Making a template query language for EcmaScript

Author: Rolf Martin Glomsrud

Supervisor: Mikhail Barash

Informal advisor: Yulia Startsev

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2024

Abstract

Lorem ipsum dolor sit amet, his veri singulis necessitatibus ad. Nec insolens periculis ex. Te pro purto eros error, nec alia graeci placerat cu. Hinc volutpat similique no qui, ad labitur mentitum democritum sea. Sale inimicus te eum.

No eros nemore impedit his, per at salutandi eloquentiam, ea semper euismod meliore sea. Mutat scaevola cotidieque cu mel. Eum an convenire tractatos, ei duo nulla molestie, quis hendrerit et vix. In aliquam intellegam philosophia sea. At quo bonorum adipisci. Eros labitur deleniti ius in, sonet congue ius at, pro suas meis habeo no.

Acknowledgements

Est suavitate gubergren referrentur an, ex mea dolor eloquentiam, novum ludus suscipit in nec. Ea mea essent prompta constituam, has ut novum prodesset vulputate. Ad noster electram pri, nec sint accusamus dissentias at. Est ad laoreet fierent invidunt, ut per assueverit conclusionemque. An electram efficiendi mea.

Your name
Thursday 30th May, 2024

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Intro goes here

Chapter 2

Background

2.1 Technical Committee 39

Technical Committee 39 is the committee which maintains ECMA-262 [26], the language standard for ECMAScript, and other related standards. They develop this standard following the TC39 process [22] for standard extension.

Technical Committee 39 (abbreviated as TC39) is a group within ECMA international, whose main goal is to develop the language standard for ECMAScript (JavaScript) and other related standards. These related standards include: ECMA-402, the internalization API of ECMA-262, ECMA-404, the standard for JSON, ECMA-414, the ECMAScript specification suite standard. The members of the committee are representatives from various companies, academic institutions, and various other organizations from all across the world interested in developing the ECMAScript language. The members are usually people working wit JavaScript engines, tooling surrounding JavaScript, and other sections related to the JavaScript language.

2.1.1 ECMA-262 Proposals

This section will contain what is a proposal, and how proposals are developed in TC39 for the ECMA-262 language standard.

A proposal in this context is a suggested change to the ECMA-262 language standard. These additions to the standard have to solve some form of problem with the current version of ECMAScript. Such problems can come in many forms, and can apply to any part of the language. A problem can be, features that are not present in the language, inconsistent parts of the language, simplification of common patterns, etc etc. The proposal development process is defined in the document TC39 Process.

TC39 Process

The TC39 process [22], is a process document describing how the extension ECMA-262 is performed. A suggested change to the ECMA-262 standard is in the form of a *proposal*. This process documents describes the stages a proposal has to pass through to be accepted into the ECMA-262 standard.

Stage 0 consists if ideation. The purpose of this stage is to allow for exploration and ideation around what part of the current version of ECMAScript can be improved, and then define a problem space for the committee to focus.

Stage 1, is the point the committee has started taking the suggested addition and will consider it. The are several requirements to enter this stage: A champion has to be identified, a champion is a member TC39 who is responsible for the proposal. A rough outline of the problem, and a general shape of a solution. There has to have been discussion around key algorithms, abstractions and semantics of the proposal. Potential implementation challenges and cross-cutting concerns have to have been identified. All these described requirements have to be captured in a public repository. Once all these requirements are met, a proposal is accepted into stage 1. During this stage, the committee will work on designing a solution, and resolve any cross-cutting concerns discovered.

Stage 2, a preferred solution has been identified. Requirements for a proposal to enter this stage: All high level APIs and syntax have to be described in the proposal document. Illustrative examples of usage created. An initial specification text have to be created. In this stage, the solution identified have to be refined, minor details ironed out, and experimental implementations will be created.

Stage 2.7, the proposal is principally approved, and has to be tested and validated. To enter this stage, the major sections of the proposal have to be complete. The specification text is finished, and all reviewers of the specification have approved. Once a proposal

has entered this stage, testing and validation will be performed. This is done through the prototype implementations created in stage 2, and all features of the proposal is validated.

Stage 3, proposal is recommended for implementation. Once a proposal has been sufficiently tested and verified, it is moved to stage 3. During stage 3, the proposal is implemented in all major engines. During this stage, the proposal is tested for web compatibility issues, or integration issues in the major JavaScript engines.

Stage 4, the proposal is completed and included in the standard.

2.2   AST and Babel

Abstract Syntax Tree

An abstract syntax tree is a tree representation of source code. Every node of the tree represents a construct from the source code. ASTs remove syntactic details that are present in the source code, and while maintaining the structure of the program with its tree. Each node is set to represent elements of the programming language, some common ones are statements, expressions, declarations and other language concepts. Every node type represents a grammatical construct in the language the AST was built from.

ASTs are important for working with source code, they are used by almost any tool that has to represent source code in some way to perform operations with it [38]. This is because the structure is simpler to work with then raw text, especially when considering tools like compilers, interpreters, or code transformation tools.

ASTs are built by language parsers. A language parser takes the raw source code of a language, and parses the code into an AST while maintaining its structure but discarding irrelevant information. A simple example of how JavaScript is parsed into an AST can be seen in Figure 2.1.

Babel

Babel is a JavaScript toolchain, its main usage is converting ECMASCript 2015 and newer into older versions of JavaScript. The conversion to older versions is done to increase compatibility of JavaScript in older environments such as older browsers.

Babel has a suite of libraries used to work with JavaScript source code, each library relies on Babels AST definition [4]. The AST specification Babel uses tries to stay as true to the ECMAScript standard as possible [8], which has made it a recommended parser to use for proposal transpiler implementations [23]. A simple example of how source code parsed into an AST with Babel looks like can be seen in Figure 2.1.


```
1 | let name = f(100);
```
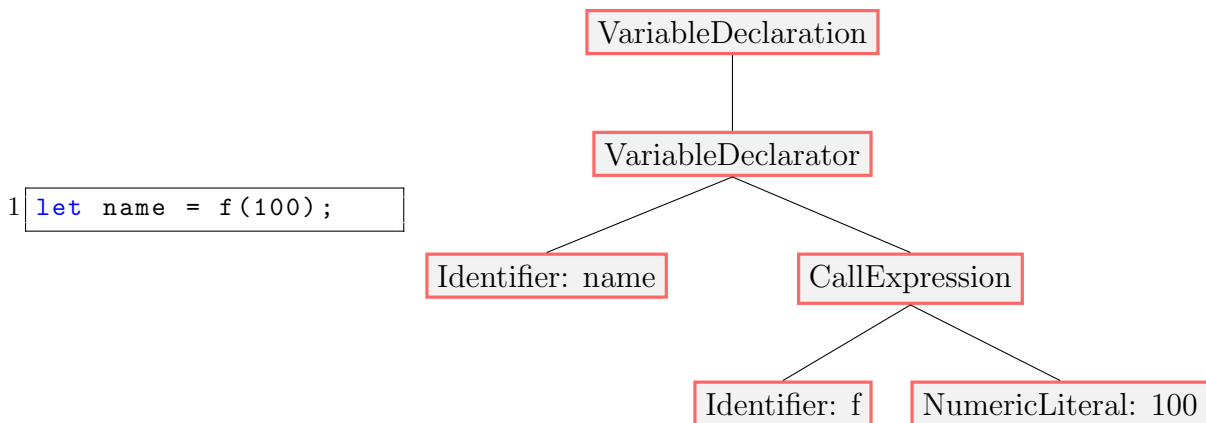
Figure 2.1: Example of source code parsed to Babel AST

Babel's mission is to transpile newer version of JavaScript into older versions that are more compatible with different environments. It has a rich plugin system to allow a myriad of features to be enabled or disabled. This makes the parser very versatile to fit different ways of working with JavaScript source code. This plugin system is built to enable or disable several language constructs,

One of Babel's more prominent features is `@babel/parse` [6] with plugins. This library allows parsing of JavaScript experimental features. These features are usually proposals that are under development by TC39, and the development of these plugins are a part of the proposal deliberation process. This allows for experimentation as early as stage one of the proposal development process. Some examples of proposals that were first supported by Babels plugin system are "Do Expression" [17] and "Pipeline" [15]. These proposals are both currently in the very early stage of development, with "Do Expression" being stage one, and "Pipeline" being stage 2.

2.3 Source Code Querying

Source code querying is the action of searching source code to extract some information or find specific sections of code. This is primarily done using several varying techniques, and is a core part of many tools developers use. The primary use cases for source code querying is code understanding, analysis, code navigation, enforcement of styles along with others. All these are important tools developers use when writing programs, and they all rely on some form of source code queries.

Source code querying comes in many forms, the simplest of which is text search. Since source code is primarily text, one can apply text search techniques to perform a query, this can be regular string search like with CTRL+F in the browser, or a more complex approach using regular expressions with tools like grep. Both these methods cannot allow for queries based on the structure of the code, and rely solely on its syntax. AST based queries allow queries to be written based on both syntax and structure, and are generally more powerful than regular text based queries. Another technique for code querying is creating queries based on semantics of code. Recently, querying based on the semantics of code is more feasible by using large language models to perform the queries.

Source code querying is used in many areas of software development. Some of the more prevalent areas is in Integrated Development Environments (IDEs), as these tools are created to write source code, and therefore rely on querying of the source code written for many of their features. Some of these features include code navigation, static code analysis, or complex code searching. One such example of code querying being used in an IDE is Jetbrains structural search and replace [20], where we define queries based on code structure to find and replace sections of our program.

2.4 Domain Specific languages

Domain specific languages are computer languages specialized to a specific domain. If we compare a DSL to a general purpose language like Python, C++ or JavaScript, these GPL are not designed with a specific task in mind, but have a more general feature set to allow them to be used in a wide array of applications. What a domain is for a DSL is not so simple to define, as there is no general way to define exactly the point in which a DSL becomes a GPL and vice versa. This difference is defined more like a spectrum, in which DSL is on one end and GPL is on the other [33].

DSL's has some clear advantages when being applied to a specific domain compared to GPL's. A DSL allows for very concise and expressive code to be written that is specifically designed for the application, in which a GPL might require specific implementations to suit the domain. Using a DSL might result in faster development because of this expressiveness within the domain, this specificity to a domain might also increase correctness. However, there are also clear disadvantages to DSL's, the restrictiveness of a DSL might become a hinderance if it is not well designed to the domain. DSL's also might have a learning curve, making the knowledge required to use them a hinderance. Developing the DSL might also be a hinderance, as a DSL requires both knowledge of the domain and knowledge of language design.

2.5   Language Workbenches

A language workbench is a tool created to facilitate the development of a computer language, such as a DSL. Language workbenches also create tooling for languages defined within them, and help with the language development process in general.

Language workbenches support generating tooling for languages, as most modern computer languages are backed by some form of tooling. This tooling comes in the form of language parsing, language servers for integrated development environments, along with other tooling for using the language created within the language workbench.

A language is defined in a language workbench using a grammar definition. This grammar is a formal specification of the language that describes how each language construct is composed and the structure of the language. This allows the language workbench to determine what is a valid sentence of the language. This grammar is used to create the the AST of the language, which is the basis for all the tools generated by the language workbench. Many such language workbenches exist, such as Langium [13], Xtext [29], Jetbrains MPS, and Racket.

Chapter 3

Collecting User Feedback for Syntactic Proposals

The goal for this project is to utilize users familiarity with their own code to gain early and worthwhile user feedback on new syntactic proposals for ECMAScript.

3.1 The core idea

When a use of ECMAScript wants to suggest a change to the language, the idea of the change has to be described in a Proposal. A proposal is a general way of describing a change and its requirements, this is done by a language specification, motivation for the idea, and general discussion around the proposed change. A proposal ideally also needs backing from the community of users that use ECMAScript, this means the proposal has to be presented to users some way. This is currently done by many channels, such as polyfills, code examples, and as beta features of the main JavaScript engines, however, this paper wishes to showcase proposals to users by using a different avenue.

Users of ECMAScript have a familiarity with code they themselves have written. This means they have knowledge of how their own code works and why they might have written it a certain way. This project aims to utilize this pre-existing knowledge to showcase new proposals for ECMAScript. This way will allow users to focus on what the proposal actually entails, instead of focusing on the examples written by the proposal authors.

Further in this chapter, we will be discussing the current version and future version of ECMAScript. What we are referring to in this case is with set of problems a proposal is trying to solve, if that proposal is allowed into ECMAScript as part of the language, there will be a future way of solving said problems. The current way is the current status quo when the proposal is not part of ECMAScript, and the future version is when the proposal is part of ECMAScript and we are utilizing the new features of said proposal.

The program will allow the users to preview proposals way before they are part of the language. This way the committee can get useful feedback from users of the language earlier in the proposal process. Using the users familiarity will ideally allow for a more efficient process developing ECMAScript.

3.1.1 Applying a proposal

The way this project will use the pre-existing knowledge a user has of their own code is to use that code as base for showcasing a proposals features. Using the users own code as base requires the following steps in order to automatically implement the examples that showcase the proposal inside the context of the users own code.

