. Java universal network/graph framework. Online; accessed October-2015. `http://jung.sourceforge.net/`.

[15] S. Kelley, E. Aftandilian, C. Gramazio, N. Ricci, S.L. Su, and S.Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. *Information Visualization*, 12(2):163–177, 2013.

[16] Donald Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127, 1968.

[17] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[18] S. Lauesen. *User Interface Design: A Software Engineering Perspective*. Pearson Education, 2005.

[19] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. *Acm sigplan notices*, 33(10):36–44, 1998.

[20] Jakob Nielsen. Response times: The 3 important limits. *Usability Engineering*, 1993.

[21] R Oechsle and T Schmitt. Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi). *SOFTWARE VISUALIZATION*, 2269:176 – 190, 2002.

[22] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. *Proceedings of the 2004 Australian Software Engineering Conference. Proceedings*, page 251, 2004.

[23] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. pages 336–, 1996.

[24] Eclipse JDT UI team. org.eclipse.jdt.astview - ast view, 2014. Online; accessed januari-2015. `http://www.eclipse.org/jdt/ui/astview/index.php`.

[25] The JastAdd Team. Jastadd concept overview. Online; accessed October-2015. `http://jastadd.org/web/documentation/concept-overview.php`.

[26] Johan Thorsberg and Joel Lindholm. Filtering the ast. `https://bitbucket.org/jastadd/drast/wiki/The%20Filter%20Configuration%20Language`.

[27] Andreas Zeller. *Why programs fail. : a guide to systematic debugging*. San Francisco, Calif. : Morgan Kaufmann ; Oxford : Elsevier Science, 2009., 2009.

[28] T Zimmermann and A Zeller. Visualizing memory graphs. *SOFTWARE VISUALIZATION*, 2269:191 – 204, 2002.

Appendices

Appendix A
The abstract grammar for FCL

```
NodeFilter ::= <ClassName:String>;
DebuggerConfig ::= Configs:FilterConfig Filter*;
FilterConfig ::= [Use] Binding*;
Filter ::= FilterName:IdDecl Nodes:Node*;
Node ::= ClassName:LangDecl NodeConfig*;
abstract NodeConfig;
When : NodeConfig ::= Expr*;
SubTree : NodeConfig ::= Expr*;
Style : NodeConfig ::= Binding*;
Show : NodeConfig ::= LangDecl*;
Use ::= IdDecl*;
Binding ::= Name:IdDecl Value;
IdDecl ::= <ID:String>;
abstract Expr;
Shown : Expr;
ChildOf : Expr ::= Not:Bool LangDecl;
ParentOf : Expr ::= Not:Bool LangDecl;
abstract BinExpr : Expr ::= Left:Value Right:Value;
In : BinExpr;
NotIn : BinExpr;
EQ : BinExpr;
LT : BinExpr;
GT : BinExpr;
NEQ : BinExpr;
LEQ : BinExpr;
GEQ : BinExpr;
abstract Value;
Num : Value ::= <NUMERAL>;
Str : Value ::= <String>;
Color : Value ::= <Color>;
StrArray : Value ::= Str*;
NumArray : Value ::= Num*;
EmptyArray : Value;
LangDecl : Value ::= <ID:String>;
Bool : Value ::= <BOOL>;
OnOff : Value ::= <ONOFF>;
```

Figure A.1: The abstract grammar for the filter configuration language (FCL).

Appendix B
Usability Tasks

Here are the 9 tasks the participant had to perform during the usability test. These were spoken aloud to the participant, so they can be instructions to the reader and not the participant.

- Open DrAST and let the participant think aloud. Let the participant *click around* in the GUI meanwhile they explain their actions.
- Take a screenshot/picture of the graph.
- The root of the AST has an attribute nodeCount. What is the value of this attribute?
- Iduse has a parameterized attribute called lookup(String arg). This attribute tries to find the IdDecl that has an attribute getID wiht a value that corresponds to the parameter arg. Find an IdUse in the AST and find its IdDecl with the lookup attribute.
- The Program node has a parameterized attribute called getIdDecl(IdUse use). This attribute receives an IdUse and returns its IdDecl. Use this attribute with the same IdUse used in the last task, and control that it returns with the same IdDecl as before.
- Create a filter that hides all nodes except Program, Let, IdDecl.
- IdUse and some other node types inherite a superclass. Create and apply a filter that only shows nodes that inherit this superclass.
- Change the last filter so that only Div and IdUse nodes are visible. Then change the filter so that IdUse's that are a direct childr of a Div node are visible. Also, make a last adjustment so that only the IdUse's that have an attribute getID with the value *a* are visible.
- IdUse has an attribute decl that points to an IdDecl somewhere in the three. Draw, for all IdUse:es, these reference edges directly in the graph.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-03-18

