MASTER'S THESIS | LUND UNIVERSITY 2018

Web-based Tree Editor for JastAdd Compilers

Marcus Lacerda

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-19

Web-based Tree Editor for JastAdd Compilers

Marcus Lacerda
dat11mla@student.lu.se

December 14, 2018

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

Supervisor: Niklas Fors, niklas.fors@cs.lth.se
Examiner: Görel Hedin, gorel@cs.lth.se

Abstract

Domain Specific Languages (DSL) are programming languages created with specific domains in mind. Programs in these domains may be more easily represented as graphs or other structures, rather than text. Structure editors do just that, represent programs as, for example, graphs (graphical editors) or trees (tree editors), by using the programs' underlying structures, making structure editors useful tools for DSL programming. We will in this thesis present a generic web-based tree editor that works for creating programs in any language specified in the metacompilation system JastAdd. The tree editor performs semantic analysis and displays semantic errors directly in the web browser.

Keywords: compilers, abstract syntax trees, tree editors, JastAdd

Acknowledgements

I would like to thank my supervisor Niklas Fors for all his invaluable feedback and support during the course of this thesis. Furthermore, my thanks go to Görel Hedin, for her enthusiasm regarding the project. Also, thanks to both Niklas and Görel, for providing me with this very interesting research topic. Lastly, thanks to Alfred Åkesson, for his council in the field of tree editors.

Contents

Chapter 1
Introduction

There exist many different programming languages today, some of which are general and used for almost any purpose and others that are created for specific problem-domains. Therefore, there exists a distinction between programming languages: *general purpose languages* (GPL) and *domain specific languages* (DSL). GPLs are languages such as Java, C and python. These are created to allow the programmer to write any program, but, as you may have encountered, some problems are quite difficult — or at least tideous — to solve with a GPL. Compare for example the complexity of using Java to make matrix calculations to making those same calculations in a specialized language such as Matlab. This is the prupose of DSLs: to facilitate solving problems in specific domains. For certain domains however, something that in the context of GPLs seem quite natural, writing programs as text can seem unecessarily complicated. For such languages an editing tool that uses a perspective with emphasis on the solution and not the implementation, the program and not the source code, can be advantageous.[9] Editing tools that use a program's underlying structure are generally called structured editors. For example: a structured editor for building finite state automata might display a state diagram instead of showing source code. What we are interested in is a specific kind of structured editor, namely a tree editor.

A tree editor enables a user to edit programs as abstract syntax trees (AST) which is a compiler's internal representation of a program. This kind of editor is generic in the sense that it can work with any compiler, as long as it can access and build ASTs for that compiler. This gives us a broad structural editing tool and it also puts less demands on its associated compilers: the compiler will not need a parses since the ASTs are built directly. This also means that the corresponding programming language does not need to have a syntax defined. Tree editors are therefore well suited for compiler development in that it both allows users to build ASTs, which is a familiar structure and used by all compilers, and use the compiler at a very early stage in development. But, for compiler development you also need a something to build the actual compilers. This is what the metacompilation system JastAdd is for.

Through JastAdd's declarative language one can specify the abstract syntax, semantic

analysis and synthesis of a compiler[4]. The abstract syntax is what specifies the structure of the compiler's ASTs. The semantic analysis is how the compiler should interpret the ASTs — what they mean in a sense — and the synthesis is generation of some target code. The JastAdd compilers are generated in Java. With JastAdd we can generate compilers, and by extension, new programming languages.

In this thesis we present a web-based tree editor for facilitating the analysis and creation of JastAdd compilers. The editor servers as a proof of concept showing how it can be built, its complexity, limits and usefulness.

In the following chapter we will explain some key concepts necessary to understand this master's thesis. After that, an overview of the system built, followed by its implementation. We will then evaluate the performance of our system and lastly discuss future improvements.

Chapter 2
Background

This chapter explains some key concepts necessary to understand this master's thesis. We will start with a brief overview of what a compiler is and how it works followed by a description of how they can be built through the metacompilation system JastAdd. Lastly, some key differences between Java and JavaScript will be examined.

2.1 Compiler Overview

A compiler is a program that translates source code written in a programming language into another form called target code. This process is known as compiling and usually consists of two main parts, analysis and synthesis. The former being when the source code is processed and the latter when target code is generated. Figure 2.1 provides a simplified overview of any generic compiler.

The compiler's analysis usually consists of three distinct types of analysis: lexical, syntactical and semantic. But, for our purposes we can view the lexical and syntactical analysis as one step, *parsing*. Parsing is the process of examining the source code and creating an internal representation describing the program. This representation is called an *abstract syntax tree* and is in a sense the core of the compiling process. After an AST

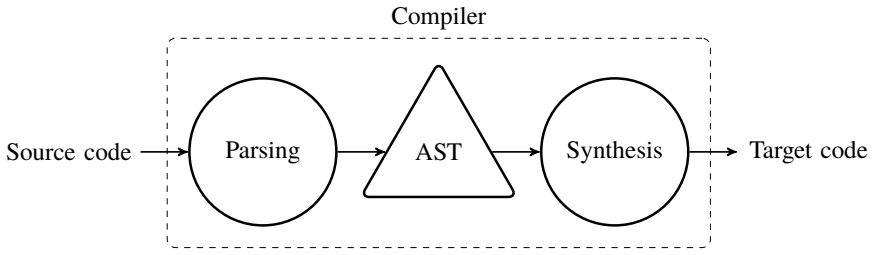

Figure 2.1: Simplified overview of a generic compiler.

has been created, the semantic analysis is performed. This is when expressions, variables and functions are bound to some sort of meaning. For example, that a variable has been declared before use. This information can be encode in different ways, e.g., symbol tables and attributed ASTs.[1]

After the semantic analysis has been completed the synthesis process can begin which consists of generating the actual target code. This thesis will focus mainly on abstract syntax trees and semantic analysis.

2.1.1 Types of Compilers

Compilers, as mentioned above, translate source code into target code. The target code is usually represented in either intermediate or executable code. Examples of such compilers are javac, which translates a program written in the Java language into Java-bytecode (intermediate code which is run on a Java Virtual Machine) and GCC, a compiler for the language C, which takes a C program and returns executable code. Although these are the most common types of compilers, there may be situations when you want to translate source code from one programming language into source code in another language, as is the case in this thesis project. This kind of compilers are called source-to-source compilers or *transpilers* and further on in this report we will explain how a transpiler was used to compile source code from Java to JavaScript.

2.1.2 Abstract Syntax Tree

The thesis work described and discussed in this report will reference abstract syntax trees. Therefore, it is important to give a basic, but detailed, description of what an AST is. It can be seen as the interface between the parsing process and the following steps of the compilation (semantic analysis and synthesis). The nodes of the abstract syntax tree are specified in the *abstract syntax*. Usually, each of the AST's nodes represent a construct in the programming language. The AST itself is usually built based on the result of the parsing, a parse tree, but as will be shown in this thesis, ASTs can also be created directly.

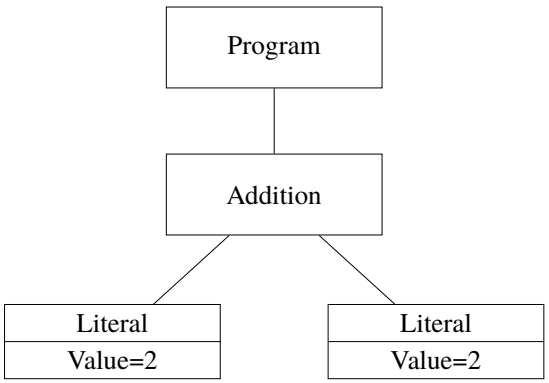

Figure 2.2: Example AST representing the expression 2 + 2.

We will clarify with an example. Say we have a toy programming language made for creating simple mathematical expressions. In this language we have the following constructs: Program, Expression, Addition and Literal. We define the language so that a

Program-node is the root-node of the AST and has an Expression-node as its only child. We also define Addition and Literal to be of the type Expression, where an Addition-node can have two Expression-nodes as its children, and a Literal-node only has a token representing its value. In this language, we can create the tree seen in Figure 2.2, where the root-node of the AST is a Program-node having a child expression of the type Addition. This child in turn has two children of its own, each a Literal with the value 2. The resulting AST describes the mathematical expression 2 + 2.

2.2 Tree Editor

An editor in this context is a tool for editing programs in some given programming language. A text editor allows a user to edit the source code of a program. Whereas a tree editor, allows a user to edit the abstract syntax trees of a program directly, see Figure 2.3.

So where a text editor has to pass its data (in the form of text) to a compiler which then parses the code and builds an AST. The tree editor instead allows you to skip the parsing since the data you are modifying is an AST already. This means that a compiler used with a tree editor does not need to have a parser, which makes it easier to create new languages. Allowing the creation of abstract syntax, semantical properties and even synthesis before constructing a parser.

