University of Bergen
Department of Informatics

Implementing Structural Search and Replace for Showcasing ECMAScript Language Proposals

Author: Rolf Martin Glomsrud
Supervisor: Mikhail Barash
Informal advisor: Yulia Startsev

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

October, 2024

Abstract

Technical Committee 39 (TC39) of Ecma International is the body responsible for the evolution of the ECMAScript programming language, better known as JavaScript. Suggested changes to the language are presented in a form of proposals. To allow JavaScript users to form opinions about a proposal during the extensive design stage, proposal descriptions mention examples that showcase various corner cases. In this thesis, we implement a tool to search a user's codebase and demonstrate how the codebase would look like if the functionality defined by a proposal were a part of the language. We evaluate our tool on two contentious ECMAScript proposals ("Do Expression" and "Pipeline") and demonstrate that specifying proposals and transforming user code is feasible. The work presented in this theses is an initial step in creating a language workbench-like tool to aid in the development and design of widely adopted programming languages.

Acknowledgements

First of all, I would like to give my deepest appreciation to my supervisor Assoc. Prof. Mikhail Barash for his incredible guidance throughout this thesis. He has given me so much in this journey, his kind words and thought-provoking discussion has been invaluable. Having him as my advisor is what made writing this thesis a positive experience, and I will always be indebted to him for that.

I would also like to thank Yulia Startsev (TC39), who has been my informal advisor throughout this thesis. She has suggested the idea of which this thesis is based on, and provided deep technical knowledge when I was stuck on the implementation. The brain-storming sessions during meetings are what has made this thesis a reality.

I also want to express my thanks to Daniel Svalestad Liland. Without his continued jokes, motivation, and competitive spirit I would not be where I am today.

My acknowledgements would not be complete without mentioning all my fellow master students. While they have been distracting at times, the support and community they provided has been incredibly important to me.

Last but not least, I want to thank my family for always supporting me throughout my studies, this thesis would not have been possible without them.

`Rolf Martin Glomsrud`
Tuesday 29th October, 2024

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The development and evolution of the programming language ECMAScript—which is defined by the ECMA-262 language standard—is done by the Technical Committee 39 of Ecma International. The committee has the responsibility to investigate proposals suggested for addition into the ECMASript language. During this process, proposals go through numerous iterations of improving the solution space of the problem identified in a proposal. The community of JavaScript developers can give feedback on proposals; this feedback has to be of a certain quality—it is, therefore, crucial that the users are confident in their understanding of the proposal, the suggested solution, and its potential corner cases. To aid users in this understanding, the description of a proposal is expected to illustrate the solution by presenting several examples—in form of ECMAScript code snippets—that highlight various scenarios for the use of the functionality suggested in a proposal. In this thesis, we suggest a way of demonstrating these scenarios in a user's own codebase. We conjecture this will lower the barrier of understanding a proposal, and will allow the user to focus solely on the concepts a proposal introduces.

This thesis discusses a way of defining transformations of code specifically for *syntactic* proposals—these are proposals that do not introduce any new semantics to the language, but merely improve the ergonomics of how the code is written. The idea is to identify code fragments in a user's codebase to which a proposal can be applied, and then replace that code with an equivalent code that uses the functionality introduced in a proposal.

We developed a domain-specific language called "JSTQL" (for "JavaScript Transformation Query Language") for specifying queries and transformations on JavaScript code. This DSL utilizes JavaScript templates to query user code and to define how the matched code should be transformed. To parse both the templates and user code, we employ the Babel [?] library. The templates defined in JSTQL may include variables—referred to as *wildcards*—which are special blocks written inside the template. These blocks facilitate matching against arbitrary code, and transforming that code according to the specified transformation; this allows the transformed code to maintain its *context*. To specify what kind of code a wildcard will match against, we use *type expressions*, which are Boolean propositions on the node types as defined in the Babel abstract syntax tree specification.

The evaluation of the transformation tool implemented in this thesis involved specifying the proposals "Do Expression" [?] and "Pipeline" [?] in our DSL. These specifications were applied to existing large codebases in order to assess the functionality of the transformations. The results obtained from this process confirmed the functionality of the tool, and provided insights into how significant of an "impact" the design decisions in each proposal might have on existing codebases.

The transformation tool presented in this thesis is meant to be the initial step in creating a language workbench-like tool for designing widely adopted programming languages. We created the core machinery of transforming code based on a proposal specification, while implementing ways to present this to users and gather feedback on proposals is left up to future work.

Chapter 2

Background

Below we give an overview of the evolution process of the ECMAScript programming language, abstract syntax trees, source code querying, domain-specific languages, and language workbenches. These are instrumental to the implementation of the tool described in this thesis.

2.1 Evolution of the JavaScript programming language

Technical Committee 39 (TC39) is a technical committee within Ecma International, whose main goal is to develop the language standard for the ECMAScript programming language (informally known as JavaScript); this standard is known as ECMA-262 [**?**]. Apart from this standard, the committee is also responsible for maintaining related standards: on internalization API (ECMA-402), the standard for JSON (ECMA-404), and ECMAScript specification suite (ECMA-414). The members of the committee are representatives of companies, academic institutions, and other organizations interested in developing and maintaining the ECMAScript language. The delegates include experts in JavaScript engines, tooling surrounding JavaScript, and other areas of the JavaScript ecosystem.

ECMA-262 Proposals We explain now what a proposal is, and how proposals are developed in TC39 for the ECMA-262 language standard.

A *proposal* is a suggested change to the ECMA-262 language standard. These additions to the standard have to solve some form of problem with the current version of ECMAScript. Such problems can come in many forms, and can apply to any part of the language. Examples include: a feature that is not present in the language, inconsistent parts of the language, simplification of common patterns, and so on. The proposal development process is defined in the *TC39 Process Document* [?], which describes each stage a proposal has to go through in order to be accepted into the ECMA-262 language standard.

The purpose of *stage 0* of the process is to allow for exploration and ideation around which parts of the current version of ECMAScript can be improved, and then to define a problem space for the committee to focus on improving.

At *stage 1*, the committee will start development of a proposal. In order for a proposal to enter this stage, several requirements have to be fulfilled. First, a champion—a delegate of the committee who will be responsible for the advancement of the proposal—has to be identified. In addition, a rough outline of the problem must be provided, and a general shape of a solution must be given. There must have been a discussion around key algorithms, abstractions and semantics of the proposal. Exploration of potential implementation challenges and cross-cutting concerns must have been done. The final requirement is for all parts of the proposal to be captured in a public repository. Once all these requirements are met, a proposal is accepted into stage 1. During this stage, the committee will work on the design of a solution, and resolve any cross-cutting concerns discovered previously.

At *stage 2*, a preferred solution has been identified. Requirements for a proposal to enter this stage are as follows: all high level APIs and syntax must be described in the proposal document, illustrative examples have to be worked out, and an initial specification text must be drafted. During this stage, the following areas of the proposal are explored: refining the identified solution, deciding on minor details, and create experimental implementations.

At *stage 2.7*, the proposal is principally approved, and has to be tested and validated. To enter this stage, the major sections of the proposal must be complete. The specification text should be finished, and all reviewers of the specification have approved. Once a proposal has entered this stage, testing and validation will be performed. This is done through the prototype implementations at stage 2.

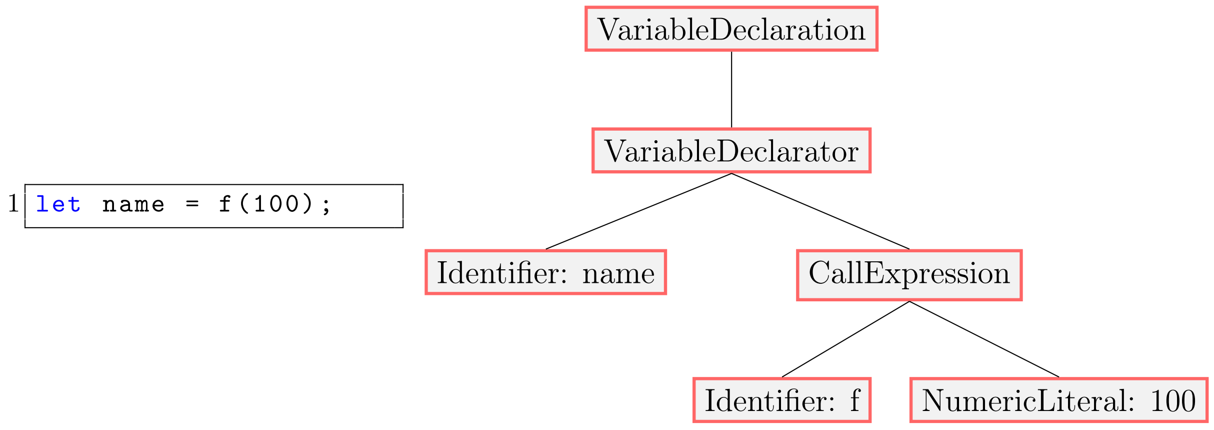
Once a proposal has been sufficiently tested and verified, it is moved to *stage 3*. During this stage, the proposal should be implemented in at least two major JavaScript engines. The proposal should be tested for web compatibility issues, and integration issues in the major JavaScript engines.