The ide is to identify where the features and additions of a proposal could have been used. This means identifying parts of the users program that use pre-existing ECMAScript features that the proposal is interacting with and trying to solve. This will then identify all the different places in the users program the proposal can be applied. This step is called *matching* in the following chapters

Once we have matched all the parts of the program the proposal could be applied to, the users code has to be transformed to use the proposal, this means changing the code to use a possible future version of JavaScript. This step also includes keeping the context and functionality of the users program the same, so variables and other context related concepts have to be transferred over to the transformed code.

The output of the previous step is then a set of code pairs, where one a part of the users original code, and the second is the transformed code. The transformed code is then ideally a perfect replacement for the original user code if the proposal is part of ECMAScript. These pairs are used as examples to present to the user, presented together so the user can see their original code together with the transformed code. This allows for a direct comparison and an easier time for the user to understand the proposal.

The steps outlined in this section require some way of defining matching and transforming of code. This has to be done very precisely and accurately in order to avoid examples that are wrong. Imprecise definition of the proposal might lead to transformed code not being a direct replacement for the code it was based upon. For this we suggest two different methods, a definition written in a custom DSL JSTQL and a definition written in a self-hosted way only using ECMAScript as a language as definition language. Read more about this in SECTION HERE.

3.2 Applicable proposals

A proposal for ECMAScript is a suggested change for the language, in the case of ECMAScript this comes in the form of an addition to the language, as ECMAScript does not allow for breaking changes. There are many different kinds of proposals, this project focuses exclusively on Syntactic Proposals.

3.2.1 Syntactic Proposals

A syntactic proposal, is a proposal that contains only changes to the syntax of a language. This means, the proposal contains either no, or very limited change to functionality, and no changes to semantics. This limits the scope of proposals this project is applicable to, but it also focuses solely on some of the most challenging proposals where the users of the language might have the strongest opinions.

3.2.2 Simple example of a syntactic proposal

Consider an imaginary proposal **declare numerical literal**. This proposal describes adding an optional keyword for declaring numerical variables if the expression of the declaration is a numerical literal.

This proposal will look something like this:

```
1  // Original code
2  let x = 100;
3  let b = "Some String";
4  let c = 200;
5
6  // Code after application of proposal
7  int x = 100;
8  let b = "Some String";
9  let c = 200;
```

Listing 3.1: Example of imaginary proposal **declare numerical literal**

See that in 3.1 the change is optional, and is not applied to the declaration of *c*, but it is applied to the declaration of *x*. Since the change is optional to use, and essentially is just *syntax sugar*, this proposal does not make any changes to functionality or semantics, and can therefore be categorized as a syntactic proposal.

3.2.3 "Pipeline" Proposal

The "Pipeline" proposal [15] is a syntactic proposal which focuses on solving problems related to nesting of function calls and other expressions that take an expression as an argument.

This proposal aims to solve two problems with performing consecutive operations on a value. In *ECMAScript* there are two main styles of achieving this functionality currently: nesting calls and chaining calls, each of them come with a differing set of challenges when used.

Nesting calls is mainly an issue related to function calls with one or more arguments. When doing many calls in sequence the result will be a *deeply nested* call expression.

Using nested calls has some specific challenges related to readability. The order of calls is from right to left, which is the opposite of the natural reading direction a lot of the users of ECMAScript are used to day to day. This means it is difficult to switch the reading direction when working out which call happens in which order. When using functions with multiple arguments in the middle of the nested call, it is not intuitive to see what call its arguments belong to. These issues are the main challenges this proposal is trying to solve. There are currently ways to improve readability with nested calls, as they can be simplified by using temporary variables. While this does introduce its own set of issues, it provides some way of mitigating the readability problem. Another

positive side of nested calls is they do not require a specific design to be used, and a library developer does not have to design their library around this specific call style.

```
1 // Deeply nested call with
    ↪ single arguments
2 f1(f2(f3(f4(v))));
```

```
1 // Deeply nested call with
    ↪ multi argument functions
2 f1(v5, f2(f3(v3, f4(v1, v2)),
    ↪ v4), v6);
```

Chaining solves some of these issues: indeed, as it allows for a more natural reading direction left to right when identifying the sequence of call, arguments are naturally grouped together with their respective function call, and it provides a way of untangling deep nesting. However, executing consecutive operations using chaining has its own set of challenges. To use chaining, the API of the code being called has to be designed to allow for chaining. This is not always the case however, making use of chaining when it has not been specifically designed for can be very difficult. There are also concepts in JavaScript not supported when using chaining, such as arithmetic operations, literals, `await` expressions, `yield` expressions and so on. This is because all of these concept would "break the chain", and one would have to use temporary variables.

```
1 // Chaining calls
2 function1().function2().function3();
3
4 // Chaining calls with multiple arguments
5 function1(value1).function2().function3(value2).function4();
```

The "Pipeline" proposal aims to combine the benefits of these two styles without the challenges each method faces. The main benefit of the proposal is to allow for a similar style to chaining when chaining has not been specifically designed to be applicable. The essential idea is to use syntactic sugar to change the writing order of the calls without influencing the API of the functions. Doing so will allow each call to come in the direction of left to right, while still maintaining the modularity of deeply nested function calls.

The proposal introduces a *pipe operator*, which takes the result of an expression on the left, and pipes it into an expression on the right. The location of where the result is piped to is where the topic token is located. All the specifics of the exact token used as a topic token and exactly what operator will be used as the pipe operator might be subject to change, and is currently under discussion [16].

The code snippets below showcase the machinery of the proposal.

```
1 // Status quo
2 var loc =
    ↪ Object.keys(grunt.config(
    ↪ "uglify.all" ))[0];
```

```
1 // With pipes
2 var loc =
    ↪ grunt.config('uglify.all')
    ↪ |> Object.keys(%)[0];
```

More intuitive ordering of function calls, to know exactly the order of execution.

```
1  // Status quo
2  const json = await
     ↪ npmFetch.json(
3        npa(pkgs[0]).escapedName,
           ↪ opts);
```

```
1  // With pipes
2  const json = pkgs[0] |>
     ↪ npa(%).escapedName |>
     ↪ await npmFetch.json(%,
     ↪ opts);
```

Seeing which argument is passed to which function call is is simpler when using pipes.

```
1  // Status quo
2  return filter(obj,
     ↪ negate(cb(predicate)),
     ↪ context);
```

```
1  // With pipes
2  return cb(predicate) |>
     ↪ _.negate(%) |>
     ↪ _.filter(obj, %, context);
```

Can be used with any number of function arguments, as long as a single topic token is used.

```
1  // Status quo
2  return
     ↪ xf['@@transducer/result'](obj
     ↪ xf), acc));
```

```
1  // With pipes
2  return xf
3      |>
           ↪ bind(%['@@transducer/step'],
           ↪ %)                        /step'],
4      |> obj[methodName](%, acc)
5      |>
           ↪ xf['@@transducer/result'](%);
```

Complex call expressions are unraveled with pipes.

The pipe operator is present in many other languages such as F# [32] and Julia [12]. The main difference between the Julia and F# pipe operator compared to this proposal, is the result of the left side expression has to be piped into a function with a single argument, the proposal suggests a topic reference to be used in stead of requiring a function.

3.2.4 "Do Expression"

The "Do Expression" [17] proposal, is a proposal meant to bring a style of *expression oriented programming* [27] to ECMAScript. Expression oriented programming is a concept taken from functional programming which allows for combining expressions in a very free manner, resulting in a highly malleable programming experience.

The motivation of the "Do Expression" proposal is to allow for local scoping of a code block that is treated as an expression. Thus, complex code requiring multiple statements will be confined inside its own scope [26, 8.2] and the resulting value is returned from the block implicitly as an expression, similarly to how a unnamed functions or arrow

functions are currently used. In order to achieve this behavior in the current stable version of ECMAScript, one needs to use immediately invoked unnamed functions [26, 15.2] and invoke them immediately, or use an arrow function [26, 15.3].

The codeblock of a `do` expression has one major difference from these equivalent functions, as it allows for implicit return of the final statement of the block, and is the resulting value of the entire `do` expression. The local scoping of this feature allows for a cleaner environment in the parent scope of the `do` expression. What is meant by this is for temporary variables and other assignments used once can be enclosed inside a limited scope within the `do` block. Allowing for a cleaner environment inside the parent scope where the `do` block is defined.

```
1  // Current status quo
2  let x = () => {
3      let tmp = f();
4      return tmp + tmp + 1;
5  };
```

```
1  // With do expression
2  let x = do {
3      let tmp = f();
4      tmp + tmp + 1;
5  };
```

The current version of JavaScript enables the use of arrow functions with no arguments to achieve similar behavior to "Do Expression". The main difference in this case, is the final statement/expression will implicitly return it's Completion Record [26, 6.2.4]

```
1  // Current status quo
2  let x = function(){
3      let tmp = f();
4      let a = g() + tmp;
5      return a - 1;
6  }();
```

```
1  // With do expression
2  let x = do {
3      let tmp = f();
4      let a = g() + tmp;
5      a - 1;
6  };
```

This example is very similar, as it uses an unnamed function [26, 15.2] which is invoked immediately to produce similar behavior to the "Do Expression" proposal.

3.2.5   Await to Promise

We discuss now an imaginary proposal that was used as a running example during the development of this thesis. This proposal is of just a pure JavaScript transformation example. The transformation this proposal is meant to display, is transforming a code using `await` [26, 27.7.5.3], into code which uses a promise [26, 27.2].

To perform this transformation, we define an equivalent way of expressing an `await` expression as a promise. This means removing `await`, this expression now will return a promise, which has a function `then()`, this function is executed when the promise

resolves. We pass an arrow function as argument to `then`, and append each following statement in the current scope [26, 8.2] inside the block of that arrow function. This will result in equivalent behavior to using `await`.

```
1  // Code containing await
2  async function a(){
3      let b = 9000;
4      let something = await
           ↪ asyncFunction();
5      let c = something + 100;
6      return c + 1;
7  }
```

```
1  // Re-written using promises
2  async function a(){
3      let b = 9000;
4      return asyncFunction()
5          .then(async (something)
               ↪ => {
6          let c = something + 100;
7          return c;
8      })
9  }
```

Transforming using this imaginary proposal, will result in a returning the expression present at the first `await` expression, with a deferred function `then`, that will execute once the expression is completed. This function `then` takes a callback containing a lambda function with a single argument. This argument shares a name with the initial `VariableDeclaration`. This is needed because we have to transfer all statements that occur after the original `await` expression into the body of the callback function. This callback function also has to be async, in case any of the statements placed into it contains `await`. This will result in equivalent behavior to the original code.

3.3 Searching user code for applicable snippets

In order to identify snippets of code in the user's code where a proposal is applicable, we need some way to define patterns of code to use as a query. To do this, we have designed and implemented a domain-specific language that allows matching parts of code that is applicable to some proposal, and transforming those parts to use the features of that proposal.

3.3.1 Structure of JSTQL

In this section, we describe the structure of JSTQL . We describe every section of the language, why each section is needed and what it is used for.

15

Proposal definition. JSTQL is designed to mimic the examples already provided in proposal descriptions [22]. These examples can be seen in each of the proposals described in Section 3.2. The idea is to allow a similar kind of notation to the examples in order to define the transformations.

The first part of JSTQL is defining the proposal, this is done by creating a named block containing all definitions of templates used for matching alongside their respective transformation. This section is used to contain everything relating to a specific proposal and is meant for easy proposal identification by tooling.

```
1  proposal Pipeline_Proposal {}
```

Case definition. Each proposal will have one or more definitions of a template for code to identify in the users codebase, and its corresponding transformation definition. These are grouped together in order to have a simple way of identifying the corresponding cases of matching and transformations. This section of the proposal is defined by the keyword *case* and a block that contains its related fields. A proposal definition in JSTQL should contain at least one `case` definition. This allows for matching many different code snippets and showcasing more of the proposal than a single concept the proposal has to offer.

```
1      case case_name {
2
3      }
```

Template used for matching In order to define the template used to match, we have another section defined by the keyword *applicable to*. This section will contain the template defined using JavaScript with specific DSL keywords defined inside the template. This template is used to identify applicable parts of the user's code to a proposal.

```
1  applicable to {
2      "let a = 0;"
3  }
```

This `applicable to` template, will create matches on any `VariableDeclaration` that is initialized to the value 0, and is stored in an `Identifier` with name `a`.

16

Defining the transformation In order to define the transformation that is applied to a specific matched code snippet, the keyword *transform to* is used. This section is similar to the template section, however it uses the specific DSL identifiers defined in applicable to, in order to transfer the context of the matched user code, this allows us to keep parts of the users code important to the original context it was written in.

```
transform to{
    "() => {
        let b = 100;
    }"
}
```

This transformation definition, will change any code matched to its corresponding matching definition into exactly what is defined. This means for any matches produced this code will be inserted in its place.

Full definition of JSTQL Taking all these parts of JSTQL structure, defining a proposal in JSTQL will look as follows.

```
proposal PROPOSAL_NAME {
    case CASE_NAME_1 {
        applicable to {
            "let b = 100;"
        }
        transform to {
            "() => {};"
        }
    }
    case CASE_NAME_2 {
        applicable to {
            "console.log();"
        }
        transform to {
            "console.dir();"
        }
    }
}
```

Listing 3.2: JSTQL definition of a proposal

This full example of JSTQL has two `case` sections. Each `case` is applied one at a time to the user's code. The first case will try to find any `VariableDeclaration` statements, where the identifier is `b`, and the right side expression is a `Literal` with value 100. The second `case` will change any empty `console.log` expression, into a `console.dir` expression.