EXAMENSARBETE DrAST – An attribute debugger for JastAdd
STUDENT Joel Lindholm, Johan Thorsberg
HANDLEDARE Görel Hedin (LTH)
EXAMINATOR Ulf Asklund

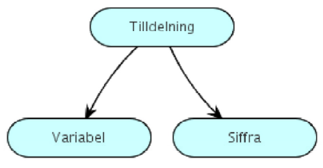
Förstå din kompilator med DrAST

POPULÄRVETENSKAPLIG SAMMANFATTNING **Joel Lindholm, Johan Thorsberg**

Verktyget DrAST har skapats för att enklare förstå hur en kompilator faktiskt fungerar, den där magiska lådan som översätter kod till något datorn begriper. DrAST är en debugger, som illustrera hur kompilatorer ser ut på insidan.

En titt in i kompilatorn
Den kod som programmeraren skriver är faktiskt bara text och inget annat. Datorn förstår dock inte text, utan den måste översättas till maskinkod med hjälp av en kompilator. Kompilatorn måste först kontrollera att koden följer vissa regler. Vi kan jämföra detta med en grammatikgranskare av svenska språket. Om någon skriver ”grisar hoppar sängar ” ska kompilatorns analys förstå att språkreglerna inte följs.

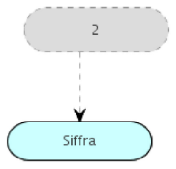
En AST (Abstract Syntax Tree) är en trädstruktur som ofta används för att utföra dessa analyser. Låt oss kolla på ett exempelprogram. Koden ”a = 2” betyder att ”a” är en låda som innehåller en 2:a. En kompilator kommer att ta de viktigaste delarna i koden för att bygga en trädstruktur med noder (Figur 1). Tilldelning (likhetstecken) pekar på två saker: Variabel (lådan a) och Siffra (innehållet 2). Om en regel säger att Tilldelning ska peka på en Siffra och en Variabel, kan vi enkelt se om det stämmer i vårt träd.



Figur 1: En AST som beskriver ett koden ”a=2”

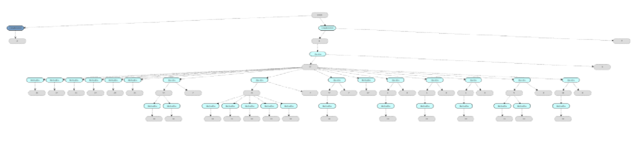
Problemet och lösningen
Till skillnad från Figur 1 så består ett program ofta av tusentals noder, och en överblick kan vara svårt att få utan hjälp. Det finns dock få hjälpmedel idag. Vårt nya verktyg DrAST är skapat för att fylla en del av detta tomrum för utvecklare av kompilatorer. DrAST kan nämligen ”plocka ut” en AST från en kompilator och sedan illustrera den grafiskt för programmeraren.

Filtrering
DrAST har också andra egenskaper, än själva visualiseringen av en AST, som förenklar arbetet med kompilatorer. Som vi nämnt ovan kan ett program bestå av många noder. För att kunna fokusera på det intressanta i trädet kan det filtreras, med hjälp av regler som beskriver vilka noder som faktiskt ska visas. De noder som inte uppfyller alla reglerna gömmer DrAST helt enkelt. Figur 2 visar samma träd som Figur 1, men bara noder av typen Siffra visas.



Figur 2: Samma AST som i figur 1 men det är några noder som göms undan.

Vem kan ha nytta av DrAST?
Målet är att vårt verktyg ska kunna användas av många, allt från studenter som lär sig kompilatorteknik till professionella som forskare och företag. Som sagt så hoppas vi att DrAST kan hjälpa till att fylla ett tomrum som funnits allt för länge.



Figur 3: En AST med över 60 000 noder som har filtrerats så den bara visar metoder för en specifik Java klass.