At *stage 4* the proposal is completed and will be included in the next revision of the ECMA-262.

2.2 Abstract Syntax Trees

An *abstract syntax tree* (AST) is a tree representation of source code. Every node of such a tree represents a construct from the source code. ASTs remove syntactic details while maintaining the *structure* of the program. Each node is set to represent constructs of the programming language, such as statements, expressions, declarations, and so on. Thus, every node type represents a grammatical construct in the language the AST was built from.

ASTs are important for manipulating source code; they are used by various tools that need to represent source code in some way to perform operations with it [?]. Using ASTs is favored over raw text due to their structured nature; this especially manifests when considering tools like compilers, interpreters, or code transformation tools. ASTs are produced by language *parsers*. For JavaScript, one of the popular libraries used for parsing is *Babel* [?]. Babel is a JavaScript toolchain, and its main usage is converting source code written in the version ECMASCript 2015 or a newer one into older versions of JavaScript. This conversion is done to increase the compatibility of JavaScript in older execution environments. Babel has a suite of libraries used to work with JavaScript source code. Each library relies on Babel's AST definition [?]. The AST specification Babel uses tries to stay as close as possible to the ECMAScript standard [?]. This fact has made Babel a recommended parser to use for proposal transpiler implementations [?]. A simple example of how source code parsed into an AST with Babel can be seen in Figure ??.



```
let name = f(100);
```

Figure 2.1: Example of source code parsed to Babel AST.

To achieve compilation of newer versions into older versions, Babel uses a *plugin* system that allows a myriad of features to be enabled or disabled. This makes the parser versatile to fit different ways of working with JavaScript source code. Because of this, Babel allows parsing of JavaScript experimental features. These features are usually proposals that are under development by TC39, and the development of these plugins are a part of the proposal deliberation process. This allows for experimentation as early as *stage 1* of the proposal development process. Some examples of proposals that were first supported by Babel's plugin system are "Do Expression" [?] and "Pipeline" [?]. These proposals are both currently at *stage 1* and *stage 2*, respectively.

In this project, we will use Babel to parse JavaScript into abstract syntax trees. This choice was made because of Babel's support of very early stage proposals.

2.3 Source Code Querying

Source code querying is the action of searching source code to extract some information or find specific sections of code. Source code querying comes in many forms, the simplest of which is text search. Since source code is primarily text, one can apply text search techniques to perform a query, or a more complex approach using regular expressions (e.g., tools like `grep`). Both these methods do not allow for queries based on the structure of the code, and rely solely on its syntax. AST-based queries allow queries to be written based on both syntax and structure, and are generally more powerful than regular text based queries. Another technique for code querying is based on semantics of code.

The primary use cases for source code querying are code understanding, analysis, code navigation, enforcement of styles, along with others. All these are important tools developers use when writing programs, and they all rely on some form of source code queries. One such tool is Integrated Development Environments (IDEs), as these tools are created to write source code, and, therefore, rely on querying the source code for many of their features. One such example of code querying being used in an IDE is JetBrains IntelliJ *structural search and replace* [?], where queries are defined based on code structure to find and replace sections of our program.

2.4 Domain-Specific languages

Domain-specific languages (DSLs) are software languages specialized to a specific narrow domain [?]. DSLs allow domain experts to get involved in the software development process, as it is expected that a domain expert would have the capabilities to read and write DSL code. A domain-specific language allows for very concise and expressive code to be written that is specifically designed for the domain. Using a DSL might result in faster development because of this expressiveness within the domain; this specificity to a domain might also increase correctness. However, there are also some disadvantages to DSLs: the restrictiveness of a DSL might become a hindrance if it is not well designed to represent the domain. Domain-specific languages also might have a learning curve, this makes these language less accessible for the target users. Developing a domain-specific language might is a non-trivial process [?], as implementing a DSL requires both knowledge of the domain and knowledge of software language engineering.

2.5 Language Workbenches

A *language workbench* [?] is an integrated development environment created to facilitate the development of a software language, such as a domain-specific language. The goal of a language workbench is to give increased productivity during development, and to enhance the design and evolution of software languages [?].

Commonly language workbenches generate tooling for a software language. One such tool is a language parser that is generated from the language definition within the language workbench. Another such tool commonly generated by a language workbench is an integrated development environment, such IDEs provide functionality such as syntax highlighting, code navigation, error highlighting, along with others.

Chapter 3

A domain-specific language for matching and transforming source code

The tool that we implement in this thesis should allow previewing how an ECMAScript proposal could affect a user's codebase. We only focus on proposals that introduce new syntactic forms that merely abstract certain use patterns that could be otherwise written in JavaScript—but a verbose or less idiomatic manner. We call these *syntactic proposals*. The idea is to identify code fragments in a user's codebase to which a proposal can be *applied*. An application of a proposal can be thought of as a process of identifying the user's code that "matches" a proposal's problem space, and then replacing that code with a *semantically equivalent* code that uses the functionality introduced in the proposal.

Thus, it will be possible to identify all the places in the user's codebase which can be affected by a proposal—and to show to the user how the modified version of the *user's own* code will look like. This way a user will be able to give a very specific feedback to the TC39 committee. Importantly, the fact that a user is familiar with their codebase could potentially allow for a more useful feedback and thus a more efficient process of developing and evolving the ECMAScript programming language.

Implementing the idea outlined here requires some way of *matching* and *transforming* code. A proposal should thus have a precise specification, where the matching and the transformation can be defined. For this purpose, we have designed and implemented a domain-specific language, which will be introduced in this chapter.

3.1 Applicable proposals

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

3.1.1 Simple example of a syntactic proposal

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

An example of this proposal can be seen below:

```
// Original code
let x = 100;
let b = "Some String";
let c = 200;

// Code after application of proposal
int x = 100;
let b = "Some String";
let c = 200;
```

See above that 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 *syntax sugar*, this proposal does not make any changes to functionality or semantics, and can therefore be categorized as a syntactic proposal.

3.1.2 "Pipeline" Proposal

The "Pipeline" proposal [?] 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 can have some specific challenges related to readability. The reading order of nested calls is from right to left rather than the regular reading direction of JavaScript code which is left to right. This means it is difficult to switch the reading direction when working out which call happens in which order. When using functions with multiple arguments in the middle of the nested call, it is not intuitive to see what call its arguments belong to. These issues are the main challenges this proposal is trying to solve. There are currently ways to improve readability with nested calls, as they can be simplified by using temporary variables. While this does introduce its own set of issues, it provides some way of mitigating the readability problem. Another positive side of nested calls is they do not require a specific design to be used, and a library developer does not have to design their library around this specific call style. In the listings below, examples of deeply nested calls with both single and multiple arguments can be seen.

```
// 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 calls solves some of these issues: indeed, as it allows for a more natural reading direction left to right when identifying the sequence of call, arguments are naturally grouped together with their respective function call, and it provides a way of untangling deep nesting. However, executing consecutive operations using chaining has its own set of challenges. To use chaining, the API of the code being called has to be designed to allow for chaining. This is not always the case however, making use of chaining when it has not been specifically designed for can be very difficult. There are also concepts in JavaScript not supported when using chaining, such as arithmetic operations, literals, `await` expressions, `yield` expressions, and so on. This is because all of these concepts would break the chain of calls, and one would have to use temporary variables. In the listings below are examples of both chaining with no arguments other than `self`, and examples where additional arguments are passed during the chain.