3.3.2 How a match and transformation is performed

To perform matching and transformation of the user's code, we first have to have some way of identifying applicable user code. These applicable code sections then have to be transformed and inserted it back into the full user code definition.

Identifying applicable code

To identify sections of code a proposal is applicable to, we use *templates*, which are snippets of JavaScript. These templates are used to identify and match applicable sections of a users code. A matching section for a template is one that produces an exactly equal AST structure, where each node of the AST sections has the same information contained within it. This means that templates are matched exactly against the users code, this does not really provide some way of querying the code and performing context based transformations, so for that we use *wildcards* within the template.

Wildcards are interspliced into the template inside a block denoted by `<< >>`. Each wildcard starts with an identifier, which is a way of referring to that wildcard in the definition of the transformation template later. This allows for transferring the context of parts matched to a wildcard into the transformed output, like identifiers, parts of statements, or even entire statements, can be transferred from the original user code into the transformation template. A wildcard also contains a type expression. A type expression is a way of defining exactly the types of AST nodes a wildcard will produce a match against. These type expressions use Boolean logic together with the AST node-types from BabelJS [3] to create a very versatile of defining exactly what nodes a wildcard can match against.

Wildcard type expressions

Wildcard expressions are used to match AST node types based on Boolean logic. This Boolean logic is based on comparison of Babel AST node types [4]. We do this because we need an accurate and expressive way of defining specifically what kinds of AST nodes a wildcard can be matched against. This means an type expression can be as simple as `VariableDeclaration`: this will match only against a node of type `VariableDeclaration`. We also special types for `Statement` for matching against a statement, and `Expression` for matching any expression.

This example will allow any `CallExpression` to match against this wildcard named `expr`.

```
1  << expr: CallExpression >>
```

To make this more expressive, the type expressions support binary and unary operators.We support the following operators, `&&` is logical conjunction, `||` means logical disjunction,`!` is logical negation. This makes it possible to build complex type expressions, making it very expressive exactly what nodes are allowed to match against a specific wildcard.

In the first example on line 1, we want to limit the wildcard to not match against any nodes with type `VariableDeclaration`, while still allowing any other `Statement`. The example on line 2 want to avoid loop specific statements. We express this by allowing any `Statement`, but we negate the expression containing the types of loop specific statements.

```
1  << notVariableDeclaration: Statement && !VariableDeclaration >>
2  << noLoopSpecificStatements: Statement && !(BreakStatement ||
   ↪ ContinueStatement) >>
```

The wildcards support matching subsequent sibling nodes of the code against a single wildcard. We achieve this behavior done by using a Keene plus at the top level of the expression. A Keene plus means one or more, so we allow for one or more matches in order when using this token. This is useful for matching against a series of one or more specific nodes, the matching algorithm will continue to match until the type expression no longer evaluates to true.

In the example below, we allow the wildcard to match multiple nodes with the Keene plus `+`. This example will continue to match against itself as long as the nodes are a `Statement` and at the same time is not a `ReturnStatement`.

```
1  << statementsNoReturn : (Statement && !ReturnStatement)+ >>
```

```
1  let variableName = << expr1: ((CallExpression || Identifier) &&
   ↪ !ReturnStatement)+ >>;
```

A wildcard section is defined on the right hand side of an assignment statement. This wildcard will match against any AST node classified as a CallExpression or an Identifier.

3.3.3 Transforming

When matching sections of the users code has been found, we need some way of defining how to transform those sections to showcase a proposal. This is done using the `transform to` template. This template describes the general structure of the newly transformed code, with context from the users code by using wildcards.

A transformation template defines how the matches will be transformed after applicable code has been found. The transformation is a general template of the code once the match is replaced in the original AST. However, without transferring over the context from the match, this would be a template search and replace. Thus, in order to transfer the context from the match, wildcards are defined in this template as well. These wildcards use the same block notation found in the `applicable to` template, however they do not need to contain the types, as those are not needed in the transformation. The only required field of the wildcard is the identifier defined in `applicable to`. This is done in order to know which wildcard match we are taking the context from, and where to place it in the transformation template.

Transforming a variable declaration from using `let` to use `const`.

```
1  // Example applicable to template
2  applicable to {
3      let <<variableName: Identifier>> = <<expr1: Expression>>;
4  }
5
6  // Example of transform to template
7  transform to {
8      const <<variableName>> = <<expr1>>;
9  }
```

3.3.4 Using JSTQL

JSTQL is designed to be used at a proposal development stage, this means the users of JSTQL will most likely be TC39 [21] delegates, or otherwise relevant stakeholders.

JSTQL is designed to closely mimic the style of the examples required in the TC39 process [22]. We chose to design it this way to specifically make this tool fit the use-case of the committee. The idea behind this project is to gather early user feedback on syntactic proposals, this would mean the main users of this kind of tool is the committee themselves.

JSTQL is just written using text, most Domain-specific languages have some form of tooling to make the process of using the DSL simpler and more intuitive. JSTQL has an extension built for Visual Studio Code, see Figure 3.1, this extension supports many common features of language servers, it supports auto completion, it will produce errors if fields are defined wrong or are missing parameters.
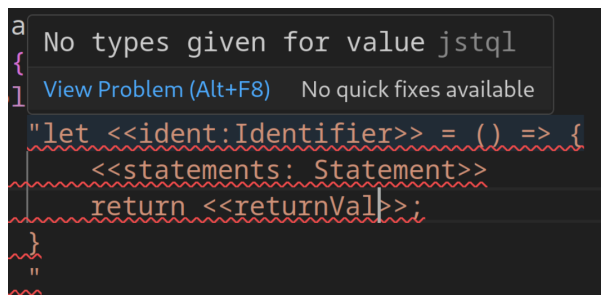
```
1   proposal Star{
2       case a {
3           applicable to {
4               "let <<ident:Identifier>> = () => {
5                   <<statements: Statement>>
6                   return <<returnVal : Expression>>;
7               }
8               "
9           }
10          transform to {
11              "let <<ident>> = do {
12                  <<statements>>
13                  <<returnVal>>
14              }"
15          }
16      }
17  }
```

Figure 3.1: Writing JSTQL in Visual Studio Code with extension

The language server included with this extension performs validation of the wildcards. This allows verification of wildcard declarations in applicable to, see Figure 3.2. If a wildcard is declared with no types, an error will be reported.

Figure 3.2: Error displayed when declaring a wildcard with no types.

The extension automatically uses wildcard declarations in `applicable to` to verify all wildcards referenced in `transform to` are declared. If an undeclared wildcard is used, an error will be reported and the name of the undeclared wildcard will be displayed, see Figure 3.3.
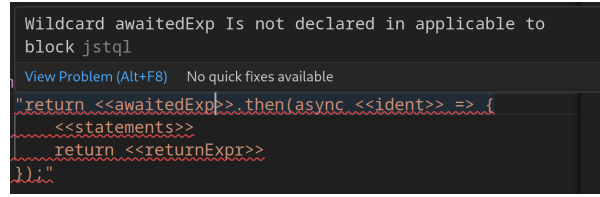
Figure 3.3: Error displayed with usage of undeclared wildcard.

3.4 Using the JSTQL with syntactic proposals

This section contains the definitions of the proposals used to evaluate the tool created in this thesis. These definitions do not have to cover every single case where the proposal might be applicable, as they just have to be general enough to create some amount of examples that will give a representative number of matches when the transformations are applied to some relatively long user code. This is because this this tool will be used to gather feedback from user's on proposals during development. Because of this use case, it does not matter that we catch every single applicable code snippet, just that we find enough to perform a "showcase" of the proposal to the user. The most important thing is that the transformation is correct, as incorrect transformations will lead to bad feedback on the proposal.

3.4.1 "Pipeline" Proposal

The "Pipeline" proposal is one of the proposals presented in Section 3.2. This proposal is applicable to call expressions, which are used all across JavaScript. This proposal is trying to solve readability when performing deeply nested function calls.

```
 1 proposal Pipeline {
 2
 3     case SingleArgument {
 4         applicable to {
 5             "<<someFunctionIdent:Identifier ||
                    ↪ MemberExpression>>(<<someFunctionParam:
                    ↪ Expression>>);"
 6         }
 7
 8         transform to {
 9             "<<someFunctionParam>> |> <<someFunctionIdent>>(%);"
10         }
11     }
12
13     case TwoArgument{
14         applicable to {
```

22

```
15            "<<someFunctionIdent: Identifier ||
              ↪ MemberExpression>>(<<someFunctionParam:
              ↪ Expression>>, <<moreFunctionParam: Expression>>)"
16        }
17        transform to {
18            "<<someFunctionParam>> |> <<someFunctionIdent>>(%,
              ↪ <<moreFunctionParam>>)"
19        }
20    }
21 }
```

Listing 3.3: Example of "Pipeline" proposal definition in JSTQL

In the Listing 3.3, the first pair definition `SingleArgument` will apply to any
`CallExpression` with a single argument. We do not expressively write a `CallExpression`
inside a wildcard, as we have defined the structure of a `CallExpression`. The first wild-
card `someFunctionIdent`, has the types of `Identifier`, to match against single identi-
fiers, and `MemberExpression`, to match against functions who are members of objects,
i.e. `console.log`. In the transformation template, we define the structure of a function
call using the pipe operator, but the wildcards change order, so the argument passed as
argument `someFunctionParam` is placed on the left side of the pipe operator, and the
`CallExpression` is on the right, with the topic token as the argument. This case will
produce a match against all function calls with a single argument, and transform them to
use the pipe operator. The main difference of the second case `TwoArgument`, is it matches
against functions with exactly two arguments, and uses the first argument as the left side
of the pipe operator, while the second argument remains in the function call.

3.4.2   "Do Expressions" Proposal

The "Do Expressions" proposal [17] can be specified in our DSL. Due to the nature of
the proposal, it is not as applicable as the "Pipeline" proposal, as it does not re-define
a style that is used quite as frequently as call expressions. This means the amount of
transformed code snippets this specification in JSTQL will be able to perform is expected
to be lower. This is due to the "Do Expression" proposal introducing an entirely new
way to write expression-oriented code in JavaScript. If the user running this tool has not
used the current way of writing in an expression-oriented style in JavaScript, JSTQL is
limited in the amount of transformations it can perform. Nevertheless, if the user has
been using an expression-oriented style, JSTQL will transform parts of the code.

```
1 proposal DoExpression {
2     case arrowFunction {
3         applicable to {
4             "() => {
5                 <<statements: (Statement && !ReturnStatement)+>>
```

```
 6              return <<returnVal : Expression>>;
 7          }
 8          "
 9      }
10      transform to {
11          "(do {
12              <<statements>>
13              <<returnVal>>
14          })"
15      }
16  }
17
18  case immediatelyInvokedAnonymousFunction {
19      applicable to {
20          "(function(){
21              <<statements: (Statement && !ReturnStatement)+>>
22              return <<returnVal : Expression>>;
23          })();"
24      }
25
26      transform to {
27          "(do {
28              <<statements>>
29              <<returnVal>>
30          })"
31      }
32  }
33 }
```

Listing 3.4: Definition of Do Proposal in JSTQL

In Listing 3.4, the specification of "Do Expression" proposal in JSTQL can be seen. It has two cases: the first case `arrowFunction`, applies to a code snippet using an arrow function [26, 15.3] with a return value. The wildcards of this template are `statements`, which is a wildcard that matches against one or more statements that are not of type `ReturnStatement`, the reason we limit the one or more match is we cannot match the final statement of the block to this wildcard, as that has to be matched against the return statement in the template. The second wildcard `returnVal` matches against any expressions. The reason for extracting the expression from the `return` statement, is to use it in the implicit return of the `do` block. In the transformation template, we replace the arrow function with with a `do` expression, this do expression has to be defined inside parenthesis, as a free floating do expression is not allowed due to ambiguous parsing against a `do  while()` statement. We and insert the statements matched against `statements` wildcard into the block of the `do` expression, and the final statement of the block is the expression matched against the `returnVal` wildcard. This will produce an equivalent transformation of an arrow function into a `do` expression. The second case `immediatelyInvokedAnonymousFunction` follows the same principle as the first case, but is applied to immediately invoked anonymous functions, and produces the exact same output after the transformation as the first case. This is because immediately invoked anonymous functions are equivalent to arrow functions.

24

3.4.3 "Await to Promise" imaginary proposal

The imaginary proposal "Await to Promise" is created to transform code snippets from using `await`, to use a promise with equivalent functionality.

This proposal was created in order to evaluate the tool, as it is quite difficult to define applicable code in this current template form. This definition is designed to create matches in code using await, and highlight how await could be written using a promise in stead. This actually highlights some of the issues with the current design of JSTQL that will be described in Future Work.

```
1  proposal awaitToPomise{
2      case single{
3          applicable to {
4              "let <<ident:Identifier>> = await <<awaitedExpr:
                 ↪ Expression>>;
5              <<statements: (Statement && !ReturnStatement &&
                 ↪ !ContinueStatement &&!BreakStatement)+>>
6              return <<returnExpr: Expression>>
7              "
8          }
9
10         transform to{
11             "return <<awaitedExpr>>.then(async <<ident>> => {
12                 <<statements>>
13                 return <<returnExpr>>
14             });"
15         }
16     }
17 }
```

Listing 3.5: Definition of Await to Promise evaluation proposal in JSTQL

The specification of "Await to Promise" in JSTQL is created to match asynchronous code inside a function. It is limited to match asynchronous functions containing a single await statement, and that await statement has to be stored in a `VariableDeclaration`. The second wildcard `statements`, is designed to match all statements following the `await` statement up to the return statement. This is done to move the statements into the callback function of `then()` in the transformation. We include `ReturnStatement` because we do not want to consume the return as it would then be removed from the functions scope and into the callback function of `then()`. We also have to avoid matching where there exists loop specific statements such as `ContinueStatement` or `BreakStatement`.

