University of Bergen
Department of Informatics

Making a template query language for EcmaScript

Author: Rolf Martin Glomsrud
Supervisor: Mikhail Barash

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`
Monday 27th May, 2024

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Intro goes here

1.1 Background

1.1.1 Figures

Chapter 2

Background

2.1 Proposals

A proposal for EcmaScript is a suggestion for a change to the language. These changes come with a set of problems that if the proposal is included as a part of EcmaScript, those problems should be solved by utilizing the additions contained within the proposal.

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 a imaginary proposal **optional let to int for declaring numerical literal variables**. 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 **optional let to int for declaring numerical literal variables**

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 [17] 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 when used. The order of calls go from right to left, which is 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. On the other hand, nested calls 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.

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

```
// Deeply nested call with multi argument functions
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 when used. In order 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 proves to be a significant downside of chaining, as it only allows for function calls when used, and if one wants to allow for use of other concepts temporary variables have to be used.

```
// Chaining calls
function1().function2().function3();

// Chaining calls with multiple arguments
function1(value1).function2().function3(value2).function4();
```

The Pipeline proposal[17] 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 [6].

The code snippets below showcase the machinery of the proposal.

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

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

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

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

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

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

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

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

The pipe operator is not a new concept, and is present in many other languages such as F# [29] and Julia[13] where the pipe operator is `|`. 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 Expressions

The Do Expressions[16] proposal, is a proposal meant to bring a style of *expression oriented programming*[26] to ECMAScript. Expression oriented programming is a concept taken from functional programming which allows for combining expressions in a very free manor allowing for 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[8] 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 unnamed functions and invoke them immediately, or use an arrow function.

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.

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

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

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

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

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`[9], into code which uses a promise[10].

To perform this transformation, we define an equivalent way of expressing an `await` expression as a promise. The equivalent way of expressing `await` with a promise, is removing `await` from the expression, 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[8] inside the block of that arrow function. This will result in equivalent behavior to using `await`.

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

```
// Re-written using promises
async function a(){
    let b = 9000;
    return asyncFunction()
        .then((something) => {
        let c = something + 100;
        return c;
    })
}
```

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

Proposal Definition JSTQL is designed to mimic the examples already provided in proposal descriptions [23]. 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 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 }
```

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;
    }"
}
```

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

3.3.2 JSTQL

Showcasing a proposal using a user's code requires some way of identifying applicable code sections to that proposal. To do this, we have designed a DSL called JSTQL , JavaScript Template Query Language.

Identifying applicable code

In order 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[2] 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 means an expression can be as simple as `VariableDeclaration`: this will match only against a node of type `VariableDeclaration`. Every type used in these expressions are compared against the AST node types from Babel[2], meaning every AST node type is supported. We also include the types `Statement` for matching against a statement, and `Expression` for matching any expression. The expressions also support binary and unary operators, an example `Statement && !ReturnStatement` will match any statement which is not of type `ReturnStatement`. The expressions support the following operators, `&&` is logical AND, this means both parts of the expression have to evaluate to true, `||` means logical OR, so either side of expression can be true for the entire expression to be true, `!` is the only unary expression, and is logical NOT, so `!Statement` is any node that is NOT a Statement. The wildcards support matching multiple sibling nodes, this is done by using `(expr)+`, this is only valid at the top level of the expression. This is useful for matching against a series of one or more Statements, while not wanting to match an entire `BlockStatement`, this is written as `(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 in an `transform to` block, this template describes the general structure of the newly transformed code.

A transformation template is used to define 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`.

```
// Example applicable to template
applicable to {
    let <<variableName: Identifier>> = <<expr1: Expression>>;
}