```
// Chaining calls
f1().f2().f3();
```

```
// Chaining calls with multiple
    ↪ arguments
f1().f2(v1, v2).f3(v2).f4();
```

The "Pipeline" proposal aims to combine the benefits of these two styles without the challenges each method faces. The proposal wants to achieve a similar style to chaining when doing deeply nested calls. The 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 [?].

The code snippets below showcase the machinery of the proposal. They are the examples used to showcase the proposal in the proposal repository [?].

The example below showcases a left to right ordering of function calls that follows the order of execution.

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

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

In the example below, we can see that functions calls with multiple arguments are supported by “Pipeline”.

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

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

In the example below, we can see the topic token can be used multiple times to pass the same expression several times on the right side of a “Pipeline” proposal expression.

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

```
// With pipes
return xf
      |> bind(%['@@tr..'], %)
      |> obj[methodName](%, acc)
      |> xf['@@tr..'](%);
```

The pipe operator is present in many other languages such as F# [?], Julia [?], Elixir [?] and Unix Shell [?]. The main difference between the these languages pipe operator and the pipe operator suggested in this proposal is the result of the left side expression has to be piped into a function with a single argument, this proposal suggests a topic reference to be used instead, clearly marking where the left side result should be piped to. Proposals suggesting pipe expressions similar to how it is done in F# have been rejected by TC39 multiple times, but were rejected both times due to syntactical concerns and technical challenges [?].

3.1.3 "Do Expression" Proposal

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

The motivation of the "Do Expression" proposal is to allow for local scoping of a code block that is treated as an expression. Thus, complex code requiring multiple statements will be confined inside its own scope [? , Sect. 8.2] and the resulting value is returned from the block implicitly as an expression, similarly to how unnamed functions or arrow functions are currently used. To achieve this behavior in the current version of ECMAScript, one needs to use immediately invoked unnamed functions [? , Sect. 15.2] or immediately invoked arrow functions [? , Sect. 15.3].

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

The current version of JavaScript enables the use of immediately invoked arrow functions with no arguments to achieve similar behavior to "Do Expression", and an example of this can be seen in the listing below. The main difference between immediately invoked arrow functions and "Do Expression" is the final statement/expression will implicitly return its Completion Record [? , Sect. 6.2.4], and thus an explicit `return` statement is not needed.

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

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

The example below is very similar to the one above, and uses an unnamed function [? , 15.2] which is invoked immediately to produce similar behavior to the "Do Expression" proposal.

```
// 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;
};
```

3.1.4 "Await To Promise" (imaginary proposal)

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

To perform this transformation, we define an equivalent way of expressing an `await` expression as a promise. This means removing `await`, this expression now will return a promise, which has a function `then`, this function is executed when the promise resolves. We pass an arrow function as argument to `then`, and append each following statement in the current scope [? , Sect. 8.2] inside the block of that arrow function. This will result in equivalent behavior to using `await`. An example of a function using `await` can be seen below on the left. The example below on the right is the same function but is using a promise.

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

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

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

3.2 Searching user code for applicable parts

To identify parts 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.

3.2.1 Structure of JSTQL

In this section, we describe the structure of JSTQL (JavaScript Template Query Language), that was designed and implemented by the author of this thesis.

Proposal definition JSTQL is designed to mimic the examples already provided in proposal descriptions. These examples can be seen in each of the proposals described in Section **??**. 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.

```
proposal Pipeline_Proposal {}
```

Case definition Each proposal will have one or more definitions of a template for code to identify in the users codebase, and its corresponding transformation definition. These are grouped together to have a simple way of identifying the corresponding cases of matching and transformations. This block 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 parts and showcasing more of the proposal than a single concept the proposal has to offer.

```
    case case_name {

    }
```

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

```
applicable to {
    "let a = 0;"
}
```

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

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

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

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

A complete specification in 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 3.1: JSTQL definition of a proposal.

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

3.2.2 Matching and transforming code

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

Identifying applicable code To identify parts of code a proposal is applicable to, we use *templates*, which are defined using JavaScript. These templates are used to identify and match applicable parts of a users code. A matching part for a template is one that produces an exactly equal AST structure, where each node of the AST 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 can 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 [**?**] to create a very versatile of defining exactly what nodes a wildcard can match against.

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

The example below will allow any node with type `CallExpression` to match against this wildcard named `expr`.

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

To make this more expressive, the type expressions use binary and unary operators. The following operators are supported: `&&` for logical conjunction, `||` for logical disjunction, and `!` for logical negation. This makes it possible to build complex type expressions, enabling us to express exactly what nodes are allowed to match against a specific wildcard.

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

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

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

In the example below, we allow the wildcard to match multiple nodes with the Kleene plus `+`. This example will continue to match against itself as long as the nodes are a `Statement` and at the same time is not a `ReturnStatement`. This example showcases how a wildcard that matches against many sibling nodes is written.

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

In the example below a wildcard block 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`. This example showcases how wildcards are interspliced with JavaScript templates.

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

3.2.3 Transforming code

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

A transformation template defines how the matches will be transformed after applicable code has been found. The transformation is a general template of the code once the match is replaced in the original AST. However, without transferring over the context from the match, this would be a template search and replace. Thus, 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 to know which wildcard match we are taking the context from, and where to place it in the transformation template.

The example below showcases a transformation that transforms a variable declaration from using `let` to use `const`. We do this by defining the name of the variable as a wildcard, and the expression as a wildcard. These are referenced in the transformation to transfer context from the match into the transformation template.

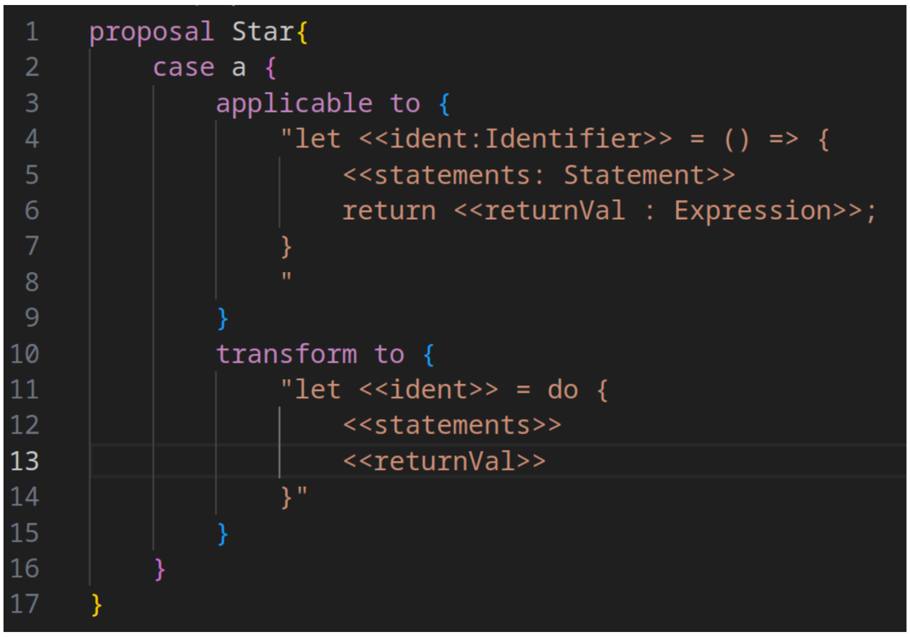
```
// Example applicable to template
applicable to {
    let <<variableName: Identifier>> = <<expr1: Expression>>;
}

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

3.2.4 Using JSTQL

JSTQL is designed to closely mimic the style of the examples required in the TC39 process [?]. We chose to design it this way to specifically make this tool fit the use-case of the committee. Since the idea behind this project is to gather early user feedback on syntactic proposals, the users of this kind of til is the committee themselves, as they are the ones that want user feedback.

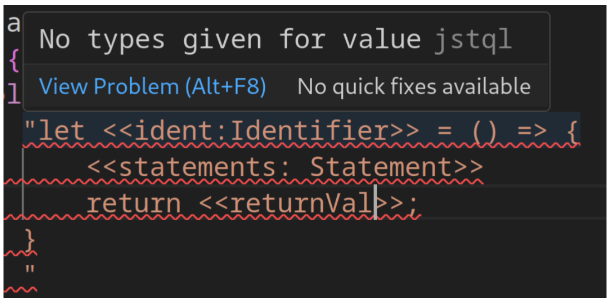
Writing a proposal definition in JSTQL is done with text, most domain-specific languages have some form of tooling to make the process of using the DSL simpler and more intuitive. JSTQL has an extension built for Visual Studio Code (see Figure ??). This extension supports auto-completion, error checking, and other common IDE features.



```
proposal Star{
    case a {
        applicable to {
            "let <<ident:Identifier>> = () => {
                <<statements: Statement>>
                return <<returnVal : Expression>>;
            }
            "
        }
        transform to {
            "let <<ident>> = do {
                <<statements>>
                <<returnVal>>
            }"
        }
    }
}
```

Figure 3.1: Writing JSTQL in Visual Studio Code with extension.