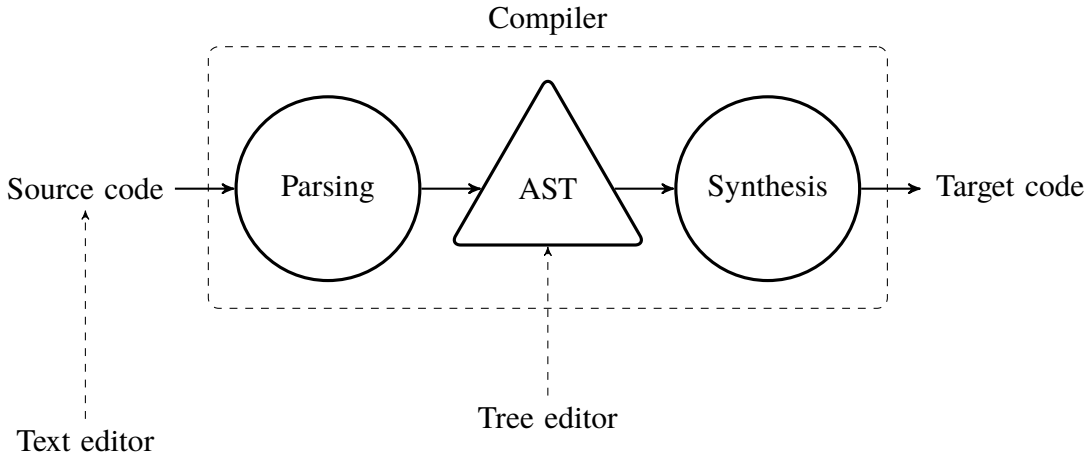

Figure 2.3: With a text editor the user modifies the source code. With a tree editor the user modifies the abstract syntax tree (AST).

Some drawbacks however, include that the format of the code in the tree editor is not as uniform as that of a text editor. Code in text editors are usually written in the Latin alphabet whereas there are many different formats for representing abstract syntax trees, e.g. XML or JSON. One could also assume that people are not generally as used to seeing programs in the context of syntax trees as they are seeing text.

An example tree editor is the JastAdd tunable tree editor called JATTE. The picture shown in Figure 2.4 demonstrates how JATTE displays the abstract syntax tree from the example in Figure 2.2.

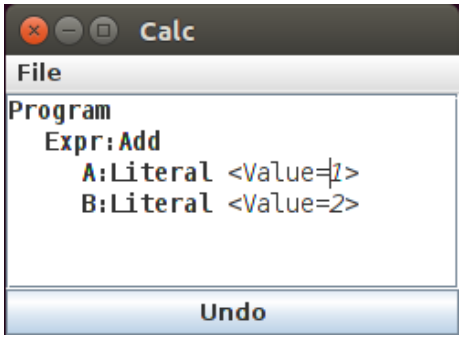

Figure 2.4: Example of how the tree editor JATTE shows the AST from Figure 2.2.

2.3 JastAdd

JastAdd[1] is a system for generating language-based tools based on attribute grammars[4], for example compilers, which is what we will use it for in this project. JastAdd takes three file types ".ast", ".jrag" and ".jadd" and generates a compiler which consists of Java-classes, each representing a node in the AST, see Figure 2.5 for an overview. The ".ast" files contain the abstract syntax, which specifies the structure of the AST-nodes.[2] The ".jrag" and ".jadd" files are for declaring the semantic analysis and synthesis of the language. A bit simplified one could say that the former is for adding declarative attributes defined by equations and the latter for adding imperative methods. So where the abstract syntax in the ".ast" files describe the AST's structure, the other files describe the nodes' attributes and methods, in other words, how to create the attributed AST (and later synthesis). The process of creating the attributed AST is, as the observant reader might recall, the result of the semantic analysis process.

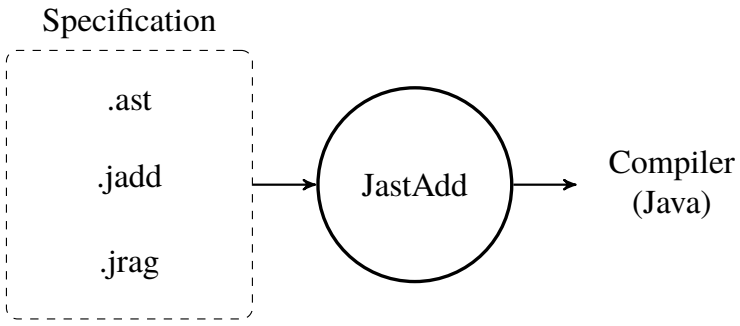

Figure 2.5: JastAdd takes specification files and generates a compiler.

To clarify the concept of attributed abstract syntax trees, we will go back to the toy example from Figure 2.2. This time we will add an attribute to the Expression type called "eval" which will represent the evaluation of an expression. Each subtype of Expression will now have an "eval" attribute, representing how that specific expression should be

[1] http://jastadd.org/web/
[2] The nodes of the abstract syntax tree.

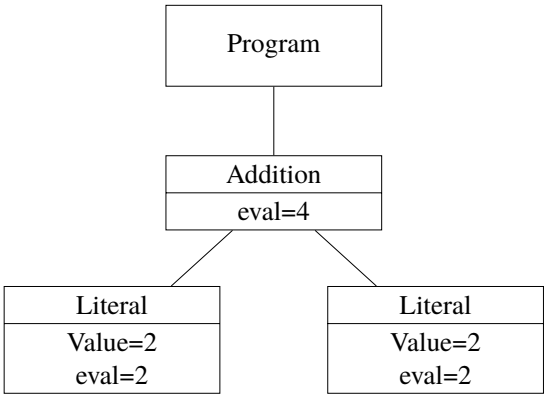

Figure 2.6: The attributed abstract syntax tree after evaluation.

evaluated. Addition's "eval" is the sum of its children's "eval" and the Literal's "eval" is simply its value. The evaluated tree is shown in Figure 2.6. This is about as simple as an attributed AST can get. The attributes are often used for more complex things such as type checking, name lookup and error handling. Although these concepts are mainly part of the semantic analysis, parts of the synthesis, such as code generation, can be written with attributes as well.

What follows is the code written in JastAdd's declarative language that specifies the AST nodes as shown in Figure 2.6. The first code snippet is that from the ".ast" file, it specifies that the Program node can have an Expression node as a child. An Expression can be either an Addition or a Literal. The Addition node has two children, A and B, which both are Expressions. Lastly the Literal node has a token called "Value" of the type int[3]. Note that Expression is abstract.

```
Program ::= Expression;
abstract Expression;
Addition : Expression ::= A:Expression B:Expression
Literal : Expression ::= <Value:int>
```

The second code snippet adds the specification of the evaluation attribute "eval" for Expression and its subtypes. The "eval" attribute is specified as synthesized attribute at Expression with the equations for the attribute specified in Addition and Literal seperately. In Addition it is specified as the added values of its child nodes' "eval" attributes.

```
syn int Expression.eval();
eq Addition.eval() = getA().eval() + getB().eval();
eq Literal.eval() = getValue();
```

2.3.1 Classes

We previously mentioned that JastAdd compilers consist of Java-classes. We will now give brief descriptions of the classes that are escpecially relevant in this report.

[3]The primitive type called int in Java corresponding to an integer.

The `ASTNode` class is the base class for all nodes in the AST. So, in our previous example from Figure 2.6, all the node classes inherit from the ASTNode class.

`List` and `Opt` are two other important classes, they are also ASTNodes but fulfill a special purpose. A List class contains zero or more ASTNodes, while an Opt class contains zero or one ASTNode. Furthermore, tokens, as shown in the code snippet when defining the node Literal, are meant for holding literal values such as strings or integers and are not implemented as ASTNodes.

2.4 Java and JavaScript

Since JastAdd generates Java code and we want to build a web application using JavaScript we have to transpile the code from Java to JavaScript. When using such transpiled code, there are some key differences to note. Java is a static type language, meaning that a variable has a type which cannot be changed, whereas JavaScript, which is a dynamic type language, variables have a type that can be changed any time. This is important to note because the compiler in Java would protest if you tried writing over the value of an integer with e.g. a string but JavaScript will not. And since what we want to do is to use a program in JavaScript, but in the manner it was intended to function in Java, we have to enforce all the rigor of static typing by ourselves.

Another important difference is that there are no private or protected variables in JavaScript. Any variable or object belonging to a class in JavaScript can be readily retrieved and manipulated. You therefore have to be very careful when handling class-variables.

In general, where you can make sure that programmers can not violate how the code was intended to work in Java, you cannot do so, at least not as easily, in JavaScript.

2.5 Domain Specific Languages

A *domain specific language* (DSL), is a programming language created to work very well in a narrow context[8]. HTML might be a familiar example. It is used for specifying the layout of webpages, but is not used for much else. For example, just as you would not write a calculator program in pure HTML, you also would not write a webpage using only Java. Java, in contrast to HTML, is a *general purpose language* (GPL), a language in which you (in theory) can write any program. What we tried to demonstrate with the example is the tradeof between DSLs and GPLs. An analogy would be that a GPL is like a swiss army knife. It has a lot of tools for doing many different things. But if you have the option of using either a potato peeler or a swiss army knife to peel a potato, the choice is obvious. The point being, some problems are really difficult to solve with a general purpose language, so we instead use a domain specific language.