// Example of transform to template
transform to {
    const <<variableName>> = <<expr1>>;
}
```

3.3.4 Using JSTQL

JSTQL is designed to be used in tandem with proposal development, this means the users of JSTQL will most likely be contributors to TC39[22] or member of TC39.

JSTQL is designed to closely mimic the style of the examples required in the TC39 process[23]. 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 modern Domain-specific languages have some form of tooling in order 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 missing parameters. The extension performs validation of the wildcards, such as checking for wildcards with missing type parameters,wrong expression definitions, or usage of undeclared wildcards, a demonstration of this can be seen in 3.2.

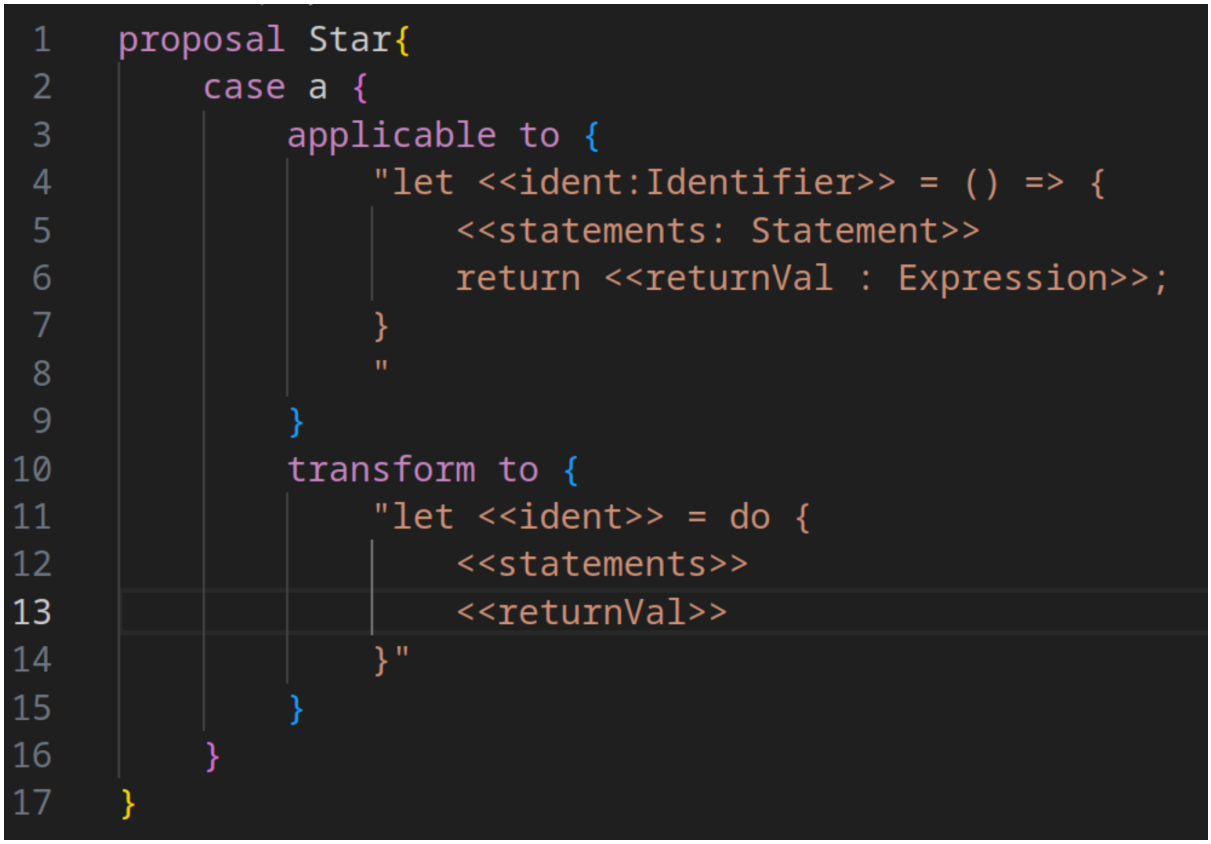

Figure 3.1: Writing JSTQL in Visual Studio Code with extension

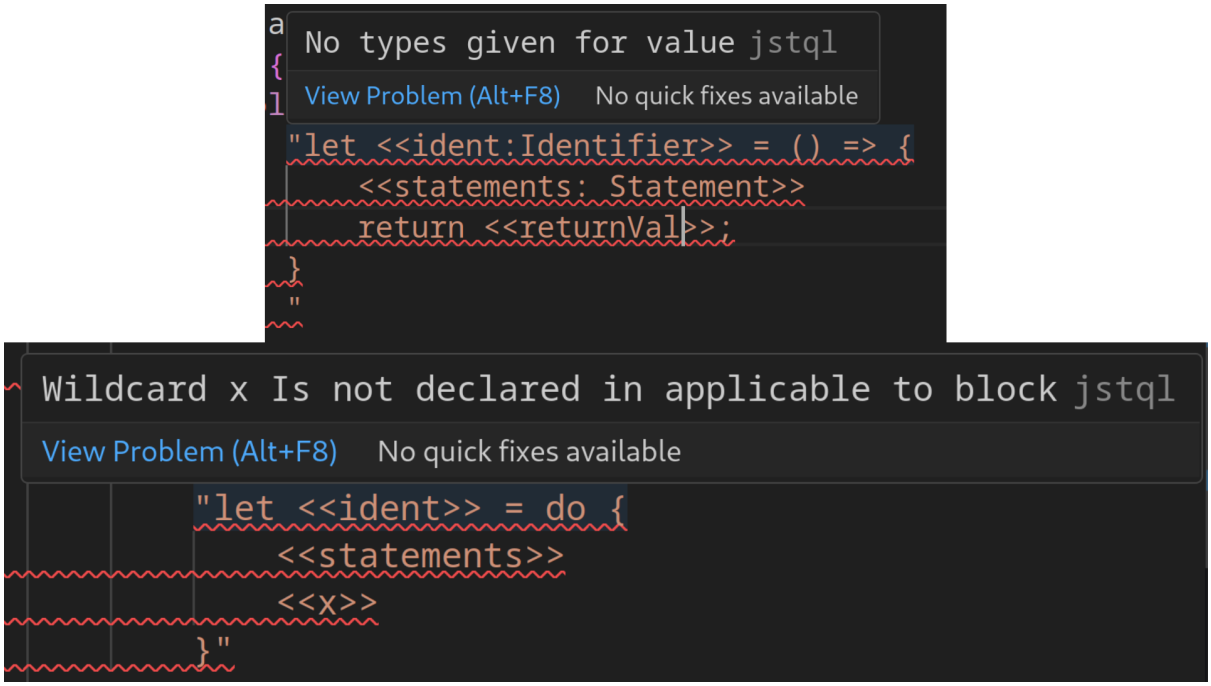

Figure 3.2: Errors of wildcards

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.

3.4.1 "Pipeline" Proposal

The Pipeline proposal is the first we define of the proposals presented in Section 3.2. This is due to the proposal being applicable to function calls, which is 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 {
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 wildcard `someFunctionIdent`, has the types of `Identifier`, to match against single identifiers, 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[16] can be specified in our tool. 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 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.