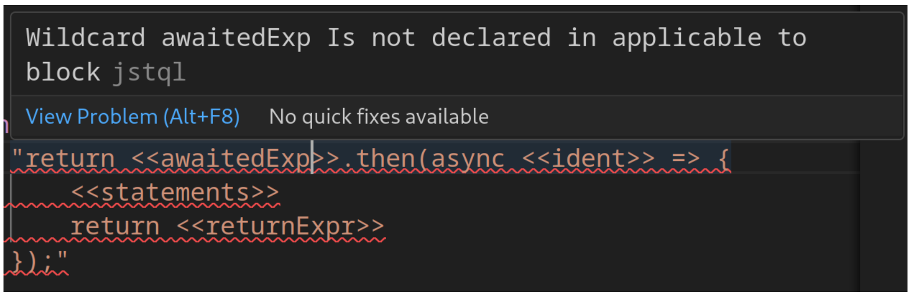
The language server used in this extension performs validation of the wildcards. This allows verification of wildcard declarations in applicable to, as shown in Figure ??. If a wildcard is declared with no types, an error will be reported.



```
No types given for value jstql
View Problem (Alt+F8)    No quick fixes available
"let <<ident:Identifier>> = () => {
    <<statements: Statement>>
    return <<returnVal>>;
}
"
```

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

The extension automatically uses wildcard declarations in `applicable to` in order to verify that all wildcards referenced in `transform to` are declared. If an undeclared wildcard is used, an error will be reported and the name of the undeclared wildcard will be displayed, as shown in Figure ??.



```
Wildcard awaitedExp Is not declared in applicable to
block jstql
View Problem (Alt+F8)    No quick fixes available
"return <<awaitedExp>>.then(async <<ident>> => {
    <<statements>>
    return <<returnExpr>>
});"
```

Figure 3.3: Error displayed with usage of undeclared wildcard.

3.3 Using the JSTQL with syntactic proposals

In this section, we present specifications of the proposals described in Section **??**. We will use these specifications to evaluate the tool created in this thesis. These specifications do not necessarily need to cover every single case where the proposal might be applicable, as they only have to be general enough to create some amount of examples that will give a representative number of matches when the transformations are applied to some relatively long user code.

This is because this tool is designed to be used by TC39 to gather feedback from user's on proposals during development. This use case means the specifications should be defined in a way that showcases the proposal. This also means it is important that the transformation is correct, as incorrect transformations might lead to bad feedback on a proposal.

3.3.1 “Pipeline” Proposal

This proposal is applicable to call expressions, and is aimed at improving code readability when performing deeply nested function calls.

```
proposal Pipeline {

    case SingleArgument {
        applicable to {
            "<<someFunctionIdent:Identifier ||
               ↪ MemberExpression>>(<<someFunctionParam:
               ↪ Expression>>);"
        }

        transform to {
            "<<someFunctionParam>> |> <<someFunctionIdent>>(%);"
        }
    }

    case TwoArgument{
        applicable to {
            "<<someFunctionIdent: Identifier ||
               ↪ MemberExpression>>(<<someFunctionParam:
               ↪ Expression>>, <<moreFunctionParam: Expression>>)"
        }
        transform to {
            "<<someFunctionParam>> |> <<someFunctionIdent>>(%,
               ↪ <<moreFunctionParam>>)"
        }
    }
}
```

Listing 3.2: Example of “Pipeline” proposal definition in JSTQL.

In the Listing ??, the first `case` definition `SingleArgument` will apply to any `CallExpression` with a single argument. We do not 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 that 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.

3.3.2 “Do Expression” Proposal

The “Do Expression” proposal [?] focuses on bringing expression-oriented programming to JavaScript.

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

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

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

Listing 3.3: Definition of Do Proposal in JSTQL.

In Listing ??, the specification of “Do Expression” proposal in JSTQL can be seen. It has two cases: the first case `arrowFunction` applies to code using an immediately invoked arrow function [? , 15.3] with a return value. The wildcard `statements` matches against one or more statements that are not of type `ReturnStatement`. The reason we limit the wildcard 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 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 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 transform an arrow function into a `do` expression.

The second case `unnamedFunction` follows the same principle as the first case, but is applied to immediately invoked unnamed functions, and produces the exact same output after the transformation as the first case. This is because immediately invoked unnamed functions are equivalent to arrow functions.

3.3.3 “Await to promise” imaginary proposal

```
proposal awaitToPromise{
    case single{
        applicable to {
            "let <<ident:Identifier>> = await <<awaitedExpr:
               ↪ Expression>>;
            <<statements: (Statement && !ReturnStatement &&
               ↪ !ContinueStatement &&!BreakStatement)+>>
            return <<returnExpr: Expression>>
            "
        }

        transform to{
            "return <<awaitedExpr>>.then(async <<ident>> => {
                <<statements>>
                return <<returnExpr>>
            });"
        }
    }
}
```

Listing 3.4: Definition of “Await To Promise” evaluation proposal in JSTQL.

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

The transformation definition has to use an `async` arrow function as argument for `then`, as there might be more await expressions contained within `statements`.

3.4 JSTQL-SH

In this thesis, we have also explored an alternative way of specifying syntactic proposals—JSTQL-SH[1]—which effectively uses JavaScript as a meta-language.

In this approach, proposal specifications are written as JavaScript objects. Each specification defines the following keys on the object:

- `prelude`, which is a sequence of JavaScript variable declarations, which are used to define wildcards.
- `applicableTo`, which is the template to perform matching. Unlike the JSTQL template specification, a corresponding JSTQL-SH specification is free of wildcards—for that purpose, the variables introduced in the `prelude` are used.
- `transformTo`, which is the template that defines the transformation. Similarly to the previous field, the value of this field is free of any wildcards; instead, a user can refer to the variables defined in `prelude` that represent wildcards.

This example below represents the first case of the "Pipeline" proposal specification (see Listing **??** for comparison).

```
{
    prelude: `
        let someFunctionIdent = "Identifier || MemberExpression";
        let someFunctionParam = "Expression";
        `,
    applicableTo: "someFunctionIdent(someFunctionParam);",
    transformTo: "someFunctionParam |> someFunctionIdent(%);"
}
```

JSTQL-SH provides an Application Programming Interface that exposes a function that takes a JavaScript object that represents a proposal specification and a string that represents the user code to which the proposal will be applied. This function then returns the transformed source code.

The main benefit of this alternative approach is that an extraction of wildcards from templates is not required. This means that a template is becomes a valid JavaScript code. Using only JavaScript to define proposal specifications could lower the barrier for use of our tool by the TC39 delegates and members of the JavaScript community, as being able to experiment with a new proposal now becomes a matter of using a library function.

[1]"SH" stands for "self-hosted", inspired by `http://udn.realityripple.com/docs/Mozilla/Projects/SpiderMonkey/Internals/self-hosting`.

Chapter 4

Implementation

In this chapter, the implementation of the tool utilizing the JSTQL and JSTQL-SH will be presented.[1] 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

As was presented in Section ??, there are two ways to specify a proposal: either using a custom domain-specific language JSTQL, or by using the corresponding JavaScript API. Figure ?? demonstrates the architecture of the implementation of these two approaches. In the figure, ellipse nodes represent data passed into the tool, and rectangular nodes represent specific components of the tool.

In the JSTQL approach (the "left-side" path in the figure), the initial step is to parse a proposal specification and then to extract the wildcard *declarations* and *references* from code templates. A corresponding step in the API-based approach (the "right-side" path) is to build the prelude, where the wildcard definitions are "extracted" from JavaScript code.

For both of the approaches, the second step (Section ??) is to parse wildcard type expressions used in the templates' specifications. After that, at step 3 (Section ??), Babel is used to parse and build abstract syntax trees for the `applicable to` templates and

[1]The source code for this implementation can be found `https://github.com/polsevev/JSTQL-JS-Transform`

the `transform to` templates in a proposal specification, and the user's code to which the proposal will be applied. At step 4 (Section ??), we process the abstract syntax trees produced by Babel and produce a custom tree data structure for simpler traversal. At step 5 (Section ??), we match the user's AST against the templates in the `applicable to` blocks. Once all matches have been found, we incorporate the wildcard matches into the `transform to` template at step 6 (Section ??), and insert it back into the users code. At this point, the AST of the user's code has been transformed, and the final step 7 (Section ??) then pretty-prints the transformed AST into a JavaScript source code.