The reason this is important, is that for many DSLs, structural editing is advantageous compared to textual editing. Also, one could argue that there is more incentive to create new DSLs than GPLs. Creating a programming language for solving problems in a specific context seems intuitively more useful than creating a new GPL that tries to do the same thing as other GPLs.

2.6 Related Work

We will now briefly introduce two structured editors, related to field of this master's thesis.

2.6.1 JATTE

JATTE is self-described as a tunable tree editor for integrated DSLs[7]. It is, just as our web-based tree editor, based on JastAdd compilers but it is implemented in Java in contrast to our editor which is implemented in JavaScript.

Through attributes in a language's specification JATTE allows users to customize how the ASTs should be displayed for different languages. Allowing users to specify which nodes should be hidden and when, how nodes should be labeled, default values and more.

2.6.2 Jetbrains MPS

Jetbrains Meta Programming System (MPS) is a system for creating and using DSLs.[8] It is meant to allow creation and use of languages in an integrated development environment, introducing a concept called *language oriented programming*. LOP is a programming paradigm based on the perspective that programming languages are tools for communicating ideas between humans and computers, and that the ideas themselves should be in focus.

The MPS has been a good source of inspiration during this thesis, showing the frontier of the field of structural/projectional editing.

Chapter 3
System Overview

Building a tree editor that works through a web browser and uses JastAdd compilers introduces one key problem. JastAdd generates compilers in Java code while the tree editor (running in the web browser) is written in JavaScript. Somehow we have to transfer information between the compilers and the editor, so that the semantic analysis can be performed on the constructed abstract syntax tree. This can be done, either by creating a server that transmits the necessary information between the compiler (in Java) and the tree editor (in JavaScript) or, by transpiling the compiler into JavaScript and building the tree editor directly on top of the compiler which is what we chose to do. This means that the server sends a compiler to the client and that the compilation of ASTs (semantic analysis and synthesis) is done in the web browser. In Figure 3.1 you can see a simple schematic overview of the system.

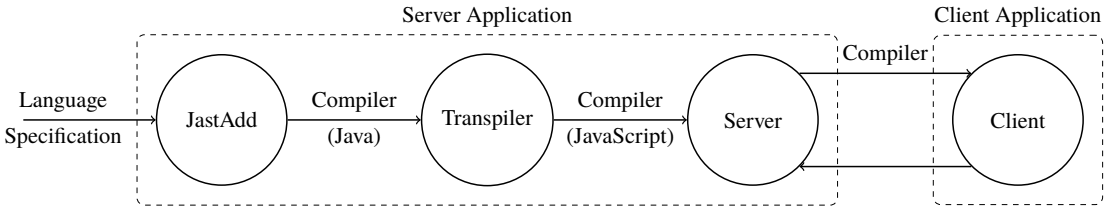

Figure 3.1: A simplified overview of how the system is structured.

What follows in this chapter is an overview of the system built during the course of this master's thesis.

3.1 Client-Server

When building a web application, the client-server paradigm is common[3]. This means splitting the application into two seperate parts, a server and a client application, where

the server runs continuously while waiting for clients to connect and retrieve or transmit information. For our purposes, the client-server model works quite well.

The server provides the client with a transpiled compiler and the associated abstract syntax. The server has a couple of different compilers stored and is also responsible for loading and saving ASTs built with the tree editor. The client application on the other hand consists of a webpage where one can choose a language (actually choosing the associated compiler) as well as edit, load and save ASTs with the tree editor.

3.2 The Tree Editor

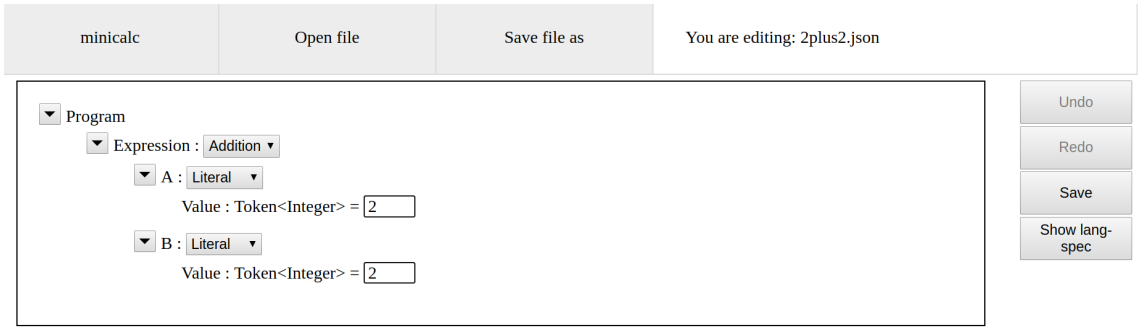

Figure 3.2: The tree editor displaying a simple AST.

The most important part of the system is the tree editor. In Figure 3.2 a simple AST is shown for the minimal language minicalc. The AST is the same as in Figure 2.2, describing the mathematical expression 2 + 2. We will thoroughly describe how it works in the following chapter regarding implementation.

The tree editor is built using the architectural pattern model-view-controller (MVC)[6]. This means that the editor is split into three separate parts namely model, view and controller.

The model is the core in this architecture. In our case, the model is an abstract syntax tree where the nodes of the tree are provided by our transpiled compiler. The view displays the state of the model, which for the tree editor means mapping the nodes of the AST to a graphical elements. In this way the structure of the AST is displayed as well as any relevant information contained in each node. Furthermore, the view also has to indicate what the user can do, which in this case means showing what actions can be performed on which nodes. For example removing and adding nodes, changing values and so on. Lastly, the controller, which is responsible for modifying the model, listens for user actions and then manipulates the AST accordingly. This architecture, gives us the ability to modify model, view and controller separately. Allowing us, for example, to create different views for different programming languages, without it affecting the model or the controller. It also makes debugging easier since the errors can be located more easily in this architecture than if the code was written in a less modular fashion. As an example, since we can examine the view and the model separately, we can see if the view is correctly displaying the contents of the model.

Chapter 4
Implementation

Here we will present the different parts of the system created during the course of this master's thesis. We will describe how the system was built and the relevant details of how it works. The first section contains the server application implementation and lays a foundation for understanding the second section regarding the client application.

4.1 Server Application

The server is built with a lightweight Python framework called Flask[1]. It allows us to register different endpoints (URLs) for different HTTP-requests, e.g. an endpoint for a GET request that retrieves an HTML-document (a webpage).

As was briefly explained in the system overview, the server application is tasked with supplying the client with compilers along with their associated abstract syntax, as well as enabling ASTs to be saved and loaded. What follows, is a description of the implementation details of each part of the server application.

Figure 4.1 describes the process of generating a compiler and transpiling it into JavaScript as well as extracting the needed information about the AST structure (the abstract syntax) from the language specification.

4.1.1 Transpilation

For our Java to JavaScript transpilation we chose to use a transpiler called JSweet[2]. It is built to enable Java programmers to write web applications in JavaScript. The programs are written in Java with certain JavaScript libraries and then translated into JavaScript. Despite JSweet not fitting our purpose perfectly, we could not find a more suitable tool,

[1] http://flask.pocoo.org/
[2] http://www.jsweet.org

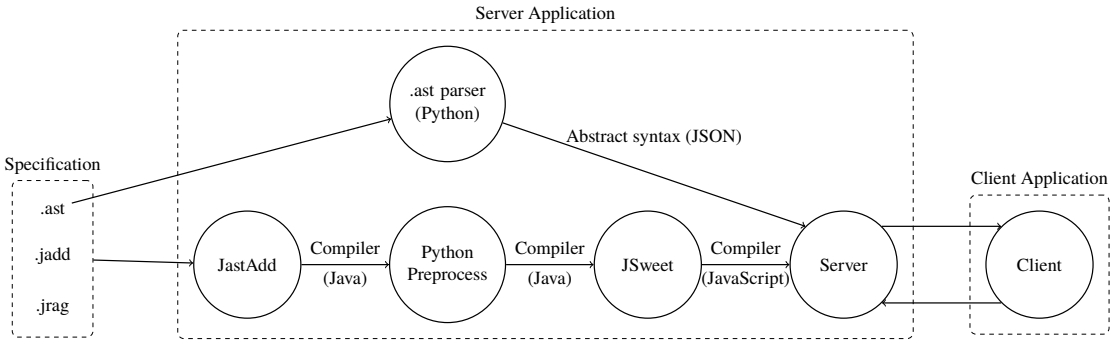

Figure 4.1: The process of generating the necessary information for the tree editor.

all Java-to-JavaScript transpilers we found seemed tailored for the same purpose (building web application with Java). However, there is an extension for JSweet called J4TS[3] which contains a close emulation of Java's standard libraries, allowing us to translate the compilers written in Java to JavaScript with only some modification.

There were however a few problems regarding JSweet, such as default methods in Java interfaces not being handled correctly. The transpiler did not copy the methods as required by Java, instead raising errors related to missing methods. To solve this problem a Python program was written that modified the compilers slightly before transpilation by simply copying the aforementioned default methods into the Java files where they were missing.

A problem that could not be solved in this fashion however, was that JastAdd annotates some methods with information related to the abstract syntax of the associated AST. This information is meant to be retrievable through *reflection*. Reflection can generally be described as the ability for a program to examine and modify itself during runtime.[2] However, since there is limited support for retrieving annotated information in JavaScript the information passed through annotations by JastAdd could not be retrieved in JavaScript, despite it being present in the transpiled target code.