The transformation definition has to use an async function in `.then()`, as there might be more await expressions contained within `statements`.

3.5 JSTQL-SH

In this thesis, we also created an alternative way of defining proposals and their respective transformations, this is done using JavaScript as it's own meta language for the definitions. The reason for creating a way of defining proposals using JavaScript is, it allows us to limit the amount of dependencies of the tool, since we no longer rely on JSTQL , and it allows for more exploration in the future work of this project.

JSTQL-SH is less of an actual language, and more of a program API at the moment, it allows for defining proposals purely in JavaScript objects, which is meant to allow a more modular way of using this idea. In JSTQL-SH you define a *prelude*, which is just a list of variable declarations that contain the type expression as a string for that given wildcard. This means we do not need to perform wildcard extraction when wanting to parse the templates used for matching and transformation.

```
1  // Definition in JSTQL
2  proposal a{
3      case {
4          applicable to {
5              <<a:Expression>>
6          }
7          transform to {
8              () => <<a>>
9          }
10     }
11 }
```

```
1  // Equivalent definition in
   ↪  JSTQL-SH
2  {
3      prelude: 'let a =
       ↪  "Expression"'`,
4      applicableTo: "a;",
5      transformTo: "() => a;"
6  }
```

Chapter 4

Implementation

In this chapter, the implementation of the tool utilizing the JSTQL and JSTQL-SH will be presented. It will describe the overall architecture of the tool, the flow of data throughout, and how the different stages of transforming user code are completed.

4.1 Architecture of the solution

The architecture of the solution described in this thesis is illustrated in Figure 4.1

In this tool, there exists two multiple ways to define a proposal, and each provide the same functionality, they only differ in syntax and writing-method. One can either write the definition in JSTQL , or one can use the program API with JSTQL-SH , which is more friendly for programs to interact with.

In the architecture diagram of Figure 4.1, ellipse nodes show data passed into the program sections, and rectangular nodes is a specific section of the program. The architecture is split into seven levels, where each level is a step of the program. The initial step is the proposal definition, the definition can have two different forms, either it is JSTQL code, or it can be a JavaScript object using the self hosted in JSTQL-SH . If we use JSTQL , the second step is parsing it using Langium [13], this parses the raw source code into an AST. If JSTQL-SH is used, we have to build the prelude, so we have to extract the wildcard definitions from JavaScript source code. At this point the two paths meet at the second step, which is wildcard extraction. At this step, if JSTQL was used, the wildcards are extracted from the template. If JSTQL-SH was used extraction is not

needed. In both cases we parse the wildcard type expressions into an AST. The third step is parsing the raw source code with Babel [3]. It is also at this point we parse the users source code into an AST. The fourth step is translating the Babel AST into our own custom tree structure for simpler traversal. Once all data is prepared, the fifth step is matching the user's AST against the `applicable to` template AST. Once all matches have been found, we transplant the wildcard matches into the `transform to` template, and insert it back into the users code in step six. We have at this point transformed the users code, the final step seven is generating it back into source code.
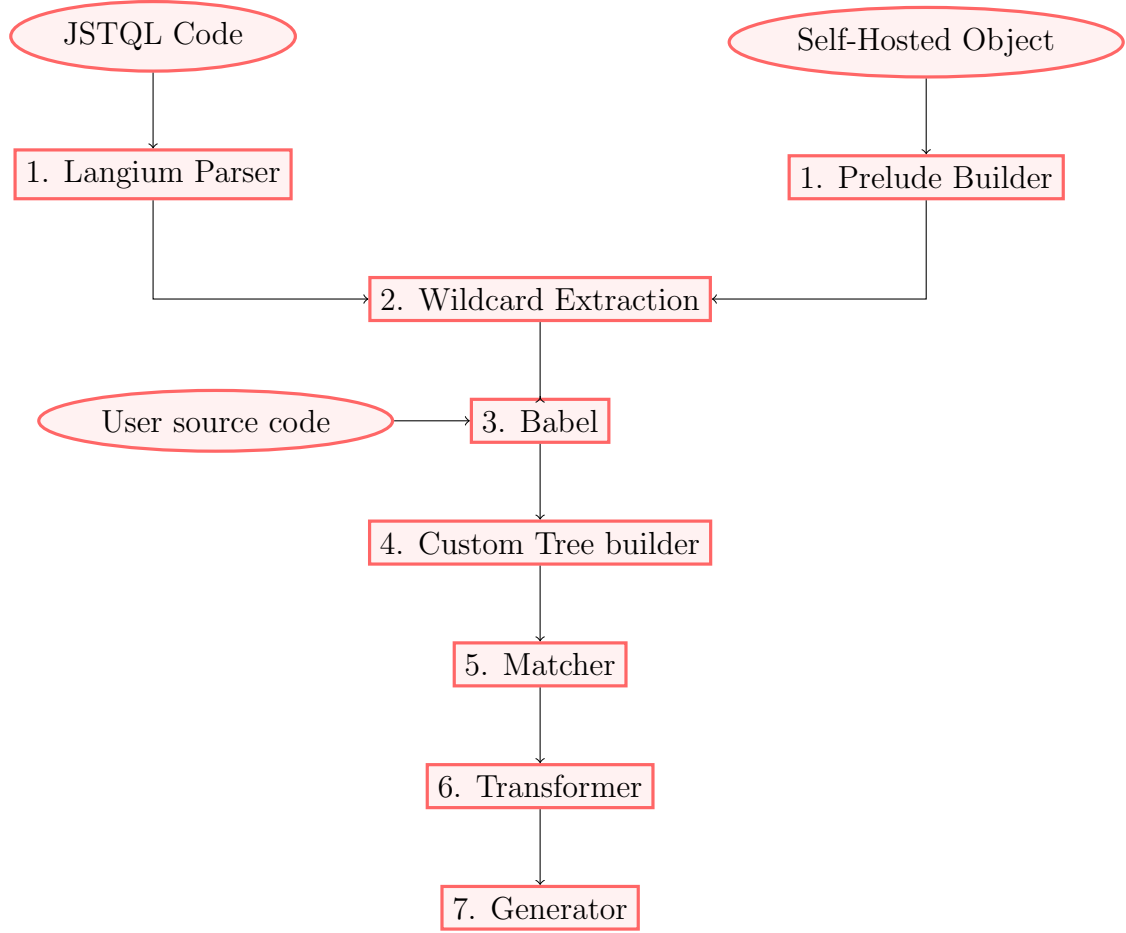

Figure 4.1: Overview of tool architecture

4.2 Parsing JSTQL using Langium

In this section, we describe the implementation of the parser for JSTQL . We outline the tool Langium, used as a parser-generator to create the AST used by the tool later to perform the transformations.

4.2.1 Langium

Langium [13] is a language workbench [30] primarily used to create parsers and Integrated Development Environments for domain specific languages. These kinds of parsers produce Abstract Syntax Trees that are later used to create interpreters or other tooling. In this project, we use Langium to generate an AST definition in the form of TypeScript objects. These objects and their structure are used as definitions for the tool to do matching and transformation of user code.

To generate this parser, Langium requires a definition of a grammar. A grammar is a specification that describes syntax a valid programs. The JSTQL grammar describes the structure of JSTQL , such as `proposals`, `cases`, `applicable to` blocks, and `transform to` blocks. A grammar in Langium starts by describing the `Model`. The model is the top entry of the grammar; this is where the description of all valid top level statements.

Contained within the `Model` rule, is one or more proposals. Each proposal is defined with the rule `Proposals`, and starts with the keyword `proposal`, followed by a name, and a code block. This rule is designed to contain every definition of a transformation related to a specific proposal. To hold every transformation definition, a proposal definition contains one or more cases.

The `Case` rule is created to contain a single transformation. Each case specification starts with the keyword `case`, followed by a name for the current case, then a block for that case's fields. Cases are designed in this way to separate different transformation definitions within a proposal. Each case contains a single definition used to match against user code, and a definition used to transform a match.

The rule `AplicableTo`, is designed to hold a single template used for matching. It starts with the keywords `applicable` and `to`, followed by a block designed to hold the matching template definition. The template is defined as the terminal `STRING`, and is parsed as a raw string for characters by Langium [13].

The rule `TransformTo`, is created to contain a single template used for transforming a match. It starts with the keywords `transform` and `to`, followed by a block that holds the transformation definition. This transformation definition is declared with the terminal `STRING`, and is parser at a string of characters, same as the template in `applicable to`.

In order to define exactly what characters/tokens are legal in a specific definition, Langium uses terminals defined using regular expressions, these allow for a very specific

character-set to be legal in specific keys of the AST generated by the parser generated by Langium. In the definition of `Proposal` and `Pair` the terminal `ID` is used; this terminal is limited to allow for only words and can only begin with a character of the alphabet or an underscore. In `Section` the terminal `STRING` is used, this terminal is meant to allow any valid JavaScript code and the custom DSL language described in 3.3.2. Both these terminals defined allows Langium to determine exactly what characters are legal in each location.

```
grammar Jstql

entry Model:
    (proposals+=Proposal)*;

Proposal:
    'proposal' name=ID "{"
        (case+=Case)+
    "}";

Case:
    "case" name=ID "{"
        aplTo=ApplicableTo
        traTo=TransformTo
    "}";

ApplicableTo:
    "applicable" "to" "{"
        apl_to_code=STRING
    "}";
TransformTo:
    "transform" "to" "{"
        transform_to_code=STRING
    "}";
hidden terminal WS: /\s+/;
terminal ID: /[_a-zA-Z][\w_]*/;
terminal STRING: /"[^"]*"|'[^']*'/;
```

Listing 4.1: Definition of JSTQL in Langium.

With JSTQL , we are not implementing a programming language meant to be executed. We are using Langium in order to generate an AST that will be used as a markup language, similar to YAML, JSON or TOML [24]. The main reason for using Langium in such an unconventional way is Langium provides support for Visual Studio Code integration, and it solves the issue of parsing the definition of each proposal manually. However, with this grammar we cannot actually verify the wildcards placed in `apl_to_code` and `transform_to_code` are correctly written. To do this, we have implemented several validation rules.

Langium Validator

A Langium validator allows for further checks DSL code, a validator allows for the implementation of specific checks on specific parts of the grammar.

JSTQL does not allow empty typed wildcard definitions in `applicable to` blocks, this means we cannot define a wildcard that allows any AST type to match against it. This is not defined within the grammar, as inside the grammar the code is defined as a `STRING` terminal. This means further checks have to be implemented using code. In order to do this we have a specific `Validator` implemented on the `Case` definition of the grammar. This means every time anything contained within a `Case` is updated, the language server created with Langium will perform the validation step and report any errors.

The validator uses `Case` as its entry point, as it allows for a checking of wildcards in both `applicable to` and `transform to`, allowing for a check for whether a wildcard identifier used in `transform to` exists in the definition of `applicable to`.

```
 1 export class JstqlValidator {
 2     validateWildcardAplTo(pair: Pair, accept: ValidationAcceptor):
          ↪ void {
 3        try {
 4            if (validationResultAplTo.errors.length != 0) {
 5                accept("error",
                      ↪ validationResultAplTo.errors.join("\n"), {
 6                    node: pair.aplTo,
 7                    property: "apl_to_code",
 8                });
 9            }
10            if (validationResultTraTo.length != 0) {
11                accept("error", validationResultTraTo.join("\n"), {
12                    node: pair.traTo,
13                    property: "transform_to_code",
14                });
15            }
16        } catch (e) {}
17    }
18 }
```

Using Langium as a parser

Langium is designed to automatically generate extensive tool support for the language specified using its grammar. However, in our case we have to parse the JSTQL definition using Langium, and then extract the Abstract syntax tree generated in order to use the information it contains.

To use the parser generated by Langium, we created a custom function `parseDSLtoAST`, which takes a string as an input (the raw JSTQL code), and outputs the pure AST using the format described in the grammar, see Listing 3.3.2. This function is exposed as a custom API for our tool to interface with. This also means our tool is dependent on the

implementation of the Langium parser to function with JSTQL . The implementation of JSTQL-SH is entirely independent.

When interfacing with the Langium parser to get the Langium generated AST, the exposed API function is imported into the tool, when this API is executed, the output is on the form of the Langium `Model`, which follows the same form as the grammar. This is then transformed into an internal object structure used by the tool, this structure is called `TransformRecipe`, and is then passed in to perform the actual transformation.

4.3 Wildcard extraction and parsing

In order to refer to internal DSL variables defined in `applicable to` and `transform to` blocks of the transformation, we need to extract this information from the template definitions and pass that on to the matcher.

Why not use Langium for wildcard parsing?