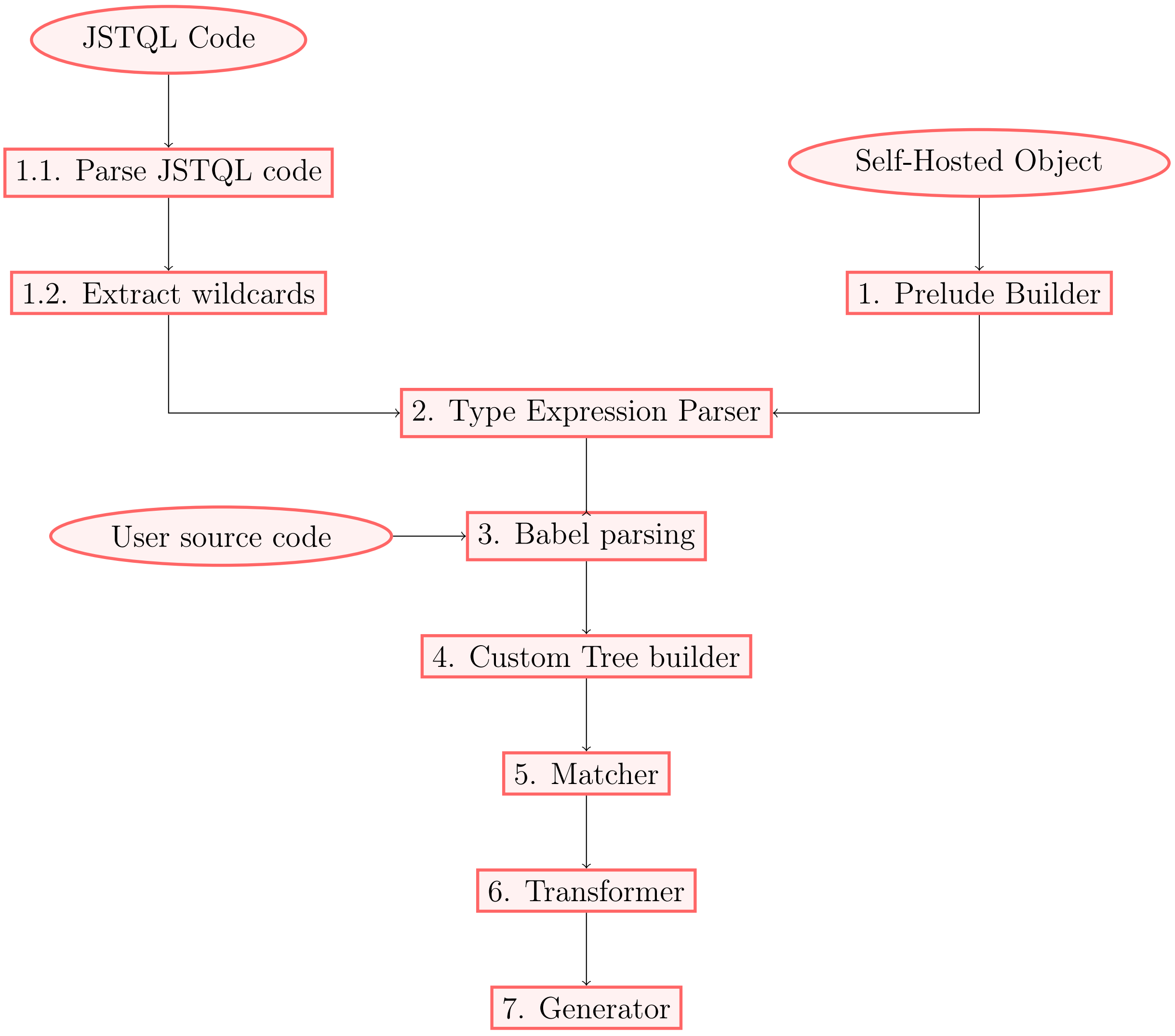


Figure 4.1: Overview of tool architecture

4.2 Parsing JSTQL using Langium

In this section, we describe the implementation of the parser for JSTQL. We start with outlining the language workbench which we used to generate a parser for JSTQL.

Langium [?] is a language workbench [?] that can be used to generate parsers for software languages, in addition to producing a tailored Integrated Development Environment for the language.

A parser generated by Langium produces abstract syntax trees which are TypeScript objects. These objects and their structure are used as definitions for the tool to do matching and transformation of user code.

To generate a parser, Langium requires a definition of a grammar. A grammar is a specification that describes syntax a valid programs in a language. The grammar for JSTQL describes the structure of JSTQL specifications. The starting symbol of the grammar represents valid specifications:

```
grammar Jstql

entry Model:
    (proposals+=Proposal)*;
```

In its turn, a proposal's specification includes its name and a specification of at least one *transformation case*.

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

A transformation case specification is comprised of a code template to match a JavaScript code to which the case is applicable, and a code template that specifies how a match should be transformed.

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

Case specifications are designed in this way in order to separate different transformation definitions within a single proposal.

An `applicable to` block specifies a JavaScript code template with wildcard declarations. This code template is represented in the grammar using the terminal symbol `STRING`, and will be thus parsed as a raw string of characters.

```
ApplicableTo:
    "applicable" "to" "{"
        apl_to_code=STRING
    "}";
```

The decision to use the STRING terminal, rather than a designated nonterminal symbol that would represent valid JavaScript programs with wildcards, is motivated by two reasons: (i) we separate parsing of the JSTQL specification structure (which is done by Langium) and parsing of JavaScript code (for which we use Babel[2]); and (ii) we use a custom processor of wildcards to enable reuse of such a processor for both JSTQL and JSTQL-SH[3].

A `transform to` block is specified in a similar manner:

```
TransformTo:
    "transform" "to" "{"
        transform_to_code=STRING
    "}";
```

Notwithstanding the fact that the code templates in `applicable to` and `transform to` blocks are treated as strings by Langium—and thus by the Visual Studio Extension for JSTQL generated by Langium—we perform validation of the wildcard declarations and references, as explained below.

Langium Validator

A Langium validator allows for further check to be applied to DSL code, a validator allows for the implementation of specific checks on specific parts of the code.

JSTQL does not allow empty wildcard type expression definitions in `applicable to` blocks. This is not defined within the grammar, and needs to be enforced with a validator. Concretely, we have implemented a specific `Validator` for the `Case` rule of the grammar. This means every time anything contained within a `Case` is updated, Langium will perform the validation step and report any errors. The validator implemented for our tool checks for the following errors: empty wildcard type expressions, undeclared wildcards in `transform to` block, and wildcards used multiple times in `transform to block`.

In the listing below is the validator, it performs checks on the `applicable to` block and `transform to` block of the `case`. If any errors are found it reports them with the function `accept`.

[2]See Sections ?? and ??.
[3]See Section ??.

```
export class JstqlValidator {
    validateWildcards(case_: Case, accept: ValidationAcceptor): void {
        try {
            let validationResultAplTo = validateWildcardAplTo(
                collectWildcard(case_.aplTo.apl_to_code.split(""))
            );
            if (validationResultAplTo.errors.length != 0) {
                accept("error", validationResultAplTo.errors.join("\n"), {
                    node: case_.aplTo,
                    property: "apl_to_code",
                });
            }

            let validationResultTraTo = validateWildcardTraTo(
                collectWildcard(case_.traTo.transform_to_code.split("")),
                validationResultAplTo.env
            );

            if (validationResultTraTo.length != 0) {
                accept("error", validationResultTraTo.join("\n"), {
                    node: case_.traTo,
                    property: "transform_to_code",
                });
            }
        } catch (e) {}
    }
}
```

Interfacing with Langium

To use the parser generated by Langium, we have give our tool a way to interface with Langium. To do this, we create a custom function that calls the generated parser on some JSTQL code and transforms the AST into a JavaScript object compatible with our tool.

4.3 Wildcard extraction and parsing

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.

Why not use Langium for wildcard extraction?

Langium supports creating a generator to output an artifact, which is some transformation applied to the AST built by the Langium parser. This would facilitate extraction of the wildcards, however this would make JSTQL-SH dependent on Langium. This is not preferred as that would mean both ways of defining a proposal are reliant on 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

To parse the templates in `applicable to` blocks and `transform to` blocks, we have to make the templates valid JavaScript. This is done by using a wildcard extractor 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 wildcard opening token is encountered, everything after that until the closing token is treated as a wildcard definition or reference and will parsed using the wildcard parser.

Once the wildcard is parsed, and we know it is valid, we insert the identifier into the JavaScript template where the wildcard would reside. This introduces a problem of *collisions* between the wildcard identifiers inserted and identifiers present in the users code. In order to avoid this, we prepend and append every identifier inserted in place of a wildcard with the sequence of characters `_$$_`.