```
proposal DoExpression {
    case arrowFunction {
        applicable to {
            "() => {
                <<statements: (Statement && !ReturnStatement)+>>
                return <<returnVal : Expression>>;
            }
            "
        }
        transform to {
            "(do {
                <<statements>>
                <<returnVal>>
            })"
        }
    }

    case immediatelyInvokedAnonymousFunction {
        applicable to {
            "(function(){
                <<statements: (Statement && !ReturnStatement)+>>
                return <<returnVal : Expression>>;
            })();"
        }

        transform to {
            "(do {
                <<statements>>
                <<returnVal>>
            })"
        }
    }
}
```

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[11] 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.

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 awaitToPromise {
2      case single{
3          applicable to {
4              "let <<ident:Identifier>> = await <<awaitedExpr:
                   ↪ Expression>>;
5              <<statements: (Statement && !ReturnStatement)*>>
6              return <<returnExpr: Expression>>
7              "
8          }
9
10         transform to{
11             "return <<awaitedExpr>>.then((<<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.

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.

```
// Definition in JSTQL
proposal a{
    case {
        applicable to {
            <<a:Expression>>
        }
        transform to {
            () => <<a>>
        }
    }
}
```

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

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, circular nodes show data passed into the program sections, and square nodes is a specific section of the program.

JSTQL Code is the raw text definition of proposals
Self-Hosted Object is the self-hosted version in JSTQL-SH format

1. **Langium Parser** takes raw JSTQL source code, and parses it into a DSL
2. **Pre-parser** extracts the wildcards from the raw template definitions

1. **Prelude-builder** translates JavaScript prelude into array of wildcard strings

3. **Babel** parses the templates and the users source code into an AST
4. **Custom Tree Builder** translates the Babel AST structure into our tree structure
5. **Matcher** finds matches with `applicable to` template in user code
6. **Transformer** performs transformation defined in `transform to` template to each match of the users AST
7. **Generator** generates source code from the transformed user AST

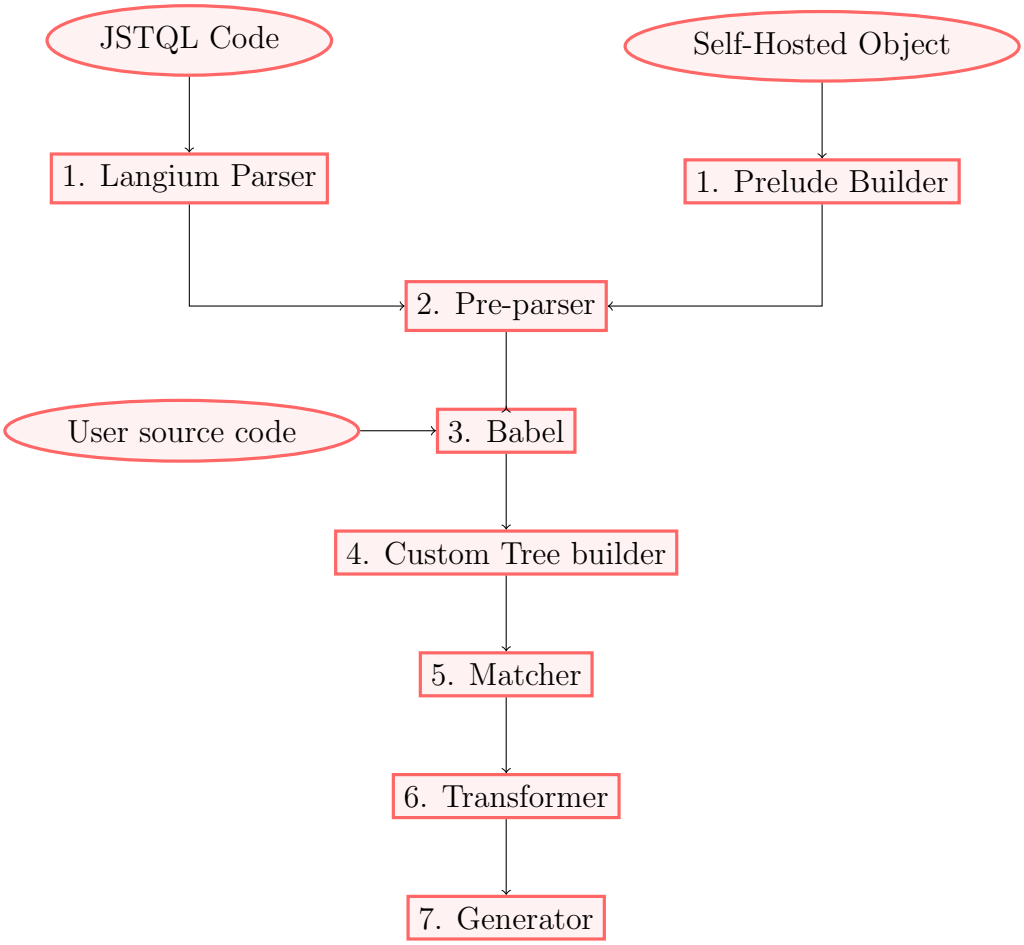

Figure 4.1: Overview of tool architecture

4.2 Parsing JSTQL using Langium

In this section, the implementation of the parser for JSTQL will be described. This section will 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 [14] is a language workbench primarily used to create parsers and Integrated Development Environments for domain specific languages. These kinds of parsers produce Abstract Syntax Trees that is 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.

In order to generate this parser, Langium requires a definition of a grammar. A grammar is a specification that describes a valid program. The JSTQL grammar describes the structure of JSTQL , such as `proposals`, `cases`, `applicable to`, and `transform to`. 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 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[14].

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.