Since the information about the abstract syntax is absolutely vital for any tree editor, we had to find another way of transferring this information. The solution implemented was a program written in Python. It translated the information contained in .ast files (used to specify the abstract syntax in JastAdd) into JSON (JavaScript Object Notation). The JSON files were then used as a basis for construction of ASTs in the tree editor.

4.1.2 Encoding AST Information

To save and load ASTs we chose to encode the information in JSON format. This due to the client application being written in JavaScript and the server application being written in Python, which both have close connections to JSON. JavaScript for obvious reasons, and Python because its syntax for lists and dictionaries is almost identical to the syntax for lists and objects in JavaScript, making JSON an almost seamless bridge between the two languages.

However, there were arguments to be made for XML encoding too, specifically, compatibility with the related tree editor JATTE. However, due to time constraints we chose to

[3] https://github.com/j4ts/j4ts

only implement JSON encoding.

The actual encoding is done by creating an equivalent tree structure to that of the AST to be saved. This tree does not contain all the information that the AST does, only the information absolutely necessary for recreating the AST. For each node the information stored is what children it has and any token values. This is stored as key value pairs where the key is the child or token name and the value either the child's class or the token's value. Attributes are not stored.

```
{
    "Program": {
        "Expr : Expr": {
            "Addition": {
                "A : Expr": {
                    "Literal": {
                        "Value : Token<Integer>": 2
                    }
                },
                 "B : Expr": {
                     "Literal": {
                         "Value : Token<Integer>": 2
                     }
                 }
             }
         }
     }
}
```

As an example, the code above shows the JSON encoding of the AST shown in Figure 2.6.

4.1.3 Automating the generation

To allow easy generation of compilers from language specifications written in JastAdd, another Python script was created that automates the process described in Figure 4.1. I. e. running JastAdd, followed by preprocessing, JSweet and so on and lastly placing the resulting files in a structured manner and supplying the client application with JavaScript compilers and corresponding abstract syntax.

4.2 Client Application

The client application consists of some HTML documents supplied by our server as well as CSS for styling. All functionality is written in JavaScript, as is standard practice for web applications. This includes, choosing compiler, loading and saving files, as well as the tree editor. We chose not to use external JavaScript libraries such as jQuery, which is quite common practice, instead writing the client application in standard JavaScript. The only external code dependency is on the JavaScript library J4TS[4] previously mentioned.

[4] J4TS version 0.5.0 to be specific, https://github.com/j4ts/j4ts

In Figure 4.2 below, you can see a screenshot of the client application, as shown in a Chromium web browser. The tree editor is displaying the AST from Figure 2.2.

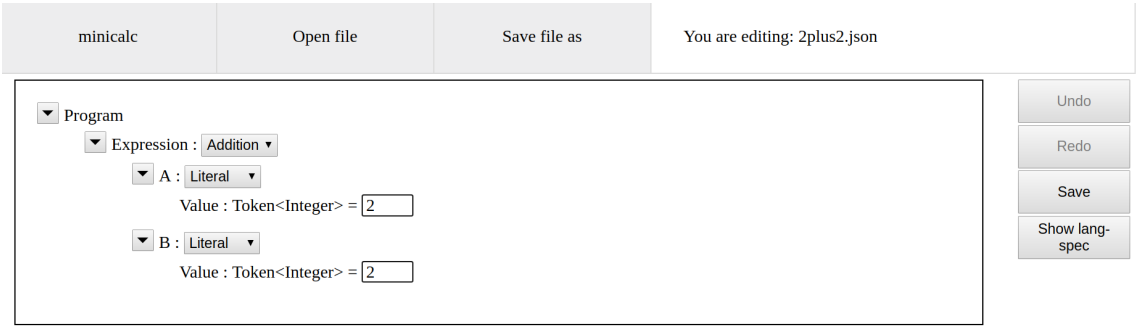

Figure 4.2: Showing the graphical user interface, the tree editor is showing the example from Figure 2.2.

The user can change the types of expressions, e.g. from Addition to Literal or vice versa, as well as collapse parts of tree and change the token values of the Literal-nodes. Another screenshot, shown in Figure 4.3 on the next page, displays a more complex AST. What is shown is a program in SimpliC consisting of one empty function declaration 'f'. Here you can add nodes of different types using the add buttons, building the AST.

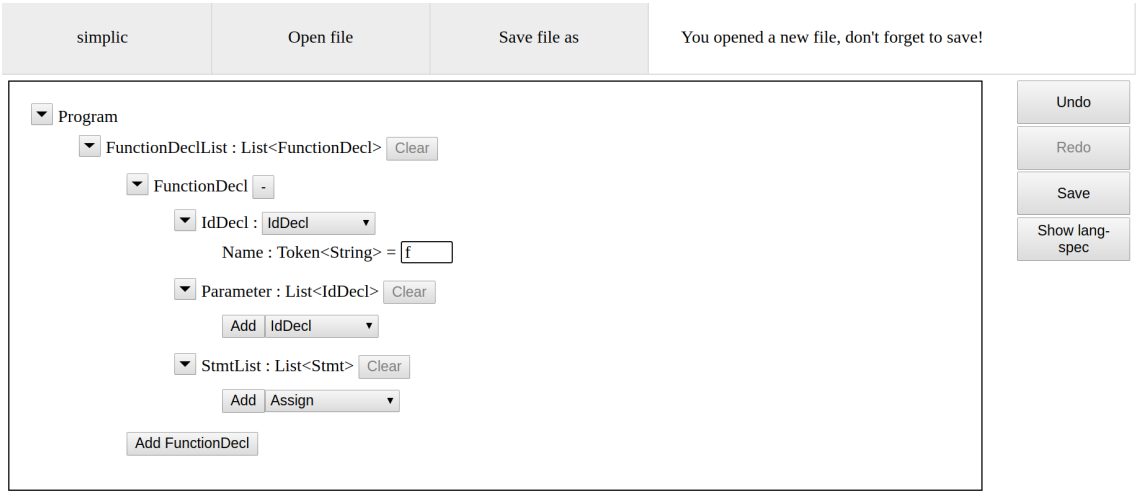

Figure 4.3: The tree editor is showing the example from a function declaration "f" in the SimpliC language.

What follows will be a thorough explanation of the inner workings and implementation of the client application.

4.2.1 Tree Editor

Building the tree editor was the most time consuming part of the whole implementation process. We started by including a transpiled compiler in the web page and exploring how

the classes in the compiler could be manipulated. After being convinced by our exploration that the compiler was at the very least able to build abstract syntax trees we started implementing the actual tree editor.

The first iteration resulted in a somewhat working tree editor that could add nodes and display an AST. After the first iteration, the tree editor was split into three separate parts, as mentioned in chapter 3, namely model, view and controller.

Tree Model

The model is built as an interface to access the imported compiler. It contains methods for different manipulations of ASTs such as creating and removing nodes as well as setting child nodes and tokens. Furthermore, it provides different getters and some methods for retrieving information about the AST nodes such as inheritance and child nodes. The model is also tasked with creating an AST from the JSON representation as well as encoding its internal state, the AST, into JSON.

We want to make sure that the AST is always a complete tree that does not violate the abstract syntax of the language it corresponds to. To achieve this, we have to create a complete and correct subtree when the user makes changes to the tree. But, creating a complete subtree is not entirely trivial. We made the choice that we should premier smaller subtrees, minimizing both the amount of nodes in the subtree and its depth. We constructed an algorithm that weighs each node class based on the class's children and tokens. If the class only has tokens we define its weight as the number of tokens, which can be computed directly. If a class has child nodes the class's weight is defined as the number of tokens plus the minimal possible weight for each child node plus the number of child nodes. When a weight is added, the choice of children corresponding to that weight is also stored. This weighing process is iterated until all nodes have weights and a list of children that make up this minimal weight. A class that contains child nodes that do not yet have weights will be skipped and re-evaluated next iteration.

If we process the available classes by taking the AST leaf nodes first (the classes with only tokens), followed by the rest of the classes sorted by number of child nodes in ascending order, we get fewer iterations in the weighing process. This is mainly a heuristic to minimize the times a node class has to be skipped due to incomplete weight.

Tree View

The view is built with an object oriented approach, where the classes in the view correspond to nodes and tokens in the model. The base class is called ViewNode and corresponds to base class ASTNode. The sub classes are ViewList, ViewOpt, ViewRoot and ViewToken. The first two correspond to the List and Opt classes whereas the ViewRoot is its on class simply because the root node cannot be handled analogously with other nodes due to it being placed differently in the HTML document. Lastly, the tokens need to be handled as if they were nodes in the AST since we want to enable user input for token values, and this makes it easy to handle tokens differently from nodes.

We choose to limit the number of classes representing nodes and tokens to the mentioned five since we want to keep the view as generic as possible. Enabling it to work in tandem with any compiler generated by JastAdd. Another approach that might be feasible

would be to represent only the ASTNodes that are neither Opts nor Lists. Instead showing the Lists and Opts as a part of its parent node's view. This might correspond better to the general structure of JastAdd compilers.