In the Listing **??** the function used to extract the wildcards declarations can be seen. This function iterates through each character of `applicable to` template (line 2). When an opening token for a wildcard is encountered (line 3), we collect each character into a separate variable until the closing token is encountered. This separate variable is passed to the wildcard parser to create the type expression AST of the wildcard (lines 14-16). We insert the collision avoidance characters into the wildcard (line 18), and insert the identifier into `cleanedJS` (line 19).

```
 1 export function parseInternal(code: string): InternalParseResult {
 2      for (let i = 0; i < code.length; i++) {
 3         if (code[i] === "<" && code[i + 1] === "<") {
 4             // From now in we are inside of the DSL custom block
 5             flag = true;
 6             i += 1;
 7             continue;
 8         }
 9 
10         if (flag && code[i] === ">" && code[i + 1] === ">") {
11             // We encountered a closing tag
12             flag = false;
13             try{
14                 let wildcard = new WildcardParser(
15                     new WildcardTokenizer(temp).tokenize()
16                 ).parse();
17                 wildcard.identifier.name =
18                     "_$$_" + wildcard.identifier.name + "_$$_";
19                 cleanedJS += wildcard.identifier.name;
20 
21 
22                 prelude.push(wildcard);
23                 i += 1;
24                 temp = "";
25                 continue;
26             }
27             catch (e){
28                 // We probably encountered a bitshift operator, append
                      ↪ temp to cleanedJS
29             }
30 
31         }
32         if (flag) {
33             temp += code[i];
34         } else {
35             cleanedJS += code[i];
36         }
37     }
38     return { prelude, cleanedJS };
39 }
```

Listing 4.1: Extracting wildcard from template.

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

Our tokenizer takes the contents of wildcard block template and splits it into tokens. Given the straighforward grammar type expressions, there is no ambiguity is present with the tokens, thus making is easy to identify which character corresponds to which token. The tokenizer adds a *token type* to the tokens, this is later used by the parser to determine which nonterminal to use.

A recursive descent parser mimics the grammar of the language the parser is implemented for: we define functions for handling each of the nonterminals.

```
Wildcard:
    Identifier ":" MultipleMatch

MultipleMatch:
    GroupExpr "+"
    | TypeExpr

TypeExpr:
    BinaryExpr
    | UnaryExpr
    | PrimitiveExpr

BinaryExpr:
    TypeExpr { Operator TypeExpr }*

UnaryExpr:
    UnaryOperator TypeExpr

PrimitiveExpr:
    GroupExpr | Identifier

GroupExpr:
    "(" TypeExpr ")"
```

Listing 4.2: Grammar of type expressions

The grammar of the type expressions used by the wildcards can be seen in Figure **??**.

Building prelude in JSTQL-SH The self-hosted version JSTQL-SH also requires some form of parsing to prepare the internal DSL environment.

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-hand 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 strictly defined, we can expect every statement of the program to be a variable declaration; otherwise we 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` into the template.

4.4 Using Babel to parse

Allowing the tool to perform transformations of code requires the generation of an abstract syntax trees from the user's code, as well as the `applicable to` and `transform to` blocks. This means parsing JavaScript into an AST; to do this we use Babel [?].

The reason for choosing to use Babel is the fact that it supports very early-stage JavaScript language proposals. Babel's maintainers collaborate closely with the TC39 Committee in order to provide extensive support of experimental syntax [?] through its plugin system. This allows the parsing of JavaScript code that uses language features which are not yet part of the language standard.

Custom Tree Structure

The AST structure used by Babel does not suit traversing multiple trees at the same time, which is a requirement for matching. Therefore, based on Babel's AST, we produce our own custom tree structure that allows for simple traversal of multiple trees at once.

As can be seen in Figure ??, we use a recursive definition of a `TreeNode`, where a node's parent either exists or is `null` (the root), and a node can have any number of child elements. This definition allows for simple traversal both up and down the tree. This means traversing two trees at the same time can be done when searching for matches in the user's code.

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

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

Listing 4.3: Simple definition of a Tree structure in TypeScript

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

Visiting a node using the `enter()` function means we traversed from a parent node to its child node. When we then initialize the `TreeNode` of the current child, we add the parent previously visited as its parent node. Whenever leaving a node the function `exit()` is called, this means we are moving back up the tree, and we have to update what node was the *last* visited to keep track of the correct parent.

In the example below is the algorithm that transforms the Babel AST into our custom tree structure. We start by defining a variable `last` (line 1), this variable will keep track of the previous node we visited. When the visitor enters a new node, the function `enter` is called (line 3). This function creates a new node in our custom tree structure (lines 4-6), and sets its parent to the previous node visited (line 5). Once our new node has been created, we update `last` to point to our new node (line 8).

Every time we walk back up the tree, the function `exit` (line 10) is called. Whenever this happens, we have to update `last` such that it will always contain the parent of a node when we visit it (line 11).

```
 1 let last = 0;
 2 traverse(ast, {
 3         enter(path: any) {
 4             let node: TreeNode<t.Node> = new TreeNode<t.Node>(
 5                 last,
 6                 path.node as t.Node
 7             );
 8             last = node;
 9         },
10         exit(path: any) {
11                 last = last.parent;
12         },
13     });
14     if (first != null) {
15         return first;
16     }
```

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

4.5 Outline of transforming user code

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

Algorithm 1 An outline of the steps to perform the transformation. Here: A denotes the `applicable to` template with wildcards extracted, B denotes the `transform to` template with wildcards extracted, W denotes extracted wildcards, C denotes the abstract syntax tree of the applicable to template, D denotes the abstract syntax tree of the `transform to` template, E denotes the abstract syntax tree of the user's code, F denotes the `applicable to` template in our custom tree structure, G denotes the `transform to` template in our custom tree structure, H denotes the user code in our custom tree structure, J denotes an array of all the found matches, K denotes an array that contains all `transform to` templates with context from user code inserted, L denotes the abstract syntax tree of the transformed user code, and *SourceCode* is the transformed user code pretty-printed as JavaScript.

```
1:  $A, B, W \leftarrow extractWildcards()$
2:  $C, D, E \leftarrow babel.parse(A, B, UserCode)$
3:  $F, G, H \leftarrow Tree(C, D, E)$
4:  **if** $F.length > 1$ **then**
5:      $J \leftarrow multiMatcher(F, E, W)$
6:  **else**
7:      $J \leftarrow singleMatcher(F, E, W)$
8:  **end if**
9:  $K \leftarrow []$                                  ▷ Array of transformed code
10: **for each** $m$ **in** $J$ **do**
11:     $K.insert \leftarrow buildTransform(m, G, W)$
12: **end for**
13: $L \leftarrow insertTransformations(K)$
14: $SourceCode \leftarrow babel.generate(L)$
```

Each part of Algorithm **??** is a step to transform user code based on a proposal specification in our tool.

In the initial of the algorithm (line 1), the wildcards are extracted from the templates `applicable to` and `transform to`, and replaced by identifiers. The extracted wildcards are then parsed into ASTs using a parser built into the tool.

We parse the `applicable to` template, `transform to` template and the user's code into ASTs with Babel (line 2). These ASTs are immediately translated into our custom tree structure (line 3). This ensures simple traversal of multiple trees.

To decide which matching algorithm we apply, the length of the `applicable to` template is checked (line 5), if it is more than one statement long we use `multiMatcher` (line 5), if it is a single statement we use `singleMatcher` (line 7). These algorithms will find all matching parts of the user AST to the `applicable to` template.

We use these matches to prepare the `transform to` templates (lines 9-12). The AST nodes from the user code that was matched with a wildcard is inserted into the wildcard references present in the `transform to` template (line 11). All the transformed `transform to` templates are stored in a list(line 9, 11).

Once all transformations are prepared, we traverse the user AST (line 13), and insert the transformations where their corresponding match originated. The final step, is to generate JavaScript from the transformed AST (line 14).

4.6 Matching

This section discusses how we find matches in the users code; this is the step described in lines 4-8 of Listing **??**. We will discuss how individual nodes are compared, how the two traversal algorithms are implemented, and how matches are discovered using these algorithms.

4.6.1 Determining if AST nodes match

To determine if two nodes are a match, we need some method to compare AST nodes of the `applicable to` template to AST nodes of the user code. This step also has to take into account comparisons with wildcards and pass that information back to the AST matching algorithms.

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

NoMatch The nodes do not match.

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

MatchedWithWildcard The nodes are a match, and the node of `applicble to` is a wildcard.

MatchedWithPlussedWildcard The nodes are a match, and the node of `applicable to` is a wildcard with the Kleene plus.

To compare two AST nodes, we start by comparing their types, if the types are not the same, the result is `NoMatch`. If the types are the same, further checks are required.

Firstly we need to determine if the current AST node of `applicable to` is a wildcard. To do this, we check if its type is either an `Identifier` or an `ExpressionStatement` with an `Identifier` as its expression. During the wildcard extraction step, we replace the wildcard with an identifier. As a result, an identifier might be placed as a statement. When this occurs, the identifier will be wrapped inside an `ExpressionStatement`. If we encounter either of these two types, we must then check if the name of the identifier matches the name of a wildcard. If it does, we evaluate the type of the user AST node against the wildcards type expression.