```
1  grammar Jstql
2
3  entry Model:
4      (proposals+=Proposal)*;
5
6  Proposal:
7      'proposal' name=ID "{"
8          (case+=Case)+
9      "}";
10
11 Case:
12     "case" name=ID "{"
13         aplTo=ApplicableTo
14         traTo=TransformTo
```

```
15      "}";
16
17  ApplicableTo:
18      "applicable" "to" "{"
19          apl_to_code=STRING
20      "}";
21  TransformTo:
22      "transform" "to" "{"
23          transform_to_code=STRING
24      "}";
25  hidden terminal WS: /\s+/;
26  terminal ID: /[_a-zA-Z][\w_]*/;
27  terminal STRING: /"[^"]*"|'[^']*'/;
```

Listing 4.1: Definition of JSTQL in Langium.

In the case of 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[25]. 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 only the grammar we cannot actually verify the wildcards placed in `apl_to_code` and `transform_to_code` are correctly written. This is done by using a feature of Langium called `Validator`.

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 a wildcard cannot be untyped or allow any AST type to match against it. This is not possible to verify within the grammar, as inside the grammar the code is simply 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[14] 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 from each `pair` and create the `TransformRecipe`. This is the official 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 code. Therefore, it was decided for this project to keep the parsing of the wildcards within the tool itself. If we were to use Langium generators to parse the wildcards, it would make JSTQL-SH not entirely independent, and the entire tool would rely 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[2], 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 just 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 187 of the example below.

```
export function parseInternal(code: string): InternalParseResult {
    for (let i = 0; i < code.length; i++) {
        if (code[i] === "<" && code[i + 1] === "<") {
            // From now in we are inside of the DSL custom block
            flag = true;
            i += 1;
            continue;
        }

        if (flag && code[i] === ">" && code[i + 1] === ">") {
            // We encountered a closing tag
            flag = false;
            try{
                let wildcard = new WildcardParser(
                    new WildcardTokenizer(temp).tokenize()
                ).parse();
                cleanedJS +=
                    ↪ collisionAvoider(wildcard.identifier.name);

                prelude.push(wildcard);
                i += 1;
                temp = "";
                continue;
            }
            catch (e){
                // We probably encountered a bitshift operator, append
                    ↪ temp to cleanedJS
            }

        }
        if (flag) {
            temp += code[i];
        } else {
            cleanedJS += code[i];
        }
    }
    return { prelude, cleanedJS };
}
```

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 Tree to be used when matching against the wildcard. This is accomplished by using a simple tokenizer and a [27]recursive descent parser.

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 determining 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 is created to closely mimic the grammar of the language the parser is implemented 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.

```
Wildcard:
    Identifier ":" MultipleMatch

MultipleMatch:
    GroupExpr "*"
    | TypeExpr

TypeExpr:
    BinaryExpr
    | UnaryExpr
   | PrimitiveExpr

BinaryExpr:
    TypeExpr { Operator TypeExpr }*

UnaryExpr:
    {UnaryOperator}? TypeExpr

PrimitiveExpr:
    GroupExpr | Identifier

GroupExpr:
    "(" 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 *solvable* by the Recursive Descent parser.