Langium has support for creating a generator to output an artifact, which is some transformation applied to the AST built by the Langium parser. This suits the needs of JSTQL quite well and could be used to extract the wildcards and parse the type expressions. This is the way the developers of Langium want this kind of functionality to be implemented, however, the implementation would still be mostly the same, as the parsing of the wildcards still has to be done "manually" with a custom parser. Therefore, we decided for this project to keep the parsing of the wildcards separate. If we were to use Langium generators to parse the wildcards, it would make JSTQL-SH dependent on Langium. This is not preferred as that would mean both ways of defining a proposal are reliant of Langium. The reason for using our own extractor is to allow for an independent way to define transformations using our tool.

Extracting wildcards from JSTQL

In order to allow the use of Babel [3], the wildcards present in the `applicable to` blocks and `transform to` blocks have to be parsed and replaced with some valid JavaScript.

This is done by using a pre-parser that extracts the information from the wildcards and inserts an `Identifier` in their place.

To extract the wildcards from the template, we look at each character in the template. If a start token of a wildcard is discovered, which is denoted by `<<`, everything after that until the closing token, which is denoted by `>>`, is then treated as an internal DSL variable and will be stored by the tool. A variable `flag` is used (line 5,10 4.2), when the value of flag is false, we know we are currently not inside a wildcard block, this allows us to pass the character through to the variable `cleanedJS` (line 196 4.2). When `flag` is true, we know we are currently inside a wildcard block and we collect every character of the wildcard block into `temp`. Once we hit the end of the wildcard block, when we have consumed the entirety of the wildcard, the contents of the `temp` variable is passed to a tokenizer, then the tokens are parsed by a recursive descent parser (line 10-21 4.2).

Once the wildcard is parsed, and we know it is safely a valid wildcard, we insert an identifier into the JavaScript template where the wildcard would reside. This allows for easier identifications of wildcards when performing matching/transformation as we can identify whether or not an Identifier in the code is the same as the identifier for a wildcard. This however, does introduce the problem of collisions between the wildcard identifiers inserted and identifiers present in the users code. In order to avoid this, the tool adds __ at the beginning of every identifier inserted in place of a wildcard. This allows for easier identification of if an Identifier is a wildcard, and avoids collisions where a variable in the user code has the same name as a wildcard inserted into the template. This can be seen on line 17 of Listing 4.2.

```
1  export function parseInternal(code: string): InternalParseResult {
2      for (let i = 0; i < code.length; i++) {
3          if (code[i] === "<" && code[i + 1] === "<") {
4              // From now in we are inside of the DSL custom block
5              flag = true;
6              i += 1;
7              continue;
8          }
9
10         if (flag && code[i] === ">" && code[i + 1] === ">") {
11             // We encountered a closing tag
12             flag = false;
13             try{
14                 let wildcard = new WildcardParser(
15                     new WildcardTokenizer(temp).tokenize()
16                 ).parse();
17                 cleanedJS +=
18                     ↪ collisionAvoider(wildcard.identifier.name);
19                 prelude.push(wildcard);
20                 i += 1;
21                 temp = "";
22                 continue;
23             }
24             catch (e){
```

```
25              // We probably encountered a bitshift operator, append
    ↪ temp to cleanedJS
26          }
27
28      }
29      if (flag) {
30          temp += code[i];
31      } else {
32          cleanedJS += code[i];
33      }
34   }
35   return { prelude, cleanedJS };
36 }
```

Listing 4.2: Extracting wildcard from template.

Parsing wildcard   Once a wildcard has been extracted from definitions inside JSTQL ,
they have to be parsed into a simple AST to be used when matching against the wildcard.
This is accomplished by using a simple tokenizer and a recursive descent parser [28].

Our tokenizer takes the raw stream of input characters extracted from the wildcard
block within the template, and determines which part is what token. Due to the very
simple nature of the type expressions, no ambiguity is present with the tokens, so de-
termining what token is meant to come at what time is quite trivial. We use a switch
case on the current token, if the token is of length one we accept it and move on to the
next character. If the next character is an unexpected one it will produce an error. The
tokenizer also groups tokens with a *token type*, this allows for an simpler parsing of the
tokens later.

A recursive descent parser mimics the grammar of the language the parser is imple-
mented for, where we define functions for handling each of the non-terminals and ways
to determine what non terminal each of the token-types result in. The type expression
language is a very simple Boolean expression language, making parsing quite simple.

```
1  Wildcard:
2      Identifier ":" MultipleMatch
3
4  MultipleMatch:
5      GroupExpr "*"
6      | TypeExpr
7
8  TypeExpr:
9      BinaryExpr
10     | UnaryExpr
11     | PrimitiveExpr
12
13 BinaryExpr:
14     TypeExpr { Operator TypeExpr }*
15
16 UnaryExpr:
17     {UnaryOperator}? TypeExpr
18
```

```
19 PrimitiveExpr:
20     GroupExpr | Identifier
21
22 GroupExpr:
23     "(" TypeExpr ")"
```

Listing 4.3: Grammar of type expressions

The grammar of the type expressions used by the wildcards can be seen in Figure 4.3, the grammar is written in something similar to Extended Backus-Naur form, where we define the terminals and non-terminals in a way that makes the entire grammar parseable by the recursive descent parser.

Our recursive descent parser produces an AST, which is later used to determine when a wildcard can be matched against a specific AST node, the full definition of this AST can be seen in Appendix A.1. We use this AST by traversing it using a [37]visitor pattern and comparing each `Identifier` against the specific AST node we are currently comparing with, and evaluating all subsequent expressions and producing a boolean value, if this value is true, the node is matched against the wildcard, if not then we do not have a match.

Extracting wildcards from JSTQL-SH   The self-hosted version JSTQL-SH also requires some form of pre-parsing in order to prepare the internal DSL environment. This is relatively minor and only parsing directly with no insertion compared to JSTQL .

In order to use JavaScript as the meta language, we define a `prelude` on the object used to define the transformation case. This prelude is required to consist of several `Variable declaration` statements, where the variable names are used as the internal DSL variables and right side expressions are strings that contain the type expression used to determine a match for that specific wildcard.

We use Babel to generate the AST of the `prelude` definition, this allows us to get a JavaScript object structure. Since the structure is very strictly defined, we can expect every `stmt` of `stmts` to be a variable declaration, otherwise throw an error for invalid prelude. Then the string value of each of the variable declarations is passed to the same parser used for JSTQL wildcards.

The reason this is preferred is it allows us to avoid having to extract the wildcards and inserting an Identifier.

4.4 Using Babel to parse

Allowing the tool to perform transformations of code requires the generation of an Abstract Syntax Tree from the users code, `applicable to` and `transform to`. This means parsing JavaScript into an AST, in order to do this we use Babel [3].

The most important reason for choosing to use Babel for the purpose of generating the AST's used for transformation is due to the JavaScript community surrounding Babel. As this tool is dealing with proposals before they are part of JavaScript, a parser that supports early proposals for JavaScript is required. Babel works closely with TC39 to support experimental syntax [7] through its plugin system, which allows the parsing of code not yet part of the language.

Custom Tree Structure

Performing matching and transformation on each of the sections inside a `case` definition, they have to be parsed into and AST in order to allow the tool to match and transform accordingly, for this we use Babel [3]. However, Babels AST structure does not suit traversing multiple trees at the same time, this is a requirement for matching and transforming. Therefore we take the AST and transform it into a simple custom tree structure to allow for simple traversal of the tree.

As can be seen in Figure 4.4 we use a recursive definition of a `TreeNode` where a nodes parent either exists or is null (it is top of tree), and a node can have any number of children elements. This definition allows for simple traversal both up and down the tree. Which means traversing two trees at the same time can be done in the matcher and transformer section of the tool.

```
export class TreeNode<T> {
    public parent: TreeNode<T> | null;
    public element: T;
    public children: TreeNode<T>[] = [];

    constructor(parent: TreeNode<T> | null, element: T) {
        this.parent = parent;
        this.element = element;
        if (this.parent) this.parent.children.push(this);
    }
}
```

Listing 4.4: Simple definition of a Tree structure in TypeScript

Placing the AST generated by Babel into this structure means utilizing the library [9]Babel Traverse. Babel Traverse uses the visitor pattern [37] to perform traversal of the AST. While this method does not suit traversing multiple trees at the same time, it allows for very simple traversal of the tree to place it into our simple tree structure.

To place the AST into our tree structure, we use `@babel/traverse` [9] to visit each node of the AST in a *depth first* manner, the idea is we implement a *visitor* for each of the nodes in the AST and when a specific node is encountered, the corresponding visitor of that node is used to visit it. When transferring the AST into our simple tree structure we simply have to use the same visitor for every kind of AST node, and place that node into the tree.

Visiting a node using the `enter()` function means we went from the parent to that child node, and it should be added as a child node of the parent. The node is automatically added to its parent list of children nodes from the constructor of `TreeNode`. Whenever leaving a node the function `exit()` is called, this means we are moving back up into the tree, and we have to update what node was the *last* in order to generate the correct tree structure.

```
 1 traverse(ast, {
 2         enter(path: any) {
 3             let node: TreeNode<t.Node> = new TreeNode<t.Node>(
 4                 last,
 5                 path.node as t.Node
 6             );
 7
 8             if (last == null) {
 9                 first = node;
10             }
11             last = node;
12         },
13         exit(path: any) {
14             if (last && last?.element?.type != "Program") {
15                 last = last.parent;
16             }
17         },
18     });
19     if (first != null) {
20         return first;
21     }
```

One important nuance of the way we place the nodes into the tree, is we still have the same underlying data structure from Babel. Because of this, the nodes can still be used with Babels APIs, and we can still access every field of each node. Transforming it into a tree only creates an easy way to traverse up and down the tree by references. We perform no copying.

4.5 Outline of transforming user code

Below is an outline of every major step performed, and how data is passed through the program.

Algorithm 1 Outline of steps of algorithm

1: $CA, CT, W \leftarrow extractWildcards()$
2: $A, T \leftarrow babel.parse(CA, CT)$ ▷ Parse templates
3: $C \leftarrow babel.parse()$ ▷ Parse user code
4: $AT, TT, CT \leftarrow Tree(A, T, C)$ ▷ Build the tree structure from Babel AST
5: **if** $AT.length > 1$ **then**   ▷ Decide which matcher to use
6:     $M \leftarrow multiMatcher(CT, AT, W)$
7: **else**
8:     $M \leftarrow singleMatcher(CT, AT, W)$
9: **end if**
10: $TMap \leftarrow \text{Map}()$
11: **for each** m **in** M **do**   ▷ Build transformation templates
12:     TMap.insert $\leftarrow$ buildTransform(m, TT, W);
13: **end for**
14: **for** $traverse(C)$ **do**
15:     **if** $TMap.has(c)$ **then**
16:         C.replaceMany($TMap.get(c)$);
17:     **end if**
18: **end for**
19: **return** babel.generate(C);

Each part of Algorithm 1 is a step in the full algorithm for transforming user code based on a proposal specification in our tool. The initial step (line 1) is extraction of wildcards from the template definition. This step also parses the wildcard type expressions into an AST. The second step (lines 2,3) is to parse all templates into an AST with `@babel/parser` [6]. Once we have parsed all code into ASTs, we decide which matching algorithm to use (line 5) based on the `applicable to` template. These algorithms will find all matching sections of the user AST to the template. We then build the transformation templates(lines 11-13), and insert the sections from the use code that was matched with a wildcard. These transformations are stored in a `Map`(line 10). Once all transformations are prepared, we traverse the user AST (line 14), and insert the transformations if the current node traversed is in the `Map` (line 16). The final step, is to generate JavaScript from the transformed AST (line 19).

4.6 Matching

This section discusses how we find matches in the users code, this is the step described in lines 5-10 of Listing 1. Firstly, we will discuss how individual nodes are compared, then how the two traversal algorithms are implemented, and how matches are discovered using these algorithms.

4.6.1 Determining if AST nodes match

The initial problem we have to overcome is a way of comparing AST nodes from the template to AST nodes from the user code. This step also has to take into account comparing against wildcards and pass that information back to the AST matching algorithms.

When comparing two AST nodes in this tool, we use the function `checkCodeNode`, which will give the following values based on what kind of match these two nodes produce.

NoMatch The nodes do not match.

Matched The nodes are a match, and the node of `applicable to` is not a wildcard.

MatchedWithWildcard The node of the user AST produced a match against a wildcard.

MatchedWithPlussedWildcard The node of the user AST produced a match against a wildcard that can match one or more nodes against itself.

When we are comparing two AST nodes, we have to perform an equality check. Due to this being a structural matching search, we can get away with just performing some preliminary checks, such as that names of identifiers, otherwise it is sufficient to just perform an equality check of the types of the nodes we are currently trying to match. If the types are the same, they can be validly matched against each other. This is sufficient because we are currently trying to determine if a single node can be a match, and not the entire template structure is a match. Therefore false positives that are not equivalent are highly unlikely due to the fact the entire structure has to be a false positive match.