In the example below we determine if the node of `applicable to` might be a wildcard

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

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

If we have determined the node of `applicable to` is not a wildcard, we then compare the two nodes to see if they match. For certain nodes, like `Identifier`, this involves explicitly checking specific fields, such as comparing the name field. For most nodes, however, we compare their types. Based on this comparison, the result will be either `Match` or `NoMatch`.

When comparing an AST node type against a wildcard type expression, we evaluate the wildcard type expression relative to the type of the node being compared. This evaluation employs the visitor pattern to traverse the AST, where each leaf node is checked against the type of the node being compared, yielding a Boolean result. All expressions are subsequently evaluated, with the values passed through the visitors until the entire expression is evaluated, producing a final result. If the evaluation result is `false`, we return `NoMatch`. If the evaluation result is `true`, we have to check if the wildcard uses a Kleene plus, if it does we return `MatchedWithPlussedWildcard`, if it does not we retrun `MatchedWithWildcard`.

4.6.2 Matching a single Expression/Statement template

In this section, we discuss how matching is performed when the `applicable to` template is a single expression/statement. This section will cover line 7 of Listing **??**.

Determining if we are currently matching with a template that is only a single expression/statement, we must verify that the program body of the template has the length of one, if this is the case we use the single length traversal algorithm.

There is a special case when the template is a single expression. When this is the case, the first node of the AST generated by `@babel/generate` [?] will be of type `ExpressionStatement`. This will miss many applicable parts of the users code, because expressions within other statements are not wrapped in an `ExpressionStatement`. This makes the template incompatible with otherwise applicable expressions. This means the statement has to be removed, and the search has to be done with the expression as the top node of the template.

Discovering Matches Recursively The matching algorithm used with single expression/statement templates is based on depth-first search to traverse the trees. The algorithm can be split into two steps. The first step is to start a new search on each child of the current node explored, and the second is the check the current node for a match.

The first step is to ensure we search for a match at all levels of the code AST. This is done by starting a new search on every child node of the code AST if the current node of the `applicable to` AST is the root node. This ensures we have explored a match at every level of the tree. As an added benefit of this approach is it ensures we have no partial matches, as we store a match only if it was called with the root node of the `applicable to` AST. This behaviour can be seen in the listing below.

```
if(aplTo.element === this.aplToRoot){
    // Start a search from root of aplTo on all child nodes
    for(let codeChild of code.children){
        let childMatch = singleMatcher(codeChild, aplTo);

        // If it is a match, we know it is a full match and store it.
        if(childMatch){
            this.matches.push(childMatch);
        }
    }
}
```

The second step is to compare the nodes of the current search. This means the current code AST node is compared against the current `applicable to` AST node. Based on this comparison, different steps must be performed, these steps can be seen below.

NoMatch: If a comparison between the nodes return a `NoMatch` result, we perform an early return of undefined, as no match was discovered.

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

MatchedWithWildcard: When a comparison results in a wildcard match, we immediately pair the current code node with the template wildcard and return early. This is possible because, once a wildcard matches, the child nodes are irrelevant and will be included in the transformation regardless.

MatchedWithPlussedWildcard: This is a special case for a wildcard match. When a match occurs against a wildcard with a Kleene plus we do the same as `MatchedWithWildcard`, but give a different comparison result as this necessitates a special traversal of the current nodes siblings.

A comparison result of `Matched` means the two nodes match, but the `applicable to` node is not a wildcard. If this is the case, we perform a search on each child nodes of `applicable to` AST and the user AST. This can be seen in Listing **??**.

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

In the Listing **??** below, we search the children of a comparison that returned the result `Match`. For this, we use a two pointer technique with `codeI` and `aplToI` (lines 1,2). This search continues until one of the pointers reaches the end of the list of children for its respective node (line 4). If any of the child nodes to not return a match the entire match is discarded (lines 8-10). We prepare the paired tree by appending the current child search to the parent pair (line 14,15). We handle the special case with a Kleene plus (line 18), by continuing the search with the same `aplToI` pointer while incrementing `codeI` (lines 19-22). As long as the result is `MatchedWithPlussedWildcard` we add the node matched with the wildcard to the pair of matches, meaning the pair will contain multiple nodes from the user AST matched with the same wildcard (line 28). If the result is not `MatchedWithPlussedWildcard`, we decrement `codeI`, stop the comparisons against the wildcard, and continue searching all the child nodes as normal(lines 23-26). When one of the child lists is completely searched, we check if it is a full match of all the child nodes of the current code AST parent by verifying that we reached the end of the code AST children (lines 39-41). Once all these searches have been completed, and we confirm a `Match`, we return the paired tree structure along with match result.

```
let codeI = 0;
let aplToI = 0;

while (aplToI < aplTo.children.length && codeI < code.children.length){
    let [pairedChild, childResult] =
      ↪ singleMatcher(code.children[codeI], aplTo.children[aplToI]);

    // If a child does not match, the entire match is discarded
    if(childResult === NoMatch){
        return [undefined, NoMatch];
    }


    // Add the match to the current Paired Tree structure
    pairedChild.parent = currentPair;
    currentPair.children.push(pairedChild);

    // Special case for Kleene plus wildcard match
    if(childResult === MatchedWithPlussedWildcard){
        codeI += 1;
        while(codeI < code.children.length){
            let [nextChild, plusChildResult] =
              ↪ singleMatcher(code.children[codeI],
              ↪ aplTo.children[aplToI]);

            if(plusChildResult !== MatchedWithPlussedWildcard){
                codeI -= 1;
                break;
            }

            pairedChild.element.codeNode.push(
                ...nextChild.element.codeNode);

            codeI += 1;
        }
    }

    codeI += 1;
    aplToi += 1;
}

if(codeI !== code.children.length){
    return [undefined, NoMatch]
}

return [currentPair, Match];
```
Listing 4.4: Pseudocode of child node matching

4.6.3 Matching multiple statements

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

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

```
multiStatementMatcher(code, aplTo) {
    if (
        code.element.type === "Program" ||
        code.element.type === "BlockStatement"
    ) {
        matchMultiHead(code.children, aplTo.children);
    }

    for (let code_child of code.children) {
        multiStatementMatcher(code_child, aplTo);
    }
}
```

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

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

Output of the matcher

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

```
interface PairedNode{
    codeNode: t.Node[],
    aplToNode: t.Node
}
```

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

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

4.7 Transforming

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

An important note is we have 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.

Building the transformation

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

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

```
function extractWildcardPairs(match: Match): Map<string, t.Node[]> {
    let map: Map<string, t.Node[]> = new Map();

    function recursiveSearch(node: TreeNode<PairedNodes>) {
        let name: null | string = null;
        if (node.element.aplToNode.type === "Identifier") {
            name = node.element.aplToNode.name;
        } else if (
           // Node is ExpressionStatement with Identifier
        ) {
            name = node.element.aplToNode.expression.name;
        }

        if (name) {
            // Store in the map
            map.set(name, node.element.codeNode);
        }
        // Recursively search the child nodes
        for (let child of node.children) {
            recursiveSearch(child);
        }
    }
    // Start the initial search
    for (let stmt of match.statements) {
        recursiveSearch(stmt);
    }
    return map;
}
```

Listing 4.5: Extracting wildcard from match

Once the full map of all wildcards has been built, we have to insert the node matched with the wildcard into the `transform to` template. To do this, we traverse the template with `@babel/traverse` [?], as this provides us with a powerful API for modifying the AST. `@babel/traverse` allows us to define visitors, that are executed when traversing specific types of AST nodes. In this traversal, we define a visitor for `Identifier`, and a visitor for `ExpressionStatement`.

When we visit a node with these visitors, we check if that nodes name is in the map of wildcards. If the name of the identifier is a key in the wildcard map, we replace the identifier with the value in the map, which is AST nodes from the user's code that matched with that wildcard. See Listing **??**

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

Listing 4.6: Traversing `transform to` AST and inserting user context

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

Inserting the template into the AST

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

```
transformedTransformTo.set(
    match.statements[0].element.codeNode[0],
    [
        transformMatchFaster(wildcardMatches, traToWithWildcards),
        match,
    ]
);
```

We now traverse the AST generated from the users code with `@babel/traverse`. In this case we cannot use a specific visitor, and therefore we use a generic visitor that applies to every node type of the AST. If the current node we are visiting is a key to the map of transformations created earlier, we know we have to insert the transformed code. This is done similarly to before where we use `replaceWithMultiple`.