Our recursive descent parser produces an [30, 31]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 [34]visitor pattern and comparing each `Identifier` against the specific AST node we are currently checking, 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 a tool [2, Babel].

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[18] through its plugin system, which allows the parsing of code not yet part of the language.

Custom Tree Structure

To allow for matching and transformations to be applied to each of the sections inside a `pair` definition, they have to be parsed into and AST in order to allow the tool to match and transform accordingly. To do this the tool uses the library [2, Babel] to generate an AST data structure. However, this structure does not suit traversing multiple trees at the same time, this is a requirement for matching and transforming. Therefore we use this Babel 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.

```
1  export class TreeNode<T> {
2      public parent: TreeNode<T> | null;
3      public element: T;
4      public children: TreeNode<T>[] = [];
5
6      constructor(parent: TreeNode<T> | null, element: T) {
7          this.parent = parent;
8          this.element = element;
9          if (this.parent) this.parent.children.push(this);
10     }
11 }
```

Listing 4.4: Simple definition of a Tree structure in TypeScript

Placing the AST generated by Babel into this structure means utilizing the library [5]Babel Traverse. Babel Traverse uses the [34]visitor pattern to allow for 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 in order to place it into our simple tree structure.

[5]Babel Traverse uses the [34]visitor pattern to visit each node of the AST in a *depth first* manner, the idea of this pattern is one implements a *visitor* for each of the nodes in the AST and when a specific node is visited, that visitor is then used. In the case of transferring the AST into our simple tree structure we simply have to use the same visitor for all nodes, 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.

```
traverse(ast, {
        enter(path: any) {
            let node: TreeNode<t.Node> = new TreeNode<t.Node>(
                last,
                path.node as t.Node
            );

            if (last == null) {
                first = node;
            }
            last = node;
        },
        exit(path: any) {
            if (last && last?.element?.type != "Program") {
                last = last.parent;
            }
        },
    });
    if (first != null) {
        return first;
    }
```

4.5 Matching

Performing the match against the users code it the most important step, as if no matching code is found the tool will do no transformations. Finding the matches will depend entirely on how well the definition of the proposal is written, and how well the proposal actually can be defined within the confines of JSTQL . In this chapter we will discuss how matching is performed based on the definition of `applicable to`

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.

In the pre-parsing step of JSTQL we are replacing each of the wildcards with an expression of type Identifier, this means we are inserting an Identifier at either a location where an expression resides, or a statement. In the case of the identifier being placed where a statement should reside, it will be wrapped in an ExpressionStatement. This has to be taken into account when comparing statement nodes from the template and user code, as if we encounter an ExpressionStatement, its corresponding expression has to be checked for if it is an Identifier.

Since a wildcard is replaced by an Identifier, when matching a node in the template, we have to check if it is the *Identifier* or *ExpressionStatement* with an identifier contained within, if there is an identifier, we have to check if that identifier is a registered wildcard. If an Identifier shares a name with a wildcard, we have to compare the node against the Type expression of that wildcard. When we do this, we traverse the entirety of the wildcard expression AST and compare each of the leaves against the type of the current code node. These resulting values are then passed through the type expression and the resulting value is whether or not that code node can be matched against the wildcard. We differentiate between if a node matched against a wildcard with the `+` notation, as if that is the case we have to keep using that wildcard until it returns false in the tree exploration algorithms.

When we are either matching against an Identifier that is not a registered wildcard, or any other AST node in the template, we have to perform an equality check, in the case of this template language, we can get away with just performing some preliminary checks, such as that names of Identifiers are the same. 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.

The function used for matching singular nodes will give different return values based on how they were matched. The results NoMatch and Matched are self explanatory, they are used when either no match is found, or if the nodes types match and the template node is not a wildcard. When we are matching against a wildcard, if it is a simple wildcard that cannot match against multiple nodes of the code, the result will be `MatchedWithWildcard`. If the wildcard used to match is a one or many wildcard, the result will be `MatchedWithPlussedWildcard`, as this shows the recursive traversal algorithm used that this node of the template have to be tried against the code nodes sibling.

```
enum MatchResult {
    MatchedWithWildcard,
    MatchedWithPlussedWildcard,
    Matched,
    NoMatch,
}
```

Matching a singular Expression/Statement template

The method of writing the `applicable to` section using a singular simple expression/statement is by far the most versatile way of defining matching template, this is because there will be a higher probability of discovering applicable code with a template that is as generic and simple as possible. A very complex matching template with many statements or an expression containing many AST nodes will result in a lower chance of finding a resulting match in the users code. Therefore using simple, single root node matching templates provide the highest possibility of discovering a match within the users code.

Determining if we are currently trying to match 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 1, if it does we can use the singular expression matcher, if not, we have to rely on the matcher that can handle multiple statements at the head of the tree.