Tree Controller

The controller might be interesting from an implementation point of view, despite it using a somewhat standard implementation of the Command pattern. For each possible modification of the AST, which we will call an action, there is a corresponding command class. This class consists of the encapsulated action, how to undo the action and how to redo it. Redo might seem redundant but we actually want to end up in the exact same state as if we had just performed the action. E.g., if we add a node with some action, then undo the action, the following redo should add the node we just removed, not a new node, which is what the action would have done. The reason we want to make sure we are not, as the example showed, adding new nodes when we redo an action is twofold. The first one being that we do not want to perform unnecessary computations; it is much easier to reattach a child node to its parent than to create a new node and then attach it. Secondly, if we see each action as a state transition, taking the AST from one state to another, then the undo should do the opposite state transition, going from the later state to the previous. The redo should then do the opposite of the undo transition, making sure that the state we end up in is identical to the next state, and not a new state. Ensure that these transitions change the states as expected is a way to ensure that the program represented by the AST is not altered in some subtle, unexpected way.

As mentioned, each action corresponds to a command class. When an action should be performed the corresponding command is created and added to what we call the command stack after which the encapsulated action is performed. The command is created by a view component but all model alterations are contained in the controller according to the MVC pattern. One thing that was not implemented completely according to the MVC pattern is that after a command has altered the model, the model should update the view. In our implementation, it is the controller that prompts the view to update, the model itself does not know if it has been updated or not.

Error Handling

After the tree editor had been implemented and ASTs could be manipulated, we wanted to see if we could display semantic errors in some meaningful way. And this seems like a good time to mention it, for most of the testing we used a language called SimpliC. It is basically a simplified version of the language C and is created as an laboratory exercise in a compiler course at LTH.[5] The error aspect specified for this language links the semantic errors to specific line numbers that are part of the information supplied by the parser. But line numbers in our context of a tree editor do not correspond to anything. Thus the error aspect had to be modified to instead refer to nodes. To do this we generated an identification number for each node and refered to this number instead of a line number, otherwise the aspect is unchanged.

[5] It is called EDAN65 Compilers, for anyone who might be interested.

The resulting language and corresponding compiler is something we can use with our tree editor. To provide the desired feedback about erroneous nodes, we mapped each node ID to its corresponding view component. After an update of the view or directly after the editor has finished loading, it calls the model to see if there are any errors and outlines the nodes containing errors accordingly.

Since there is nothing that ensures that an attribute is valid after the AST has been altered, and the only attributes that we examine at this point are those related to errors, all attributes are recomputed on change.

Figure 4.4 shows how errors are displayed in the tree editor. In this case, it highlights the invalid node "WholeNumber" which has a value of 2.1, informing the user that 2.1 is not a valid value for whole number.

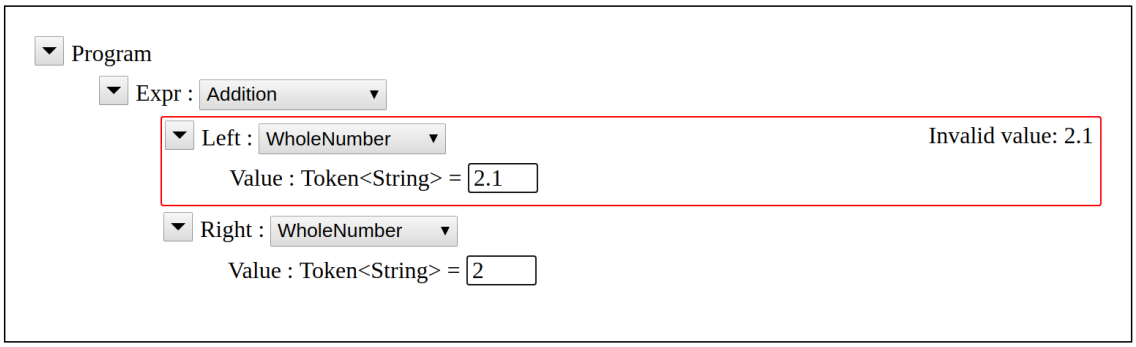

Figure 4.4: Example of how the tree editor displays errors.

Figure 4.5 shows another error, now in the SimpliC language. Here we can see that the variable x is being used before it has been declared. The user is trying to assign it the value 10.

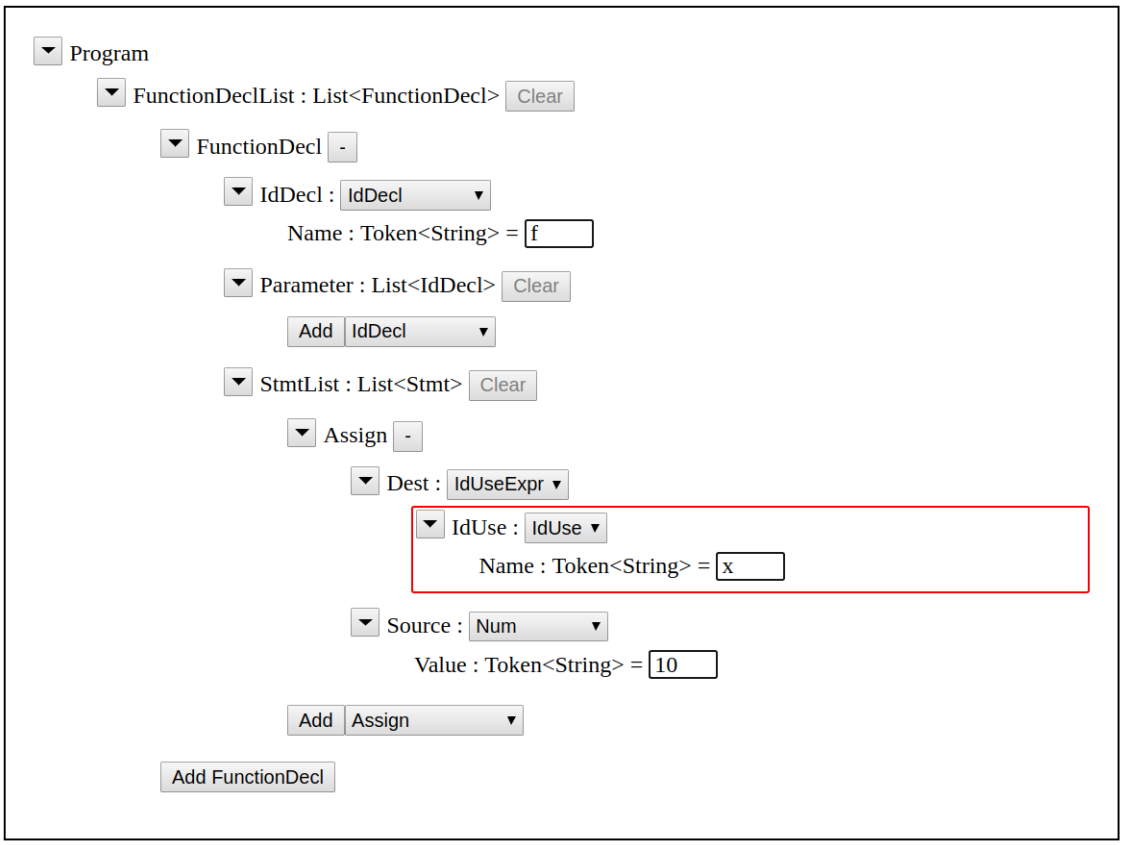

Figure 4.5: Example of a semantic error in a SimpliC AST displayed in the tree editor.

Chapter 5
Evaluation

To evaluate the editor we wanted to examine its runtime performance, getting a sense of how execution time and tree size relate to each other. To measure this link, we chose to generate randomized abstract syntax trees. What follows is a section about how we chose to implement random AST generation followed by the performance evaluation of the tree editor.

5.1 Random AST Generation

We chose to implement the random tree generator in Java. The original reason for this was to allow for comparing performance between the Java compiler and transpiled JavaScript compiler. A later decision took us in a different direction regarding evaluation but the generator was at that point already built. Furthermore, the generator is built specifically for the language SimpliC and is thus only tested, at this point, with that language. However, it can easily be modified to support any language specified with JastAdd since the implementation is generic and does not assume any specific properties of the language, except for the generation of random token values.

The purpose of the random tree generator is to generate random, somewhat erroneous (from a semantic perspective) ASTs, so that we can use these trees for evaluation.

5.1.1 Retrieving AST Information

To be able to construct a valid abstract syntax tree for a given programming language one has to know its abstract syntax, which in our case is given as Java annotations by JastAdd. These annotations give information regarding the specific node's children. It describes each child of the node by supplying its name, type and kind. Name is simply the name of the child. Type is the class of the child and lastly kind describes whether the child is a node, list, opt or a token. (Note that the term child is used loosely, it usually refers to

a child node, a token is not technically a node. We can view child in this context as any meaningful class or construct in the context of the AST.)

But solely the abstract syntax is not enough to build a random tree. We also need to know the class inheritance of all our AST nodes. To do this we had to evaluate each class in the compiler's package and build a tree describing the class inheritance of all AST nodes. When building the tree we also checked to see which classes were abstract.

After computing both the class inheritance and examining the constructor annotations, there is enough information to build random abstract syntax trees.