There is a special case when comparing two nodes, namely when encountering a wildcard. To know if we have encountered a wildcard, the current AST node of `applicable to` will be either an `Identifier` or a `ExpressionStatement` where the expression is an `Identifier`. The reason it might be an `ExpressionStatement` is due to the wildcard

extraction step, where we replace the wildcard with an identifier of the same name. Due to this replacement, we might place an identifier as a statement, the identifier will then be wrapped inside an `ExpressionStatement` AST node. If the node of `applicable to` is of either of these types, we have to check if the name of the identifier is the same as a wildcard. If it is, we have to compare the type of the user AST node against the type expression of the wildcard.

```
if((aplToNode.type === "ExpressionStatement" &&
    aplToNode.expression.type === "Identifier") ||
    aplToNode.type === "Identifier"){

    // Check if aplToNode is a wildcard
}
```

When comparing an AST node type against a wildcard type expression, we pass the node type into a function `WildcardEvaluator`. This evaluator will traverse through the AST of the wildcard type expression. Every leaf of the tree is equality checked against the type, and the resulting boolean value is returned. Then we evaluate the expression, passing the values through the visitors until we have evaluated the entire expression, and have a result. If the result of the evaluator is `false`, we return `NoMatch`. If the result of the evaluation is `true`, we know we can match the user's AST node against the wildcard. If the wildcard type expression contains a Kleene plus, the comparison returns `MatchedWithPlussedWildcard`, if not, we return `MatchedWithWildcard`.

4.6.2 Matching a single Expression/Statement template

In this section, we will discuss how matching is performed when the `applicable to` template is a single expression/statement. A very complex matching template with many statements might result in a lower chance of finding matches in the users code. Therefore using simple, single root node matching templates provide the highest possibility of discovering a match within the users code. This section will cover line 11 of Listing 1.

Determining if we are currently matching with a template that is only a single expression/statement, we have to verify that the program body of the template has the length of one, if it does we can use the single length traversal algorithm.

There is a special case for if the template is a single expression, as the first node of the AST generated by `@babel/generate` [5] will be of type `ExpressionStatement`, the reason for this is Babel will treat free floating expressions as a statement. This will miss many applicable parts of the users code, because expressions within other statements are

not wrapped in an `ExpressionStatement`. This will give a template that is incompatible with a lot of otherwise applicable expressions. This means the statement has to be removed, and the search has to be done with the expression as the top node of the template. If the node in the body of the template is a statement, no removal has to be done, as a statement can be used directly.

Discovering Matches Recursively The matcher used against single expression/statement templates is based Depth-First Search to traverse the trees. The algorithm can be split into two steps. The initial step is to check if we are currently at the root of the `applicable to` AST, the second is to try to match the current nodes, and start a search on each of their child nodes.

It is important we try to match against the template at all levels of the code AST, this is done by starting a new search one every child node of the code AST if the current node of the template AST is the root node. This ensures we have tried to perform a match at any level of the tree. This also ensures we have no partial matches, as we store it only if it returns a match when being called with the root node of `applicable to`.

```
1  if(aplTo.element === this.aplToRoot){
2      // Start a search from root of aplTo on all child nodes
3      for(let codeChild of code.children){
4          let childMatch = singleMatcher(codeChild, aplTo);
5
6          // If it is a match, we know it is a full match and store it.
7          if(childMatch){
8              this.matches.push(childMatch);
9          }
10     }
11 }
```

We can now determine if we are currently exploring a match. This means the current code AST node is checked against the current node of `applicable to` AST. Based on what kind of result the comparison between these two nodes give, we have perform different steps.

NoMatch: If a comparison between the nodes return a `NoMatch` result, we perform an early return of undefined, as no match was discovered. We can safely discard this search, because we have started a search at all levels of the code AST.

Matched: The current code node matches against the current node of the template, and we have to perform a search on each of the child nodes.

41

MatchedWithWildcard: When a comparison results in a wildcard match, we pair the current code node and the template wildcard, and do an early return. We can do this because if a wildcard matches, the nodes of the children does not matter and will be placed into the transformation.

MatchedWithPlussedWildcard: this is a special case for a wildcard match. When a match against a wildcard that has the Kleene plus tied to it we also perform an early return. This result means some special traversal has to be done to the current nodes siblings, this is described below.

A comparison result of `Matched` means the two nodes match, but the `applicable to` node is not a wildcard. With this case, we perform a search on each child nodes of `applicable to` AST and the user AST. This is performed in order, meaning the n-th child node of `applicable to` is checked against the n-th child node of the user AST.

When checking the child nodes, we have to check for a special case when the comparison of the child nodes result in `MatchedWithPlussedWildcard`. If this result is encountered, we have to continue matching the same `applicable to` node against each subsequent sibling node of the code node. This is because, a wildcard with a Keene plus can match against multiple sibling nodes. This behavior can bee seen in line 17-31 of Listing 4.5.

If all child nodes did not give the result of NoMatch, we have successfully matched every node of the `applicable to` AST. This does not yet mean we have a match, as there might be remaining nodes in the child node of the code AST. To check for this, we check whether or not `codeI` is equal to the length of `code.children`. If it is unequal, we have not matched all child nodes of the code AST and have to return `NoMatch`. This can be seen on lines 37-39 of Listing 4.5.

```
1  let codeI = 0;
2  let aplToI = 0;
3
4  while (aplToI < aplTo.children.length && codeI < code.children.length){
5      let [pairedChild, childResult] =
          ↪ singleMatcher(code.children[codeI], aplTo.children[aplToI]);
6
7      // If a child does not match, the entire match is discarded
8      if(childResult === NoMatch){
9          return [undefined, NoMatch];
10     }
11
12     // Add the match to the current Paired Tree structure
13     pairedChild.parent = currentPair;
14     currentPair.children.push(pairedChild);
15
16     // Special case for Keene plus wildcard match
17     if(childResult === MatchedWithPlussedWildcard){
18         codeI += 1;
```

```
19        while(codeI < code.children.length){
20            let [nextChild, plusChildResult] =
                ↪ singleMatcher(code.children[codeI],
                ↪ aplTo.children[aplToI]);
21
22            if(plusChildResult !== MatchedWithPlussedWildcard){
23                i -= 1;
24                break;
25            }
26
27            pairedChild.element.codeNode.push(...nextChild.element.codeNode);
28
29            codeI += 1;
30        }
31    }
32
33    codeI += 1;
34    aplToi += 1;
35 }
36
37 if(codeI !== code.children.length){
38    return [undefined, NoMatch]
39 }
40
41 return [currentPair, Match];
```

Listing 4.5: Pseudocode of child node matching

4.6.3 Matching multiple Statements

Using multiple statements in the template of `applicable to` means the tree of `applicable to` as multiple root nodes, to perform a match with this kind of template, we use a sliding window [31] with size equal to the amount statements in the template. This window is applied at every *BlockStatement* and `Program` of the code AST, as that is the only placement statements can reside in JavaScript [26, 14].

The initial step of this algorithm is to search through the AST for ast nodes that contain a list of *Statements*. Searching the tree is done by Depth-First search, at every level of the AST, we check the type of the node. Once a node of type `BlockStatement` or `Program` is discovered, we start the trying to match the statements.

```
1 multiStatementMatcher(code, aplTo) {
2    if (
3        code.element.type === "Program" ||
4        code.element.type === "BlockStatement"
5    ) {
6        matchMultiHead(code.children, aplTo.children);
7    }
8
9    for (let code_child of code.children) {
10        multiStatementMatcher(code_child, aplTo);
11    }
12 }
```

`matchMultiHead` uses a sliding window [31]. The sliding window will try to match every statement of the code AST against its corresponding statement in the `applicable to` AST. For every statement, we perform a DFS recursion algorithm is applied, similar to algorithm used in Section 4.6.2, however this search is not applied to all levels, and if it matches it has to match fully and immediately. If a match is not found, the current iteration of the sliding window is discarded and we move on to the next iteration by moving the window one further.

One important case here is we might not know the width of the sliding window, this is due to wildcards using the Keene plus, as they can match one or more nodes against the wildcard. These wildcards might match against `(Statement)+`. Therefore, we use a similar technique to the one described in Section 4.6.2, where we have two pointers and match each statement depending on each pointer.

Output of the matcher

The matches discovered have to be stored such that we can easily find all the nodes that were matched against wildcards and transfer them into the transformation later. To make this simpler, we make use an object `PairedNodes`. This object allows us to easily find exactly what nodes were matched against each other. The matcher will place this object into the same tree structure described in 4.4. This means the result of running the matcher on the user code is a list of `TreeNode<PairedNode>`.

```
1 interface PairedNode{
2     codeNode: t.Node[],
3     aplToNode: t.Node
4 }
```

Since a match might be multiple statements, we use an interface `Match`, that contains separate tree structures of `PairedNodes`. This allows storage of a match with multiple root nodes.

```
1 export interface Match {
2     // Every matching Statement in order with each pair
3     statements: TreeNode<PairedNodes>[];
4 }
```

4.7 Transforming

To perform the transformation and replacement on each of the matches, we take the resulting list of matches, the template from the `transform to` section of the current case of the proposal, and the Babel AST [4] version of original code. All the transformations are then applied to the code and we use `@babel/generate` [5] to generate JavaScript code from the transformed AST.

An important discovery is to ensure we transform the leafs of the AST first, this is because if the transformation was applied from top to bottom, it might remove transformations done using a previous match. This means if we transform from top to bottom on the tree, we might end up with `a(b) |> c(%)` in stead of `b |> a(%) |> c(%)` in the case of the pipeline proposal. This is quite easily solved in our case, as the matcher looks for matches from the top of the tree to the bottom of the tree, the matches it discovers are always in that order. Therefore when transforming, all that has to be done is reverse the list of matches, to get the ones closest to the leaves of the tree first.

Building the transformation

Before we can start to insert the `transform to` section into the user's code AST. We have to insert all nodes matched against a wildcard in `applicable to` into their reference locations.

The first step to achieve this is to extract the wildcards from the match tree. This is done by recursively searching the match tree for an `Identifier` or `ExpressionStatement` containing an `Identifier`. To do this, we have a function `extractWildcardPairs`, which takes a single match, and extracts all wildcards and places them into a `Map<string, t.Node[]>`. Where the key of the map is the identifier used for the wildcard, and the value is the AST nodes the wildcard was matched against in the users code.

```
1  function extractWildcardPairs(match: Match): Map<string, t.Node[]> {
2      let map: Map<string, t.Node[]> = new Map();
3
4      function recursiveSearch(node: TreeNode<PairedNodes>) {
5          let name: null | string = null;
6          if (node.element.aplToNode.type === "Identifier") {
7              name = node.element.aplToNode.name;
8          } else if (
9              // Node is ExpressionStatement with Identifier
10         ) {
11             name = node.element.aplToNode.expression.name;
12         }
13
```

```
14          if (name) {
15              // Store in the map
16              map.set(name, node.element.codeNode);
17          }
18          // Recursively search the child nodes
19          for (let child of node.children) {
20              recursiveSearch(child);
21          }
22      }
23      // Start the initial search
24      for (let stmt of match.statements) {
25          recursiveSearch(stmt);
26      }
27      return map;
28 }
```

Listing 4.6: Extracting wildcard from match

Once the full map of all wildcards has been built, we have to insert the wildcards into the Babel AST of the `transform to` template. To do this, we have to traverse the template and insert the matched nodes of the user's code. We use `@babel/traverse` [9] to traverse the AST, as this provides us with a powerful API for modifying the AST. `@babel/traverse` allows us to define visitors, that are executed when traversing specific types of AST nodes. For this, we define a visitor for `Identifier`, and a visitor for `ExpressionStatement`. These visitors will do exactly the same, however for the `ExpressionStatement`, we have to check if the expression is an identifier.

When we visit a node that might be a wildcard, we check if that nodes name is in the map of wildcards built in Listing 4.6. If the name of the identifier is a key in the wildcard, we get the value for that key, and perform a node replacement. Where we replace the identifier with the node from the user's code that was matched against that wildcard. See Listing 4.7

```
1  traverse(transformTo, {
2      Identifier: (path) => {
3          if (wildcardMatches.has(path.node.name)) {
4              let toReplaceWith =
                   ↪ wildcardMatches.get(path.node.name);
5              if (toReplaceWith) {
6                  path.replaceWithMultiple(toReplaceWith);
7              }
8          }
9      },
10      ExpressionStatement: (path) => {
11          if (path.node.expression.type === "Identifier") {
12              let name = path.node.expression.name;
13              if (wildcardMatches.has(name)) {
14                  let toReplaceWith = wildcardMatches.get(name);
15                  if (toReplaceWith) {
16                      path.replaceWithMultiple(toReplaceWith);
17                  }
18              }
19          }
20      },
21  });
```

Listing 4.7: Traversing `transform to` AST and inserting user context

Due to some wildcards allowing matching of multiple sibling nodes, we have to use `replaceWithMultiple` when performing the replacement. This can be seen on line 6 and 16 of Listing 4.7.

Inserting the template into the AST

We have now created the `transform to` template with the user's context. This has to be inserted into the full AST definition of the users code. To do this we have to locate exactly where in the user AST this match originated. We can perform an equality check on the top noe of the user node stored in the match. To do this efficiently, we perform this check by using this top node as the key to a `Map`, so if a node in the user AST exists in that map, we know it was matched.

```
1 transformedTransformTo.set(
2     match.statements[0].element.codeNode[0],
3     [
4         transformMatchFaster(wildcardMatches, traToWithWildcards),
5         match,
6     ]
7 );
```

To traverse the user AST, we use `@babel/traverse` [9]. In this case we cannot use a specific visitor, and therefore we use a generic visitor that applies to every node of the AST. If the current node we are visiting is a key to the map of transformations, we know we have to insert the transformed code. This is done similarly to before where we use `replaceWithMultiple`.