When matching an expression the first statement in the program body of the AST generated when using [3]babel generate will be of type `ExpressionStatement`, the reason for this is Babel will treat free floating expressions as a statement, and place them into an ExpressionStatement. This will miss many applicable sections in the case of trying to match against a users code because expressions within other statements are not inside an ExpressionStatement. This will give a template that is incompatible with a lot of otherwise applicable expressions. This means the statement ExpressionStatement has to be removed, and the search has to be done with the expression as the top node of the template.

In the case of the singular node in the body of the template program being a Statement, no removal has to be done, as a Statement can be used directly.

Recursively discovering matches The matcher used against single Expression/Statement templates is based upon a Depth-First Search in order to perform matching, and searches for matches from the top of the code definition. 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 tree is the top node of the template. This ensures we have tried to perform a match at any level of the tree, this also means we do not get any partial matches, as we only store matches that are returned at the recursive call when we do the search from the first node of the template tree. This is all done before ever checking the node we are currently on. The

reason for this is to avoid missing matches that reside further down in the current branch, and also ensure matches further down are placed earlier in the full match array, which makes it easier to perform transformation when partial collisions exist.

Once we have started a search on all the child nodes of the current one using the full definition of `applicable to`, we can verify if we are currently exploring a match. This means the current node is checked against the current top node of `applicable to`, if said node is a match, based on what kind of match it is several different parts of the algorithm are called. This is because there are different forms of matches depending on if it is a match against a wildcard, a wildcard with `+`, or simply a node type match.

If the current node matches against a wildcard that does not use the `+` operator, we simply pair the current template node to the matched node from the users code and return. This is because whatever the current user node contains, it is being matched against a wildcard and that means no matter what is below it, it is meant to be placed directly into the transformation. Therefore we can determine that this is a match that is valid.

When the current node is matched against a wildcard that does use the `+` operator, we have to continue trying to match against that same wildcard with the sibling nodes of the current code node. This is performed in the recursive iteration above the current one, and therefore we also return the paired AST nodes of the template and the code, but we give the match result `MatchResult.MatchedWithPlussedWildcard` to the caller function. When the caller function gets this result, it will continue trying to match against the wildcard until it receives a different match result other than `MatchResult.MatchedWithPlussedWildcard`.

When the current node is matched based on the types of the current AST nodes, some parts have to hold. Namely, all child nodes of the template and the user code have to also return some form of match, this means if any of the child nodes currently return `MatchResult.NoMatch` the entire match is discarded. The number of child nodes of the current match also has to be equal. Due to wildcards this means we have to be able to match all child nodes of the user code to either a single node of the template, or a wildcard using the `+` operator.

If the current node does not match, we simply discard the current search, as we have already started a search from the start of the template at all levels of the user code AST, we can safely end the search and rely on these to find matches further down in the tree.

To allow for easier transformation, and storage of what exact part of `applicable to` was matched against the exact node of the code AST, we use a custom instance of the simple tree structure described in 4.4, we use an interface `PairedNode`, this allows us to hold what exact nodes were matched together, this allows for a simpler transforming algorithm. The exact definition of `PairedNode` can be seen below. The reason the codeNode is a list, is due to wildcards allowing for multiple AST nodes to match against, as they might match multiple nodes of the user code against a single node of the template.

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

Matching multiple Statements

Using multiple statements in the template of `applicable to` will result in a much stricter matcher, that will only try to perform an exact match using a [28]sliding window of the amount of statements at every *BlockStatement*, as that is the only placement Statements can reside in JavaScript[12].

The initial step of this algorithm is to search through the AST for ast nodes that contain a list of *Statements*, this can be done by searching for the AST nodes *Program* and *BlockStatement*, as these are the only valid places for a list of Statements to reside [12]. Searching the tree is quite simple, as all that is required is checking the type of every node recursively, and once a node that can contain multiple Statements, we check it for matches.

Once a list of *Statements* has been discovered, the function `matchMultiHead` can be executed with that block and the Statements of `applicable to`. This function will use the technique [28]sliding window to match multiple statements in order the same length as the list of statements are in `applicable to`. This sliding window will try to match every Statement against its corresponding Statement in the current *BlockStatement*. When matching a singular Statements in the sliding window, a simple DFS recursion algorithm is applied, similar to algorithm used for matching a single expression/statement template, however the difference is that we do not search the entire AST tree, 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 `+`, as they can match one or more nodes against each other. These wildcards might match against `(Statement)+`. Therefore, we have to use a two point technique when iterating through the statements of the users code. As we might have to use the same statement from the template multiple times.

Output of the matcher

The resulting output of the matcher after finding all available matches, is a two dimensional array of each match, where for every match there is a list of statements in AST form, where paired ASTs from `applicable to` and the users code can be found. This means that for every match, we might be transforming and replacing multiple statements in the transformation function.