5.1.2 Nodes

To understand how the random trees are built we will start by describing how a single node is built. First, we instantiate a node of a given class by that class's constructor without any arguments. The node instantiated will have any List or Opt child initialized but nothing else meaning that it may or may not violate the abstract syntax. If there are uninitialized children or tokens, they must be initialized so any child node is therefore instantiated and any token is given a value (more on token values shortly). As you might see, this leads to some recursive calls, since the instantiated child might have children of its own that have to be constructed.

To see which nodes can be chosen as children to a parent node we first have to examine what class this child has to extend. Then we can examine our class inheritance tree and see which classes inherit from that given class. After retrieving a list of classes we can chose one of them randomly and instantiate it, setting it as the child. But, this is where we run in to a problem. For example, say that we want to choose a class inheriting from an abstract super class Expression. Maybe there are 20 expressions where 10 of them are binary, 9 unary and 1 is a literal. If the probability of choosing any of these class is distributed evenly over all classes we see that the probability of a child being a leaf node, which is only the case if the child is a literal, is $\frac{1}{20}$ and the probability of choosing a binary expression is $\frac{1}{2}$. Which will most likely lead to an infinite recursion.

To offset this, we change the probability of choosing a leaf node. If there is are classes corresponding to leaf nodes in the set of valid child classes we choose such a child with probability $\frac{2}{3}$ and choose a child from the whole set of valid classes with probability $\frac{1}{3}$.

At this point, we can construct random abstract syntax trees that coincide with to the abstract syntax. However, the sizes of these random trees are also random and that is something we whish to control. To do this, we will use the List and Opt classes. The List class holds any number of nodes and the Opt class can hold zero or one node. For each node we create, we register its List and Opt nodes, if any. Then, if after a complete AST is built, it has less than the desired number of nodes, we chose one of the List or Opt nodes registered by random[1] and add a node to it. (Which in turn will generate a few children of its own.) This process is iterated until we have reached the desired number of nodes or more. Now we can generate arbitrarily large random abstract syntax trees.

[1]The random selection of List and Opt nodes is not as uniformly distributed over all these classes. It is instead distributed over all classes cotaining Lists and Opts. (Opts however can only be choosen once since it can only hold one node, resulting in approximately uniform distribution of over all nodes containing List nodes.)

5.1.3 Token Values

The last part of the random tree generator is the generation of random token values. We wanted the token values to at least mimic both general coding conventions and some sort of meaningful use of variables and functions. So we choose to randomly join metasyntactic variable names such as foo and bar as function names, sequences of random letters followed by numbers as variable names and random integers, doubles and floats for numbers.

To determine if a token should be a variable, function or a number it suffices to examine the class which has the token and its parent.

Now, to the actual generation of random values. For functions, we chose to construct a list of commonly used metasyntactical variable names, which we will refer to as function phonemes. To create a function name, a random phoneme is chosen. If a function with that name already exists, we do one of the following: append another phoneme with probability 0.5, append a random digit with probability 0.25 or accept a name conflict with probability 0.25. If the outcome is other than accepting the name conflict, we try again to see if the name is in our list of already declared function names and so on in a loop until either we accept a conflict or an unused name.

The variable names are choosen similarly although the probability of choosing the same name is higher. We choose to accept the same name with the probability 0.5 and append a random character or digit with the probability 0.5. The somewhat arbitrary decision of having more name collisions of variable names is based on the scoping of SimpliC where all functions share the same scope, whereas variables do not. A variables scope in this language is limited to the function where it is declared. Furthermore, we want erroneous name collisions, we want the AST we generate to have semantic errors so that we can examine these errors in the tree editor.

5.2 Performance Evaluation

We approached the evaluation thinking: what are the important performance aspects of our tree editor from a user perspective and how can we measure them. We chose to select the loading time, meaining the time it takes for a saved AST to be loaded into the tree editor, followed by the time it takes to create nodes in the tree as well as the time it takes to compute the semantic errors in the AST.

5.2.1 Methodology

The tests made to measure the editors performance are based on A. Georges et al. *Statiscally Rigorous Java Performance Evaluation*[5] and the assumption that the JavaScript compiler we used, the V8 engine, exhibits a startup and steady-state behaviour, which should be the case for any compiler with runtime optimizations. We will later show that this is the case but we will first describe the concept of startup and steady-state.

In Java, the startup includes class loading as well as just-in-time (JIT) compilation, which is when code is being compiled and optimized during runtime. This results in quite rapid decrease in execution time, due to the code being optimized with each iteration. This

is similar to our JavaScript case at least in the sense that both the Java Virtual Machine (JVM) and the V8 engine use runtime optimizations.

To measure the startup performance one should execute the code with only one iteration at a time, so as to only measure the worst case, since every iteration, to a point, should result in more optimized code.

Since most JIT compilation is performed during startup, and thus also optimized during startup, the steady-state varies less based on these optimizations. The code can only be optimized to a point, after which the execution time for each iteration will be more consistent. Our approach to finding the steady-state was more lax than the rigorous methodology described by A. Georges et al. We examined the data from our steady-state tests (Figure 5.1) and approximated when the steady-state was reached.

5.2.2 Chromium V8

The V8 engine is written in the language C++ and does not interpret JavaScript, it directly compiles the source code into native machine code (JIT compilation). Finding out exactly which optimizations the JavaScript compiler performs seemed like tedious task so we instead focused on examining the behaviour of the V8 engine. One of the key behaviours we were looking for was if the compiler exhibited a distinct startup and steady-state phase. As mentioned previously, this means that if you execute some process n times, the first few times will vary greatly due to optimizations and after some amount of iterations, the code will be optimized and the execution time will be stable.

We can assume that the V8 engine exhibits this startup - steady-state behaviour since we know it performances runtime optimizations but we would like to actually prove it and also see how many iterations it takes to get to the steady-state. So we made two series of tests. Each test serie consists of 50 tests with 200 iterations each. We made the arbitrary choice of using a random tree with 800 nodes for all of these tests.

The first test serie, see Figure 5.1, measures the time it takes for the editor to build an AST from the encoded format described in section 4.1.2. This means going through a tree structure and instantiating classes for each node in the tree. Each iteration consists of rebuilding the same AST from the same encoded data. The time it takes to load the encoded data is not included in the measured time.

The second test, see Figure 5.1, measures the time it takes to remove all attribute values (flush) in the AST and then compute the semantic errors. To be specific, we call a method that clears all attributes, starting from the root, and then an error attribute which goes through the AST in search of semantic errors. Computing the semantic errors should be much more computationally expensive than removing the attribute values. Each iteration consists of first performing the flush operation and then the error operation.

Both of these tests were done without any graphical components, using only the tree editors model, as we wanted to reduce any noise generated by graphical updates of the HTML document.

As is shown in Figure 5.1, the computation time seems to strongly decrease with the first couple of iterations to ultimately stabilize around some value. This is exactly what we were looking for and we can conclude that the V8 engine exhibits the startup and steady-state behaviour, as is expected of any compiler using runtime optimizations.

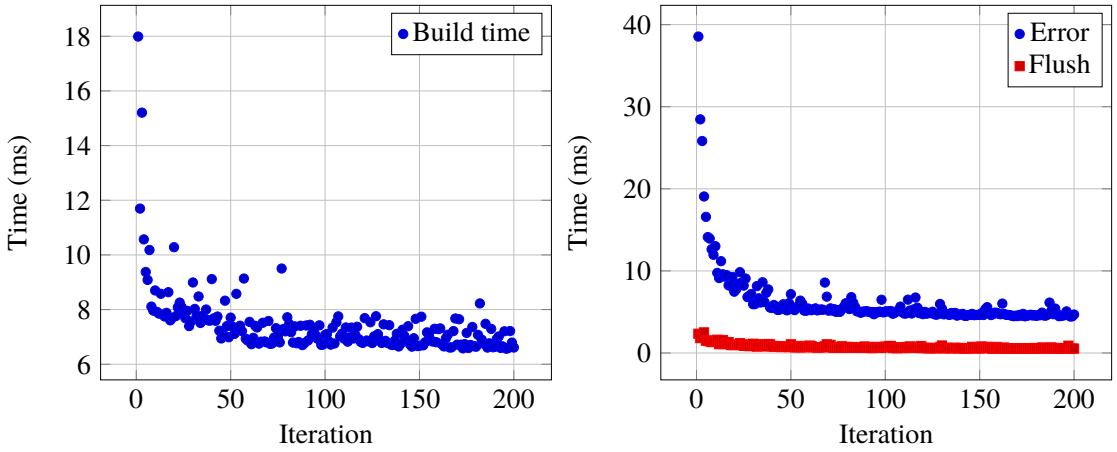

Figure 5.1: The graphs represents the mean computation time for error, flush and build. Each dot represents the mean of 50 tests.

Testing and Asynchrony

JavaScript is an asynchronous language, meaning that the execution order of a program is not necessarily the same as the order of the instructions in the source code. This is quite useful when dealing with HTTP-requests and user inputs so that the execution of a program does not halt when waiting for such responses. But, this introduces difficulties for evaluating performance.