Some matches have multiple root nodes. This is likely when matching was done with multiple statements as top nodes. This means we have to remove n-1 following sibling nodes. Removal of these sibling nodes can be seen on lines 12-15 of Listing 4.8.

```
1 traverse(codeAST, {
2     enter(path) {
3         if (transformedTransformTo.has(path.node)) {
4             let [traToWithWildcards, match] =
5                 transformedTransformTo.get(path.node) as [
6                     t.File,
7                     Match
8                 ];
9             path.replaceWithMultiple(
10                 traToWithWildcards.program.body);
11
12             let siblings = path.getAllNextSiblings();
13
14             // For multi line applicable to
15             for (let i = 0; i < match.statements.length - 1; i++) {
16                 siblings[i].remove();
17             }
18
```

```
19              // When we have matched top statements with +, we
                ↪ might have to remove more siblings
20              for (let matchStmt of match.statements) {
21                for (let codeStmt of matchStmt.element
22                  .codeNode) {
23                  let siblingnodes = siblings.map((a) => a.node);
24                  if (siblingnodes.includes(codeStmt)) {
25                    let index = siblingnodes.indexOf(codeStmt);
26                    siblings[index].remove();
27                  }
28                }
29              }
30            }
31          },
32        });
```

Listing 4.8: Inserting transformed matches into user code

There is a special case when a wildcard with a Keene plus, allowing the match of
multiple siblings, means we might have more siblings to remove. In this case, it is not so
simple to know exactly how many we have to remove. Therefore, we have to iterate over
all statements of the match, and check if that statement is still a sibling of the current
one being replace. This behavior can be seen on lines 20-29 of Listing 4.8.

After one full traversal of the user AST. All matches found have been replaced with
their respective transformation. All that remains is generating JavaScript from the trans-
formed AST.

Generating source code from transformed AST

To generate JavaScript from the transformed AST created by this tool, we use a
JavaScript library titled [5]babel/generator. This library is specifically designed for use
with Babel to generate JavaScript from a Babel AST. The transformed AST definition of
the users code is transformed, while being careful to apply all Babel plugins the current
proposal might require.

Chapter 5

Evaluation

In this chapter we will discuss how we evaluated JSTQL and its related tools. This chapter will include some testing of the tool on demo code snippets, as well as running each of the proposals discussed in this thesis on some large scale JavaScript projects.

5.1   Real Life source code

In order to perform actual large scale trial of this program, we have collected some github projects containing many or large JavaScript files. Every JS file within the project is then passed through the entire tool, and we will evaluate it based upon the amount of matches discovered, as well as manual checking that the transformation resulted in correct code on the matches.

Each case study was evaluated by running this tool on every .js file in the repository, then collecting the number of matches found in total and how many files were successfully searched. Evaluating if the transformation was correct is done by manually sampling output files, and verifying that it passes through Babel Generate [5] without error.

"Pipeline" [15] is very applicable to most files, as the concept it touches (fucntion calls) is widely used all across JavaScript. This is by far the best result, and it found matches in almost all files that Babel [6] managed to parse.

The Do proposal [17] is expected to not find as many matches, as code that has not been written in expression-oriented programming style will not produce many matches.

However, this also highlights how impactful this proposal is to previously written code compared to "Pipeline".

Await to promise also has an expected number of matches, but this evaluation proposal is not meant to be real life representative. As it is limited to functions containing only a single await statement and that statement has to be a `VariableDeclaration`.

Next.js   [14] is one of the largest projects on the web. It is used with React [18] to enable feature such as server-sire rendering and static site generation.

Proposal	Matches found	Files with matches	Files processed
"Pipeline"	242079	1912	3340
"Do" expression	480	111	3340
Await to Promise	143	75	3340

Figure 5.1: Evaluation with Next.js source code

Three.js   [25] is a library for 3D rendering in JavaScript. It is written purely in JavaScript and uses GPU for 3D calculations. It being a popular JavaScript library, and being written in mostly pure JavaScript makes it a good case study for our tool. It currently sits at over 1 million downloads weekly.

Proposal	Matches found	Files with matches	Files searched
Pipeline	84803	1117	1384
"Do" expression	277	55	1384
Await to Promise	13	7	1384

Figure 5.2: Evaluation with Three.js source code

React   [18] is a graphical user interface library for JavaScript, it facilitates the creation of user interfaces for both web and native platforms. React is based upon splitting a user interface into components for simple development. It is currently one of the most popular libraries for creating web apps and has over 223000 stars on Github.

Proposal	Matches found	Files with matches	Files searched
"Pipeline"	16353	1266	2051
"Do" expression	79	60	2051
Await to Promise	30	13	2051

Figure 5.3: Evaluation with React source code

Bootstrap [10] is a front-end framework used for creating responsive and mobile-first websites, it comes with a variety of built-in components, as well as a built in styling. This styling is also customizable using CSS. This library is a good evaluation point for this thesis as it is written in pure JavaScript and is used by millions of developers.

Proposal	Matches found	Files with matches	Files searched
""Pipeline"	13794	109	115
"Do" expression	13	8	115
Await to Promise	0	0	115

Figure 5.4: Evaluation with Bootstrap source code

Atom [2] is a text editor made in JavaScript using the Electron framework. It was created to give a very minimal and modular text editor. It was bought by Microsoft, and later discontinued in favor for Visual Studio Code.

Proposal	Matches found	Files with matches	Files searched
"Pipeline"	40606	361	401
"Do" expression	46	26	401
Await to Promise	12	7	401

Figure 5.5: Evaluation with Atom source code

5.1.1   Example transformations

Highlights of transformations from evaluation

```
tracks.push( parseKeyframeTrack( jsonTracks[ i ] ).scale( frameTime )
    ↪ ) );
```

```
frameTime
|> (jsonTracks[i] |> parseKeyframeTrack(%)).scale(%)
|> tracks.push(%);
```

Transformation taken from `three.js/src/animation/AnimationClip.js`

```
const logger = createLogger({
  storagePath: join(__dirname, '.progress-estimator'),
});
```

```
const logger = {
  storagePath: __dirname |> join(%, '.progress-estimator')
} |> createLogger(%);
```

"Pipeline" transformation, taken from `react/scripts/devtools/utils.js`

```
if (isElement(content)) {
this._putElementInTemplate(getElement(content), templateElement)
return
}
```

```
if (content |> isElement(%)) {
    content |> getElement(%) |> this._putElementInTemplate(%,
        ↪ templateElement);
    return;
}
```

"Pipeline" transformation, taken from `bootstrap/js/src/util/template-factory.js`

```
for (const file of typeFiles) {
const content = await fs.readFile(join(styledJsxPath, file), 'utf8')
await fs.writeFile(join(typesDir, file), content)
}
```

```
for (const file of typeFiles) {
const content = await (styledJsxPath |> join(%, file) |>
    ↪ fs.readFile(%, 'utf8'));
await (typesDir |> join(%, file) |> fs.writeFile(%, content));
}
```

"Pipeline" transformation, taken from `next.js/packages/next/taskfile.js`

```
1 if (repo && repo.onDidDestroy) {
2     repo.onDidDestroy(() =>
3     this.repositoryPromisesByPath.delete(pathForDirectory)
4     );
5 }
```

```
1 if (repo && repo.onDidDestroy) {
2     (() => pathForDirectory |>
          ↪ this.repositoryPromisesByPath.delete(%)) |>
          ↪ repo.onDidDestroy(%);
3 }
```

"Pipeline" transformation, taken from `atom/src/project.js`

```
1 await check(async () => {
2 const html = await browser.eval('document.documentElement.innerHTML')
3 return html.match(/iframe/) ? 'fail' : 'success'
4 }, /success/)
```

```
1 await check(do {
2 const html = await browser.eval('document.documentElement.innerHTML');
3 html.match(/iframe/) ? 'fail' : 'success'
4 }, /success/);
```

"Do expression" transformation, taken from `next.js/test/integration/typescript-hmr/index.tes`

```
1  async function getCurrentRules() {
2    const res = await fetch(
3      'https://api.github.com/repos/vercel/next.js/branches/canary/protection',
4      {
5        headers: {
6          Accept: 'application/vnd.github+json',
7          Authorization: 'Bearer ${authToken}',
8          'X-GitHub-Api-Version': '2022-11-28',
9        },
10     }
11   )
12
13   if (!res.ok) {
14     throw new Error(
15       'Failed to check for rule ${res.status} ${await res.text()}'
16     )
17   }
18   const data = await res.json()
19
20   return {
21     // Massive JS Object
22   }
23 }
```

```
1  async function getCurrentRules() {
2    return
         ↪ fetch('https://api.github.com/repos/vercel/next.js/branches/canary/protectio
         ↪ {
3      headers: {
4        Accept: 'application/vnd.github+json',
5        Authorization: 'Bearer ${authToken}',
6        'X-GitHub-Api-Version': '2022-11-28'
7      }
8    }).then(async res => {
9      if (!res.ok) {
10       throw new Error('Failed to check for rule ${res.status} ${await
           ↪ res.text()}');
11     }
12     const data = await res.json();
13     return {
14         // Massive JS object
15     };
16   });
17 }
```

"Await to Promise" transformation, taken from `next.js/test/integration/typescript-hmr/index.`

As can be seen in the results, ... Some proposals more impactful than others

Chapter 6

Related Work

In this chapter, we present work related to other query languages for source code, aspect-oriented programming, some code querying methods, and other JavaScript parsers. This all relates to the work described in this thesis.

6.1 Aspect-Oriented Programming

AoP, is a programming paradigm that gives increased modularity by allowing for a high degree of separation of concerns, specifically focusing on cross-cutting concerns.

Cross-cutting concerns are aspects of a software program or system that have an effect at multiple levels, cutting across the main functional requirements. Such aspects are often related to security, logging, or error handling, but could be any concern that are shared across an application.

In AOP, one creates an *aspect*, which is a module that contains some cross-cutting concern the developer wants to achieve, this can be logging, error handling or other concerns not related to the original classes it should applied to. An aspect contains advices,which is the specific code executed when certain conditions of the program are met, an example of these are *before advice*, which is executed before a method executes, *after advice*, which is executed after a method regardless of the methods outcome, and *around advice*, which surrounds a method execution. Contained within the aspect is also a *pointcut*, which is the set of criteria determining when the aspect is meant to be executed. This can be at specific methods, or when specific constructors are called etc.

Aspect oriented programming is similar to this project in that to define where *pointcuts* are placed, we have to define some structure and the AOP library has to search the code execution for events triggering the pointcut and run the advice defined within the aspect of that given pointcut. Essentially performing a rewrite of the code during execution to add functionality to multiple places in the executing code.

6.2   Other source code query languages

To allow for simple analysis and refactoring of code, there exists many query languages designed to query source code. These languages use several methods to allow for querying code based on specific paradigms such as logical queries, declarative queries, or SQL-like queries. All provide similar functionality of being able to query code. In this section we will look some of these languages for querying source code, and how they relate to JSTQL developed in this thesis.

6.2.1   CodeQL

CodeQL [11] is an object-oriented query language, it was previously known as .QL. . CodeQL is used to analyze code semantically to discover vulnerabilities [34]. CodeQL has taking inspiration from several areas of computer science to create their query language [34], such a inspiration from SQL, Datalog, Eindhoven QUantifier Notation, and Classes are Predicates.

An example of how queries are written in CodeQL can be defined below. This query will find all methods that declare a method `getNumber` and a method `setNumber`.

```
from Class c
where c.declaresMethod("getNumber") and
    (c.declaresMethod("setNumber")) and
    c.fromSource()
select c.getPackage(), c
```

The syntax of writing queries in CodeQL is not similar to JSTQL , as it is SQL-like, and not declarative patterns, which makes the writing experience of the two languages very different. Writing CodeQL queries are similar to querying a database, while queries written in JSTQL are similar to defining an example of the structure you wish to search for.

6.2.2   PMD XPath

PMD XPath is a language for Java source code querying, it supports querying of all Java constructs. The reason it has this wide support is due to it constructing the entire codebase's AST in XML format, and then performing the query on the XML. This makes the query language very versatile for static code analysis, and is used in the PMD static code analysis tool.

```
1 public class a{
2     int someVar;
3     private void run(){
4         int otherVar;
5     }
6 }
```

There are two queries with PMD XPath defined in the example below. If we execute these on the code above, the first will return both `someVar` and `otherVar`, while the second will only return `otherVar`. The reason the second limits the search, is we define that the parent of the variable declaration has to be a method. This is done by moving three levels up in the XML and checking the name of the node residing there.

```
1 //LocalVariableDeclaration
2 //LocalVariableDeclaration[name(../../..) = "methodDeclaration"]
```

Comparing this tool to JSTQL , we can see it is good at querying code based on structure, which our tool also supports. The main difference is the manor of which each tool does this, JSTQL uses JavaScript templates to perform the query, making writing queries simple for users as they are based in JavaScript. PMD XPath uses a custom query language to perform structural queries that is quite verbose, and requires extended knowledge of the AST that is currently being queried.