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

4.6 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 AST version of original code parsed by Babel. All the transformations are then applied to the code and we use [3]Babel generate to generate JavaScript code from the transformed AST.

An important discovery is to ensure we transform the leaves 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.

Preparing the transform to template

The transformations are performed by inserting the matched wildcards from the applicable to template into their respective locations in the transform to template. Then the entire transform to template is placed into the original code AST where the match was discovered. Doing this we are essentially doing a transformation that is a find and replace with context passed through the wildcards.

In order to perform the transformation, all the sections matched against a wildcard have to be transferred into the `transform to` template. We utilize the functionality from Babel here and traverse the generated AST of the transform to template using [5]Babel traverse, as this gives ut utility functions to replace, replace with many, and remove nodes of the AST. We use custom visitors for *Identifier* and *ExpressionStatement* with an Identifier as expression, in order to determine where the wildcard matches have to be placed, as they have to placed at the same location that shares a name with the wildcard. Once a shared identifier between the `transform to` template and the `applicable to` template is discovered, a babel traverse replace with multiple is performed and the node/s found in the match is inserted in place of the wildcard.

Inserting the template into the AST

Having a transformed version of the users code, it has to be inserted into the full AST definition of the users code, again we use [5]babel/traverse to traverse the entirety of the code AST using a visitor. This visitor does not apply to any node-type, as the matched section can be any type. Therefore we use a generic visitor, and use an equality check to

find the exact part of the code this specific match comes from. Once we find where in the users code the match came from, we replace it with the transformed `transform to` nodes. This might be multiple Statements, therefore the function `replaceWithMultiple` is used, to insert every Statement from the `transform to` body, and we are careful to remove any following sibling nodes that were part of the original match. This is done by removing the *n-1* next siblings from where we inserted the transform to template.

To generate JavaScript from the transformed AST created by this tool, we use a JavaScript library titled [3]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.

Testing on code

In this section, we will showcase some synthetic transformations applied to code made to fit each of the definitions discussed in this thesis.

The pipeline proposal is meant to merge the readability of chaining with the practicality of deeply nested calls. This deep nesting can be seen in the input code Listing 5. The resulting code can be seen in Listing 5.

```
1 // Original JavaScript
2 a(a(a(a(a(a(b)))))));
3 c(c(c(c(c(d, b), b), b), b), b);
```

```
1 // Transformed
2 b |> a(%) |> a(%) |> a(%) |> a(%) |> a(%) |> a(%) |> a(%);
3 d |> c(%, b) |> c(%, b) |> c(%, b) |> c(%, b) |> c(%, b);
```

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 [3] without error. You can see some highlighted transformations in Listing **INSERT FIGURE HERE**.

"Pipeline"[17] is clearly very applicable to most files, as it is really only looking for function calls. This is by far the best result, and it found matches in almost all files that Babel [4] managed to parse.

The Do proposal [16] is expected to not find many matches, as code that has not been written in expression-oriented programming style will not produce many matches.

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`.

When the amount of files searched is different for each of the proposals, it means either a transformation on that file, or generation of that file failed for some reason. This is probably due to bugs in the transform algorithm that creates this discrepancy.

Next.js [15] is one of the largest projects related to JavaScript. It is owned by Vercel and is used as a development platform for the web.

Three.js [24] 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.

React [19] 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.

Bootstrap [7] 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.

Atom [1] 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	Time
Pipeline	55787	1327	2331	89m 11s
Do	131	74	2331	35s
Await to Promise	43	38	2331	16s

Figure 5.1: Evaluation with Next.js source code

Proposal	Matches found	Files with matches	Files searched	Time
Pipeline	85050	1119	1262	242m 58s
Do	278	55	1385	3m 3s
Await to Promise	125	80	1262	59s

Figure 5.2: Evaluation with Three.js source code

Proposal	Matches found	Files with matches	Files searched	Time
Pipeline	16353	1266	2051	84s
Do	79	60	2051	8s
Await to Promise	104	88	2051	6s

Figure 5.3: Evaluation with React source code

Proposal	Matches found	Files with matches	Files searched	Time
Pipeline	13794	109	115	2m 57s
Do	13	8	115	3s
Await to Promise	0	0	115	2s

Figure 5.4: Evaluation with Bootstrap source code

Proposal	Matches found	Files with matches	Files searched	Time
Pipeline	40606	361	401	16m 17s
Do	46	26	401	6s
Await to Promise	8	6	401	4s

Figure 5.5: Evaluation with Atom source code

Chapter 6

Related Work

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

Aspect Oriented Programming

Aspect oriented programming, is a programming paradigm that allows for increased modularity by allowing for a high degree of separation of concerns, specifically focusing on cross-cutting concerns. Cross-cutting concerns are concerns that are present across classes and across separations within the program.

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 *Advice*,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 it performs a re-write of the code during execution to add functionality to multiple places in the executing code.

Other source code query languages

In order to allow for simple analysis and refactoring of code, there already exists query languages for querying 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.