Something we did not previously know about V8 is how it updates the HTML *document object model* (DOM). If you make changes to the DOM through JavaScript code, this starts an asynchronous processes that update the visual components or just the underlying document structure. Measuring the time it takes to make these updates becomes quite difficult due to the fact that there is no clear way to see if the changes have been completed.

It becomes even more difficult due to the varying implementations of JavaScript compilers between web browsers, but something that seems to work in our context of the Chromium browser is using the method `setTimeout` which takes a callback function reference and a optionally a wait time as arguments. It seems that calling this timeout function with some callback function but without the wait time argument (or 0) schedules the callback as soon as possible but after the current code has executed. Effectively calling the callback function only after in this case the DOM operations are done. If this is actually the case is quite hard to verify but it seems to at least give a decent estimate of the perceived time of the graphical updates.

We used this `setTimeout` callback method to measure the time it takes for changes that include DOM modifications.

Standard and Incognito Mode

Chromium has two modes, standard and incognito. The second being a mode where no data is cached for privacy purposes, but it is also very useful when building web applications due to it not caching e.g. JavaScript code. It can be quite frustrating to debug code when the code in the browser does not update when the source code is changed. We ran a few tests to determine if we could find significant performance difference between the

two modes. But since we could not, we chose to measure performance mainly in incognito mode, simply due to preference.

5.2.3 Results

The trees used for these tests are generated by the aforementioned random AST generator. We chose to use only fairly large trees, 800 to 6400 nodes, since evaluating performance for smaller trees seemed to generate a lot of noise. Also note that in both graphs shown in this section, the confidence intervals are very small and cannot be made visible in the graphs.

The first results, see Figure 5.2, are measures of the computational time of performing the flush and error operations. That is, removing all attribute values and computing all semantic errors in the AST respectively. The computation time is measured in steady-state, which we approximated as the last 30 measurements in every set of 100 iterations. We ran each set of iterations 50 times per tree for four different tree sizes: 800, 1600, 3200 and 6400. Each point in the graph is calculated as:

$$\bar{x_k} = \frac{\sum_{i=1}^{50} \frac{\sum_{j=71}^{100} x_{ij}}{30}}{50}$$

where $k = 800, 1600, 3200, 6400$, i is test run and j the iteration. Meaing that each point is the mean of the steady-state means.

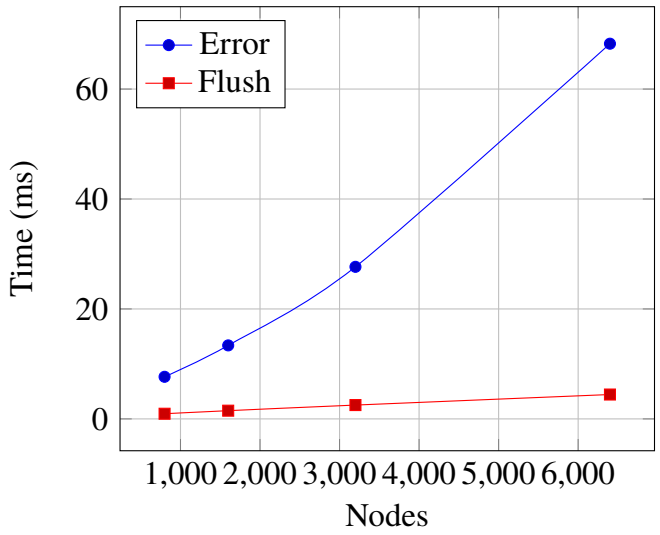

Figure 5.2: Error and flush time, without graphical componenets.

As we can see there seems to be a linear relationship between the number of nodes in the tree and the computation time to perform the flush and error operations. We can also see that the error operation takes signifacantly more time to perform than flush, about 10 times as much. This test only measures tree model time meaning that the view and controller part of the tree editor are disabled. No graphical updates are performed during the test as to avoid any noise caused by asynchronous function calls, as discussed in section 5.2.2.

It is worth noting that the timescale given in Figure 5.2 is in milliseconds; performing semantic analysis on a tree with 6400 nodes takes about 68 ms, which is quite fast.

The next test measures the time it takes to build an AST, including the graphical components. The tests are perfromed by restarting the Chromium browser for each test. This should mean that we get the worst case in terms of computation time since no runtime optimizations have yet been done. Furthermore, since incognito mode is used, nothing will be cached between the tests, which could be the case otherwise, leading to an improved runtime.

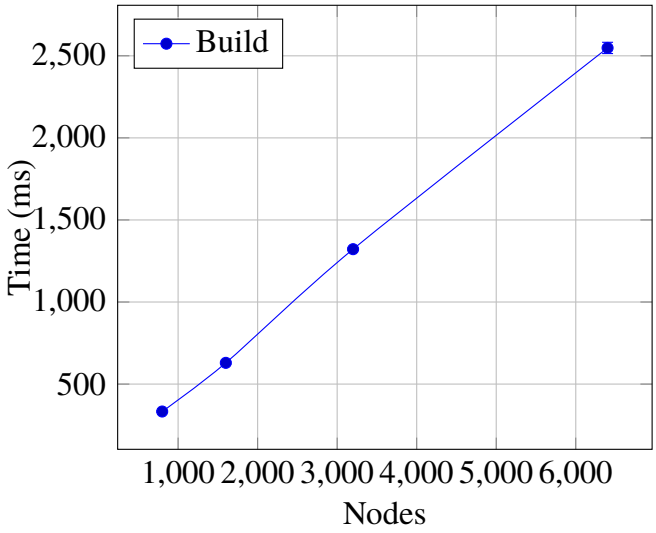

Figure 5.3: The tree editor build time, including graphical components. Each point is the mean of 5 samples.

The graph in 5.3 shows the time it takes for the tree editor to build ASTs with 800, 1600, 3200 and 6400 nodes respectively. This should correspond with the worst case for loading the tree editor. As can be seen, this graph also shows a somewhat linear relation between the number of nodes and time it takes to build it, which is reasonable, since the number of graphical elements is proportionate to the number of nodes in the AST. Furhtermore, this shows that opening the largest tree in our case takes about 2.5 seconds in the worst case scenario. This is most likely acceptable performance. One could also argue that the trees used in this tool would probably not be as large as 6400 nodes, maybe not even 800 nodes, meaning that the performance of the tree editor in general would probably not deter users from using it.

Chapter 6
Future Work

There were many things that could not be included in this thesis due to time constraints. Things that would make the tree editor we developed more useful. We will therefore discuss some known limitations of the tree editor, as well as some areas of future improvement that unfortunately could not be implemented during the course of this master's thesis.

6.1 Allowing Language Specification

One of the purposes behind building this JastAdd tree editor was to encourage users to make their own programming languages and allowing users a glimpse behind the scenes of programming; into the field of compilers. We believe it would be a good tool for learning about abstract syntax trees, semantic analysis and attribute grammars for anyone interested.

At this point, the JastAdd specifications have to be added manually to the server. To allow users to add their own languages we just need to allow them to add specifications. The simplest solution would be to let users input specifications as text or files to the client, which are subsequently sent to the server.

Another approach would be to build some sort of structured editor for JastAdd's declarative language. This approach is more in line with the rest of the project but will probably be a lot more time consuming.

6.2 Customization

A feature that unfortunately had to be left unimplemented was the graphical customization. Allowing users to specify how the ASTs were displayed for a given language. For example, say that you wanted to create a programming language for mathematical expressions. In such a language, viewing ASTs could be comparatively foreign in relation to the actual mathematical expressions. In Figure 6.1 you can see how such an AST would be displayed

in the current tree editor.

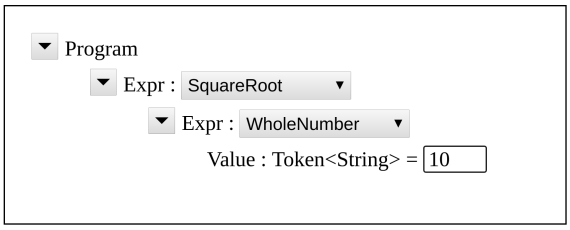

Figure 6.1: Example AST of the mathematical expression square-root of 10

As you can see, there is a lot of information displayed that might not be relevant to the user. It would be clearer to simply display:

$$\sqrt{10}$$

Something, that most users probably would prefer. There are many other cases where viewing the complete AST is just to verbose. Therefore, allowing customization of the graphical components is something that would be a valuable improvement to the current tree editor.

Exactly how graphical customization should be implemented is up for debate. JATTE's customization is based on specific attributes in the JastAdd specification. It allows a user to specify for example which nodes should be hidden and when, what labels the nodes should have, default values, as well as drag and drop behaviour.

JATTE's approach would probably work quite well in the context of our web-based tree editor as well. However, it is hard to tell if this approach is flexible enough to allow customization that changes the display from a generic tree editor to that of a graphical editor. For example, if you have a language for describing finite state automata, could you describe a customization based on attributes, that transforms an AST into something resembling a state diagram?

6.3 Full Semantic Analysis

The tree editor supports error checking which is only one part of the semantic analysis. The reason we only support error checking is that the tree editor only retrieves the abstract syntax, it has no knowledge of any added attributes and merely checks if the root node of the AST has a function called error.