6.2.3   XSL Transformations

6.2.4   Jackpot

Jackpot [36] is a query language created for the Apache Netbeans platform [35], it has since been mostly renamed to Java Declarative Hints Language, we will continue to refer to it as Jackpot in this section. The language uses declarative patterns to define source code queries, these queries are used in conjunction with multiple rewrite definitions. This is used in the Apache Netbeans suite of tools to allow for declarative refactoring of code.

This is quite similar to the form of JSTQL , as both language define som query by using similar structure, in Jackpot you define a *pattern*, then every match of that pattern can be re-written to a *fix-pattern*, each fix-pattern can have a condition attached to it. This is quite similar to the *applicable to* and *transform to* sections of JSTQL . Jackpot also supports something similar to the wildcards in JSTQL , as you can define variables in the *pattern* definition and transfer them over to the *fix-pattern* definition. This is closely related to the definition of wildcards in JSTQL , though without type restrictions and notation for matching more than one AST node.

The example of a query and transformation below, will query the code for variable declarations with initial value of 1, and then change them into a declaration with initial value of 0.

```
1 "change declarations of 1 to declarations of 0":
2     int $1 = 1;
3 =>  int $1 = 0
```

6.3   JetBrains structural search

JetBrains integrated development environments have a feature that allows for structural search and replace [20]. This feature is intended for large code bases where a developer wants to perform a search and replace based on syntax and semantics, not just a regular text based search and replace. A search is applied to specific files of the codebase or the entire codebase. It does not recursively check the entire static structure of the code, but this can be specified in the user interface of structural search and replace.

When doing structural search in Jetbrains IntelliJ IDEA, templates are used to describe the query used in the search. These templates use variables described with `$variable$`, these allow for transferring context to the structural replace.

This tool is an interactive experience, where each match is showcased in the find tool, and the developer can decide which matches to apply the replace template to. This allows for error avoidance and a stricter search that is verified by humans. If the developer wants, they do not have to verify each match and just replace everything.

When comparing this tool to JSTQL and its corresponding program, there are some similarities. They are both template based, which means a search uses a template to define query, both templates contain variables/wildcards in order to match against a free

section, and the replacing structure is also a template based upon those same variables. A way of matching the variables/wildcards of structural search and replace also exists, one can define the amount of X node to match against, similar to the + operator used in JSTQL . A core difference between JSTQL and structural search and replace is the variable type system. When performing a match and transformation in JSTQL the types are used extensively to limit the match against the wildcards, while this limitation is not possible in structural search and replace.

6.4   Other JavaScript parsers

This section will explore other JavaScript parsers that could have been used in this project. We will give a brief introduction of each of them, and discuss why they were not chosen.

Speedy Web Compiler

Speedy Web Compiler [19] is a library created for parsing JavaScript and other dialects like JSX, TypeScript faster. It is written in Rust and advertises faster speeds than Babel and is used by large organizations creating applications and tooling for the web platform.

Similar to Babel [3], Speedy Web Compiler is an extensible parser that allows for changing the specification of the parsed program. Its extensions are written in Rust. While it does not have as mature of a plugin system as Babel, its focus on speed makes it widely used for large scale web projects.

Speedy Web Compiler supports features out of the box such as Compilation, used for TypeScript and other languages that are compiled down to JavaScript. Bundling, which takes multiple JavaScript/TypeScript files and bundles them into a single output file, while handling naming collisions. Minification, to make the bundle size of a project smaller, transforming for use with WebAssembly, and custom plugins to change the specification of the languages parsed by SWC.

Compared to Babel used in this paper, SWC focuses on speed, as its main selling point is a faster way of developing web projects.

Acorn

Acorn [1] is parser written in JavaScript to parse JavaScript and it's related languages. Acorn focuses on plugin support in order to support extending and redefinition on how it's internal parser works. and has it's own tree traversal library Acorn Walk. Babel is originally a fork of Acorn, while Babel has since had a full rewrite. Acorn focuses heavily on supporting third party plugins, which Babel does not. However Acorn was not a good fit for this project, as Acorn only supports Stage 4 proposals, and support for proposals in the early stages is a requirement.

6.5 Model-to-Model transformations

Chapter 7

Future Work

Provide access and gather feedback. This project is build upon creating a tool for users of EcmaScript to see new proposals within their own codebase. The idea behind this is to use the users familiarity to showcase new syntactic proposals, and get valuable feedback to the committee developing the ECMA-262 standard. This means making the definitions of a proposal in JSTQL and this tool available to end-users to execute using their own code. This can come in multiple forms, we suggest some ideas, such as a playground on the web, an extension for Visual Studio Code, or to be used in github pull requests.

Supporting other languages. The idea of showcasing changes to a programming language by transforming user code is not only limited to EcmaScript, and could be applied to many other programming languages using a similar development method to EcmaScript. The developers of a language could write definitions of new changes for their respective language, and use a similar tool to the one discussed in this thesis to showcase possible new changes.

Parameterized specifications. The current form of JSTQL supports writing each template as its own respective case, but multiple templates might be very similar and could be written using generics that are shared between case definitions. Introducing this might give a simpler way of writing more complex definitions of a proposal transformation by re-using generic type parameters for the wildcards used in the transformations.

Fully self-hosting JSTQL-SH . The current version of JSTQL-SH relies on this tools parser to generate the AST for the type expressions used for matching by wildcards. This might make this tool more difficult to adopt for the committee. Therefore adding

functionality for writing these type expressions purely in JavaScript and allowing for the use of JavaScript as its own meta language is an interesting avenue to explore.

Support for custom proposal syntax. Currently this tool relies heavily on that a proposal is supported by [3]Babel. This makes the tool quite limited in what proposals could be defined and transformed due to relying on Babel for parsing the templates and generating the output code. Introducing some way of defining new syntax for a proposal in the proposal definition, and allowing for parsing JavaScript containing that specific new syntax would limit the reliance on Babel, and allow for defining proposals earlier in the development process. This can possibly be done by implementing a custom parser inside this tool that allows defining custom syntax for specific new proposals.

Bibliography

[1] acorn, May 2024.
URL: https://github.com/acornjs/acorn. [Online; accessed 21. May 2024].

[2] atom, May 2024.
URL: https://github.com/atom/atom. [Online; accessed 23. May 2024].

[3] Babel · Babel, May 2024.
URL: https://babeljs.io. [Online; accessed 10. May 2024].

[4] babel/packages/babel-parser/ast/spec.md at main · babel/babel, May 2024.
URL: https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md.
[Online; accessed 28. May 2024].

[5] @babel/generator · Babel, May 2024.
URL: https://babeljs.io/docs/babel-generator. [Online; accessed 12. May 2024].

[6] @babel/parser · Babel, May 2024.
URL: https://babeljs.io/docs/babel-Parser. [Online; accessed 14. May 2024].

[7] proposals, May 2024.
URL: https://github.com/babel/proposals. [Online; accessed 27. May 2024].

[8] What is Babel? · Babel, May 2024.
URL: https://babeljs.io/docs/#spec-compliant. [Online; accessed 29. May 2024].

[9] @babel/traverse · Babel, May 2024.
URL: https://babeljs.io/docs/babel-traverse. [Online; accessed 12. May 2024].

[10] bootstrap, May 2024.
URL: https://github.com/twbs/bootstrap. [Online; accessed 23. May 2024].

[11] CodeQL, May 2024.
URL: https://codeql.github.com. [Online; accessed 29. May 2024].

[12] Functions · The Julia Language, May 2024.
URL: `https://docs.julialang.org/en/v1/manual/functions/#Function-composition-and-piping`. [Online; accessed 24. May 2024].

[13] Langium, April 2024.
URL: `https://langium.org`. [Online; accessed 10. May 2024].

[14] next.js, May 2024.
URL: `https://github.com/vercel/next.js`. [Online; accessed 23. May 2024].

[15] proposal-pipeline-operator, May 2024.
URL: `https://github.com/tc39/proposal-pipeline-operator`. [Online; accessed 21. May 2024].

[16] Bikeshedding the Hack topic token · Issue #91 · tc39/proposal-pipeline-operator, May 2024.
URL: `https://github.com/tc39/proposal-pipeline-operator/issues/91`. [Online; accessed 24. May 2024].

[17] proposal-do-expressions, May 2024.
URL: `https://github.com/tc39/proposal-do-expressions`. [Online; accessed 2. May 2024].

[18] react, May 2024.
URL: `https://github.com/facebook/react`. [Online; accessed 23. May 2024].

[19] Rust-based platform for the Web – SWC, May 2024.
URL: `https://swc.rs`. [Online; accessed 21. May 2024].

[20] Structural search and replace | IntelliJ IDEA, April 2024.
URL: `https://www.jetbrains.com/help/idea/structural-search-and-replace.html`. [Online; accessed 22. May 2024].

[21] TC39 - Specifying JavaScript., May 2024.
URL: `https://tc39.es`. [Online; accessed 26. May 2024].

[22] The TC39 Process, April 2024.
URL: `https://tc39.es/process-document`. [Online; accessed 24. May 2024].

[23] how-we-work/implement.md at main · tc39/how-we-work, May 2024.
URL: `https://github.com/tc39/how-we-work/blob/main/implement.md#transpiler-implementations`. [Online; accessed 29. May 2024].

[24] TOML: Tom's Obvious Minimal Language, May 2024.
URL: `https://toml.io/en`. [Online; accessed 27. May 2024].

[25] three.js, May 2024.
URL: `https://github.com/mrdoob/three.js`. [Online; accessed 23. May 2024].

[26] ECMAScript® 2025 Language Specification, May 2024.
URL: `https://tc39.es/ecma262`. [Online; accessed 28. May 2024].

[27] Boyko B. Bantchev. Putting more meaning in expressions. *SIGPLAN Not.*, 33(9): 77–83, September 1998. ISSN 0362-1340. doi: 10.1145/290229.290237.
URL: `https://dl.acm.org/doi/10.1145/290229.290237`.

[28] Matthew S. Davis. An object oriented approach to constructing recursive descent parsers. *SIGPLAN Not.*, 35(2):29–35, feb 2000. ISSN 0362-1340. doi: 10.1145/ 345105.345113.
URL: `https://doi.org/10.1145/345105.345113`.

[29] Sven Efftinge and Miro Spoenemann. Xtext - Language Engineering Made Easy!, February 2024.
URL: `https://eclipse.dev/Xtext`. [Online; accessed 29. May 2024].

[30] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[31] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. Sliding-Window Aggregation Algorithms: Tutorial. In *DEBS '17: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 11–14. Association for Computing Machinery, New York, NY, USA, June 2017. ISBN 978-1-45035065-5. doi: 10.1145/3093742.3095107.
URL: `https://dl.acm.org/doi/abs/10.1145/3093742.3095107`.

[32] KathleenDollard. Symbol and Operator Reference - F#, May 2024.
URL: `https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/symbol-and-operator-reference/#type-symbols-and-operators`. [Online; accessed 24. May 2024].

[33] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, dec 2005. ISSN

0360-0300. doi: 10.1145/1118890.1118892.
URL: https://doi.org/10.1145/1118890.1118892.

[34] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ek-
man, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql
for source code analysis. In *Seventh IEEE International Working Conference on
Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16, 2007. doi:
10.1109/SCAM.2007.31.

[35] Apache NetBeans. Welcome to Apache NetBeans, February 2024.
URL: https://netbeans.apache.org/front/main/index.html. [Online; accessed 21. May
2024].

[36] Apache NetBeans. Java Declarative Hints Language, March 2024.
URL: https://netbeans.apache.org/front/main/jackpot/HintsFileFormat/#variables.
[Online; accessed 21. May 2024].

[37] J. Palsberg and C.B. Jay. The essence of the visitor pattern. In *Proceed-
ings. The Twenty-Second Annual International Computer Software and Applica-
tions Conference (Compsac '98) (Cat. No.98CB 36241)*, pages 9–15, 1998. doi:
10.1109/CMPSAC.1998.716629.

[38] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen,
Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and Zhenyu Chen. Abstract syntax
tree for programming language understanding and representation: How far are we?,
2023.

Appendix A

TypeScript types of wildcard type expressions

```typescript
export interface Identifier extends WildcardNode {
    nodeType: "Identifier";
    name: string;
}

export interface Wildcard {
    nodeType: "Wildcard";
    identifier: Identifier;
    expr: TypeExpr;
    star: boolean;
}

export interface WildcardNode {
    nodeType: "BinaryExpr" | "UnaryExpr" | "GroupExpr" | "Identifier";
}

export type TypeExpr = BinaryExpr | UnaryExpr | PrimitiveExpr;

export type BinaryOperator = "||" | "&&";

export type UnaryOperator = "!";

export interface BinaryExpr extends WildcardNode {
    nodeType: "BinaryExpr";
    left: UnaryExpr | BinaryExpr | PrimitiveExpr;
    op: BinaryOperator;
    right: UnaryExpr | BinaryExpr | PrimitiveExpr;
}
export interface UnaryExpr extends WildcardNode {
    nodeType: "UnaryExpr";
    op: UnaryOperator;
    expr: PrimitiveExpr;
}

export type PrimitiveExpr = GroupExpr | Identifier;

export interface GroupExpr extends WildcardNode {
    nodeType: "GroupExpr";
    expr: TypeExpr;
}
```

Listing A.1: TypesScript types of Type Expression AST