Browse-By-Query

Browse-By-Query is a language created for Java that analyses Java Bytecode files, and builds a database structure to represent the code classes, method calls, and other sections contained within the code. The language uses english-like queries combined with filtering and set operators to allow for more complex queries. This language was created as a way to browse large code-bases like one is browsing a database. Since BBQ builds the source code into something resembling a database, queries can be done with respect to each objects relationship in the original code, and complex and advanced queries based on relationships are possible using this method.

.QL

.QL is an object-oriented query language. It supports querying a wide array of data structures, code being one of them. [35].QL has a commercial implementation *Semmle-Code*, which comes with a full editor and various pre-defined code transformations that might be useful for the end developer.

PMD XPath

PMD is the most versatile query language for Java source code querying out of all the ones explored in this section. [35]PMD supports querying of all Java constructs , it has this wide support due to constructing the entire codebase in XML format. This language was build for static code analysis, and therefore is a great way to perform queries on static code, it is mostly used as a tool for code editors to enforce programming styles.

Jackpot

[32]Jackpot is a query language created for the [33]Apache Netbeans platform, 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.

JetBrains structural search

JetBrains integrated development environments have a feature that allows for [21] structural search and replace. 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 exprience, 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.

6.1 Other JavaScript parsers

Speedy Web Compiler

[20]Speedy Web Compiler 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 [2]Babel, 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 is another parser written in JavaScript to parser 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. It has a very similar syntax to and has it's own tree traversal library Acorn Walk. [2]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.

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 developement 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 [2]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] atom, May 2024. [Online; accessed 23. May 2024].

[2] Babel · Babel, May 2024. [Online; accessed 10. May 2024].

[3] @babel/generator · Babel, May 2024. [Online; accessed 12. May 2024].

[4] @babel/parser · Babel, May 2024. [Online; accessed 14. May 2024].

[5] @babel/traverse · Babel, May 2024. [Online; accessed 12. May 2024].

[6] Bikeshedding the Hack topic token · Issue #91 · tc39/proposal-pipeline-operator, May 2024. [Online; accessed 24. May 2024].

[7] bootstrap, May 2024. [Online; accessed 23. May 2024].

[8] ECMAScript® 2025 Language Specification, May 2024. [Online; accessed 24. May 2024].

[9] ECMAScript® 2025 Language Specification, May 2024. [Online; accessed 24. May 2024].

[10] ECMAScript® 2025 Language Specification, May 2024. [Online; accessed 24. May 2024].

[11] ECMAScript® 2025 Language Specification, May 2024. [Online; accessed 25. May 2024].

[12] ECMAScript® 2025 Language Specification, April 2024. [Online; accessed 13. May 2024].

[13] Functions · The Julia Language, May 2024. [Online; accessed 24. May 2024].

[14] Langium, April 2024. [Online; accessed 10. May 2024].

[15] next.js, May 2024. [Online; accessed 23. May 2024].

[16] proposal-do-expressions, May 2024. [Online; accessed 2. May 2024].

[17] proposal-pipeline-operator, May 2024. [Online; accessed 21. May 2024].

[18] proposals, May 2024. [Online; accessed 27. May 2024].

[19] react, May 2024. [Online; accessed 23. May 2024].

[20] Rust-based platform for the Web – SWC, May 2024. [Online; accessed 21. May 2024].

[21] Structural search and replace | IntelliJ IDEA, April 2024. [Online; accessed 22. May 2024].

[22] TC39 - Specifying JavaScript., May 2024. [Online; accessed 26. May 2024].

[23] The TC39 Process, April 2024. [Online; accessed 24. May 2024].

[24] three.js, May 2024. [Online; accessed 23. May 2024].

[25] TOML: Tom's Obvious Minimal Language, May 2024. [Online; accessed 27. May 2024].

[26] Boyko B. Bantchev. Putting more meaning in expressions. *SIGPLAN Not.*, 33(9):77–83, September 1998.

[27] Matthew S. Davis. An object oriented approach to constructing recursive descent parsers. *SIGPLAN Not.*, 35(2):29–35, feb 2000.

[28] 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.

[29] KathleenDollard. Symbol and Operator Reference - F#, May 2024. [Online; accessed 24. May 2024].

[30] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery.

[31] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, may 2005.

[32] Apache NetBeans. Java Declarative Hints Language, March 2024. [Online; accessed 21. May 2024].

[33] Apache NetBeans. Welcome to Apache NetBeans, February 2024. [Online; accessed 21. May 2024].

[34] J. Palsberg and C.B. Jay. The essence of the visitor pattern. In *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)*, pages 9–15, 1998.

[35] Raoul-Gabriel Urma and Alan Mycroft. Programming language evolution via source code query languages. In *Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '12, page 35–38, New York, NY, USA, 2012. Association for Computing Machinery.

Appendix A

Generated code from Protocol buffers

```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