To run the full semantic analysis we have to somehow retrieve all the added attributes. One possible route would be to implement a Java program which through reflection can extract all the information annotated in the generated JastAdd compiler. This information gives not only a full description of all the attributes added through aspects but also the structure of the abstract syntax tree. It has also been suggested that the annotated information could be gathered by writing an extension to the ExtendJ Java compiler (built with JastAdd). The ExtendJ approach is probably the most robust and rigourus. Furhtermore, actually compiling the generated compiler before transpilation would probably facilitate

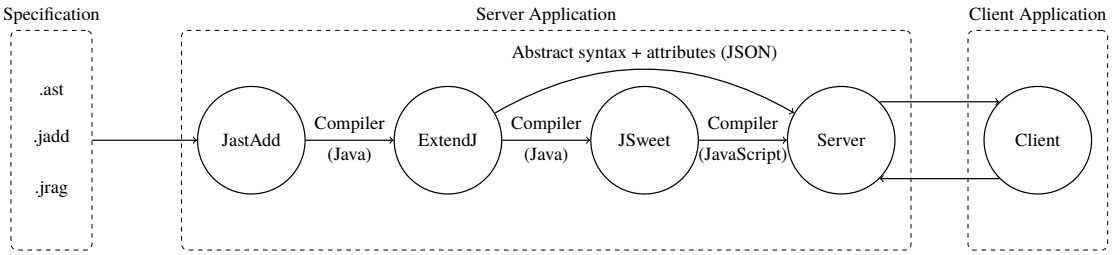

Figure 6.2: Example overview of how the compiler generation would be if ExtendJ was used for retrieving attributes and AST structure.

debugging of language specifications and could thus be useful in the context of what was discussed in section 6.1.

Using ExtendJ to extract information instead of Python, which is what is used at this point in time, would change the process from specification to compiler (in JavaScript), shown in Figure 4.1, into something like what is shown in Figure 6.2.

6.4 Synthesis

We do not support synthesis either, that is, the generation of target code. So you cannot at this point test if a compilers synthesis is working as it should. In other words, you can only examine the program that you built with the tree editor in either the form of an AST, attributed or not attributed, but you cannot transform an AST into target code, which might be desirable when building a compiler.

This problem however is closely linked to what was discussed in the previous section, if the full semantic analysis can be performed, and all the attributes are known by the tree editor, the synthesis can be performed. (As the synthesis is written as JastAdd aspects and thus correspond to the aforementioned attributes.) When the synthesis can be performed it is simply the somewhat trivial problem of returning the results in some format to the user, e.g. as a downloadable file.

6.5 Interpreter

Another area, closely related to synthesis, is interpretation. It is when a program is run directly, interpreted, without being compiled into target code.

Allowing a program constructed in the tree editor to be interpreted, run directly in the webbrowser, would be a useful feature. It would allow the user to evaluate if programs behave as expected, something quite essential when building a programming language.

One possible implementation of this would be to have some sort of console that gives feedback to the user. The easiest being textual feedback but graphical or structured feedback would be preferable since it is more in line with the rest of the project.

Chapter 7
Conclusion

This thesis has described our solution for running JastAdd on the web by building a web-based tree editor.

Through attribute grammar the JastAdd system can construct tools for analyzing languages, e.g. compilers. The compilers are generated as packages of Java classes, describing the nodes in the abstract syntax tree and their corresponding attributes. These compilers can be transpiled into JavaScript, maintaining the same structure and the same key properties as the original compilers.

We built a tree editor using a server-client paradigm, where the actual tree editor is a part of the client which is run in a web browser. The server is responsible for the process from the JastAdd specification all the way to the transpiled compiler (in JavaScript). The tree editor is split up in a model-view-controller pattern where the model is an interface to the transpiled compiler.

With the tree editor a user can build any abstract syntax tree that follows its language's abstract syntax. The compilers semantic analysis is limited to checking for semantic errors and it is run directly in the web browser. The results of the error analysis is provided as visual feedback to the user. The editor is mainly tested in the web browser Chromium.

During the testing we examined the computation time of different operations such as building an abstract syntax tree from stored data, performing semantic error analysis and removing all the attributes in the (attributed) AST. We could see that the time seemed to be linearly dependent on the number of nodes in the current AST. We could also show that the tree editor will be affected by runtime optimizations (in the Chromium browser), resulting in different computation times characterized as startup-state and steady-state. Furthermore, we stated that the performance time of the tree editor seemed adequate.

There is yet much left to be done if the tree editor is to work as a useful tool for developing new languages and compilers. It can still work as a tool for demonstrating some of the properties of JastAdd compilers, but it is far from anything like an integrated development environment. Despite this, the tree editor serves as a proof of concept; showing how web-based tools for JastAdd can be built.

Bibliography

[1] Andrew W. Appel and Jens Palsberg
Modern Compiler Implementation in Java
Cambridge University Press, 2009

[2] J. Malenfant, M. Jacques and F.-N. Demers
A Tutorial on Behavioral Reflection and its Implementation
Proceedings of the Reflection, 1996 - academia.edu

[3] Behrouz A. Forouzan
Data Communications and Networking Fifth Edition ch. 25
McGraw-Hill, 2013

[4] Görel Hedin
An Introductory Tutorial on JastAdd Attribute Grammars
Springer-Verlag Berlin Heidelberg, 2011

[5] A. Georges, D. Buytaert and L. Eeckhout
Statistically Rigorous Java Perfomance Evaluation
OOPSLA '07, ACM, 2007

[6] Glenn E. Krasner and Stephen T. Pope
A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System
ParcPlace Systems, 1988

[7] Alfred Åkesson, Görel Hedin
JATTE: A Tunable Tree Editor for Integrated DSLs
CoCoS '17, ACM, 2017

[8] Sergey Dmitriev
Language Oriented Programming: The Next Programming Paradigm
JetBrains onBoard, 2004

[9] Tim Teitelbaum and Thomas W. Reps
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment
Communications of the ACM volume 24 issue 9 (1981), (p. 563–573), ACM, 1981

EXAMENSARBETE Web-based Tree Editor for JastAdd Compilers
STUDENT Marcus de Lacerda
HANDLEDARE Niklas Fors (LTH)
EXAMINATOR Görel Hedin (LTH)

Webbaserad trädeditor

POPULÄRVETENSKAPLIG SAMMANFATTNING **Marcus de Lacerda**

Det finns ett växande behov för nya programmeringsspråk. Vi har byggt en webbaserad editor som faciliterar skapandet av nya språk och användandet av dessa vid ett tidigt utvecklingsstadium.

I en värld som blir alltmer digitaliserad finns det behov av att kunna skriva mängder av olika program med vilt skilda syften och applikationsområden. Dessa program skrivs i diverse olika programmeringsspråk, men för att en dator ska kunna köra ett program så behövs det något som översätter programmet till en form som en dator kan förstå: en kompilator.

En kompilator kan ses som tolken mellan människa och maskin. Den översätter från programmingsspråk, något vi människor kan läsa och förstå, till instruktioner som datorn förstår. Kompilatorer är alltså ytterst nödvändiga och något som personer i gemen kanske inte vet så mycket om.

Det vi har gjort är att skapa ett verktyg som fungerar direkt i webbläsare, en trädeditor, för att analysera samt utveckla kompilatorer och därmed i förlängingen även programmingsspråk. Med en trädeditor så kan användaren skapa program på ett sätt som är nära kompilatorns eget sätt att representera program. Detta står i kontrast till en texteditor där programmet istället skrivs i kod, vilket i sin tur måste analyseras i flera steg av kompilatorn. En trädeditor kan därför vara till stor fördel vid utveckling av kompilatorer och programmeringsspråk: det innebär att språket inte behöver någon syntax (regler för hur språket skrivs). Det enda som behövs är semantiken, reglerna för hur program ska tolkas, vilket ofta är det intressanta vid utveckling av nya språk och kompilatorer.

Med hjälp av ett system för att utveckla kompilatorer vid namn JastAdd kan vi smidigt gå från en formell beskrivning av ett programmeringsspråk och dess egenskaper till en kompilator som sedan kan användas i vår webbaserade trädeditor. Detta är alltså ett verktyg som kan användas väldigt tidigt i språk- och kompilatorutvecklingen. Det ger dessutom en särskild insikt i maskineriet bakom självaste programmeringen, något som så ofta annars förblir omärkt.

Men varför vill man ens skapa nya kompilatorer? En anledning kan vara att man vill ha en ny kompilator till ett redan existerande språk, exempelvis när en ny version av ett språk ges ut. I andra fall vill man skapa helt nya språk och ofta specialiserade programmeringsspråk. Språk som är särskilt anpassade för att lösa problem inom specifika områden och därmed kan fylla behov som inte tillgodoses av mer generella språk. Det finns mängder av så kallade *domänspecifika språk*: HTML, CSS och LaTeXför att nämna några. Och då tekniken alltjämnt utvecklas kommer nya knepiga problemdomäner upptäckas som kräver nischade programmeringsspråk. Därmed följer ett behov av nya kompilatorer och verktyg för att utveckla de. Vår trädeditor tar ett steg för att fylla detta behov samt att sänka tröskeln för att komma in kompilatorprogrammering.