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

```
 1 traverse(codeAST, {
 2         enter(path) {
 3             if (transformedTransformTo.has(path.node)) {
 4                 let [traToWithWildcards, match] =
 5                     transformedTransformTo.get(path.node) as [
 6                         t.File,
 7                         Match
 8                     ];
 9                 path.replaceWithMultiple(
10                     traToWithWildcards.program.body);
11 
12                 let siblings = path.getAllNextSiblings();
13 
14                 // For multi line applicable to
15                 for (let i = 0; i < match.statements.length - 1; i++) {
16                     siblings[i].remove();
17                 }
18 
19                 // When we have matched top statements with +, we
                    ↪ might have to remove more siblings
20                 for (let matchStmt of match.statements) {
21                     for (let codeStmt of matchStmt.element
22                         .codeNode) {
23                         let siblingnodes = siblings.map((a) => a.node);
24                         if (siblingnodes.includes(codeStmt)) {
25                             let index = siblingnodes.indexOf(codeStmt);
26                             siblings[index].remove();
27                         }
28                     }
29                 }
30             }
31         },
32     });
```

Listing 4.7: Inserting transformed matches into user code

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

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

4.7.1 Generating source code from transformed AST

To generate JavaScript from the transformed AST created by this tool, we use a JavaScript library titled `@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 present the results of running each of the proposals discussed in this thesis on large-scale JavaScript projects.

To evaluate this tool on existing JavaScript codebases, we have collected JavaScript projects from Github containing many or large JavaScript files.

Each case study was evaluated by running this tool on every `.js`-file in the repository, and 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 [?] without errors.

We describe below our results and observations on using our tool on the codebases of various large-scale projects that use JavaScript.

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

Proposal	Matches found	Files with matches	Files processed
“Pipeline”	242079	1912	3340
“Do Expression”	229	37	3340
Await to Promise	8	7	3340

Table 5.1: Evaluation with Next.js source code

Three.js [?] is a library for 3D rendering in JavaScript. It is written purely in JavaScript and uses GPU for 3D calculations.

Proposal	Matches found	Files with matches	Files searched
Pipeline	84803	1117	1384
"Do Expression"	248	36	1384
Await to Promise	13	7	1384

Table 5.2: Evaluation with Three.js source code

React [?] is a graphical user interface library for JavaScript, which 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.

Proposal	Matches found	Files with matches	Files searched
"Pipeline"	16353	1266	2051
"Do Expression"	0	0	2051
Await to Promise	30	13	2051

Table 5.3: Evaluation with React source code

Bootstrap [?] is a front-end framework used for creating responsive and mobile-first websites, and it comes with a variety of built-in components. This library is a good evaluation point for this thesis as it is written in "vanilla" JavaScript.

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

Table 5.4: Evaluation with Bootstrap source code

Atom [?] is a text editor developed in JavaScript.

Proposal	Matches found	Files with matches	Files searched
"Pipeline"	40606	361	401
"Do Expression"	3	3	401
Await to Promise	12	7	401

Table 5.5: Evaluation with Atom source code

The "Pipeline" proposal is applicable to most files: the reason for this is that call expressions are widely used when writing JavaScript code. Our tool found matches in most files that Babel [?] managed to parse, and with manual evaluation transformations were performed correctly.

The "Do Expression" proposal is not as "applicable" as the "Pipeline" proposal: this means that the amount of transformed code this specification in JSTQL will be able to perform is expected and proven to be lower. This is because the proposal introduces an entirely new way of writing expression-oriented code in JavaScript. If the code has not used the current way of writing expression-oriented in JavaScript, JSTQL is limited in the amount of transformations it can perform. Nevertheless, our tool is able to identify matches where it is applicable, and by manual verification transformations are correct.

The imaginary "Await to promise" proposal also has an "expected" number of matches; however, we do not evaluate this proposal since it is not an official TC39 proposal.

Our tool demonstrates its capability to perform searches on large codebases, to identify applicable code for proposals, and to transform the code. As can be seen from the tables above, some of the proposals found zero matches when evaluated on some of these codebases. This is due to the fact that the developers of these projects have not used the language construct the proposal is targeting. Because of this, no transformations can be performed. This is especially apparent with the "Do Expression" proposal, but also with the "Await to Promise" imaginary proposal. This means that our tool's ability to perform transformations depends on how widespread the adoption of the language construct targeted in a proposal is. We can hypothesize that the amount of matches reflects the "impact" that design decisions made by the TC39 committee might have on established JavaScript projects and codebases.

We give examples of some of the transformations performed on these codebases in Appendix **??**.

Chapter 6

Related Work

In this chapter, we discuss various techniques and languages for code querying, present approaches to tree manipulation and transformation, and describe several JavaScript parsers. We also discuss aspect-oriented programming and model-driven language engineering.

6.1 Source code query languages

To allow for simple analysis and refactoring of code, there exist many query languages designed to query source code. These languages use various techniques to allow for querying code based on specific paradigms (such as: logical queries, declarative queries, SQL-like queries, etc.).

6.1.1 CodeQL

CodeQL [?] is an object-oriented query language, previously known as *.QL*. CodeQL is used to semantically analyze code to discover vulnerabilities [?]. The language is inspired [?] by SQL [?], Datalog [?], Eindhoven Quantifier Notation [?], and classes are predicates [?].

An example [?] of how queries are written in CodeQL is as follows.

```
from Class c
where c.declaresMethod("equals") and
    not(c.declaresMethod("hashCode")) and
    c.fromSource()
select c.getPackage(), c
```

This query will find all class that have method `equals`, but do not have method `hashCode`.

As can be seen from this example, the SQL-like syntax of writing queries in CodeQL is substantially different from JSTQL, which aims at a more declarative syntax. This makes the writing experience of the two languages very different: writing CodeQL queries are similar to querying a database, while queries written in JSTQL are similar to defining an example of the structure one wishes to search for.

6.1.2 PMD XPath

PMD XPath is a language for Java source code querying, This language supports querying of all Java constructs [?]. The reason it has this wide support is due to it constructing the entire codebase's AST in XML format, and then performing the query on the corresponding XML. These queries are performed using XPath expressions that define matching on XML trees. This makes the query language versatile for static code analysis, and it is used in the *PMD* static code analysis tool [?].

An example [?] PMD XPath queries are as follows.

```
//VariableId[@Name = "bill"]
//VariableId[@Name = "bill" and ../../Type[@TypeImage = "short"]]
```

This query can be applied, for example, to the following Java code [?]:

```
public class KeepingItSerious{
    Delegator bill; // FieldDeclaration

    public void method(){
        short bill; // LocalVariableDeclaration
    }
}
```

If we execute the queries on this code, the first query will match against the field declaration `Delegator bill` and `short bill`, while the second query will only return `short bill`. The reason the second limits the search is that we define the type of the declaration.

JSTQL uses JavaScript code *templates* to specify queries; this supposedly makes writing such queries simpler for users as they write JavaScript. In its turn, PMD XPath uses XPath expressions to perform define structural queries that is quite verbose, and requires extended knowledge of the AST that is currently being queried.

6.1.3 XSL Transformations

XSLT [?] is a language for performing transformations of XML documents, either to other XML documents, or to different formats altogether (such as HTML or plain text).

XSLT is part of Extensible Stylesheets Language family of programs. The XSL language is expressed in the form of a stylesheet [? , Sect. 1.1], whose syntax is defined in XML. This language uses a template based approach to define matches on specific patterns in the source to find sections to transform. These transformations are defined by a transformation declaration that describes how the output of the match should look.

The example XML document represents a program, where each node `variable` has an attribute `name`.

```xml
<program>
    <variable name="a"/>
    <variable name="b"/>
    <variable name="c"/>
</program>
```

To transform the example above, we define a transformation in XSLT seen below. This transformation contains two match templates; the first template matches nodes `program`, this template copies the node in the transformation with `xsl:copy` and applies the second transformation to all child nodes. The second transformation matches element `person`, it defines a transformation that changes node from `variable` to `const`.

```xml
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="/program">
        <xsl:copy>
            <xsl:apply-templates select="variable"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="variable">
        <const name="{@name}"/>
    </xsl:template>
</xsl:stylesheet>
```

The result of running the XSLT transformation above on the XML we defined is shown below.

```xml
<program>
   <const name="a"/>
   <const name="b"/>
   <const name="c"/>
</program>
```

Though XSLT defines matching in a manner similar to JSTQL, its approach to define transformations is different: JSTQL allows the user to specify a code fragment interspliced with wildcards, while XSLT requires specifying a transformation (written in a functional style). Moreover, JSTQL's implementation is tailored for the use by the TC39 committee, while XSLT's expressive power allows specifying arbitrary complex transformations of tree-like data structures.

6.1.4 Jackpot

Jackpot [?] (also known as *Java Declarative Hints Language*) is a query language that uses declarative patterns to define source code queries: these queries are used in conjunction with multiple rewrite definitions. The language is used in the Apache Netbeans [?] suite of tools to allow for declarative refactoring of code.

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

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

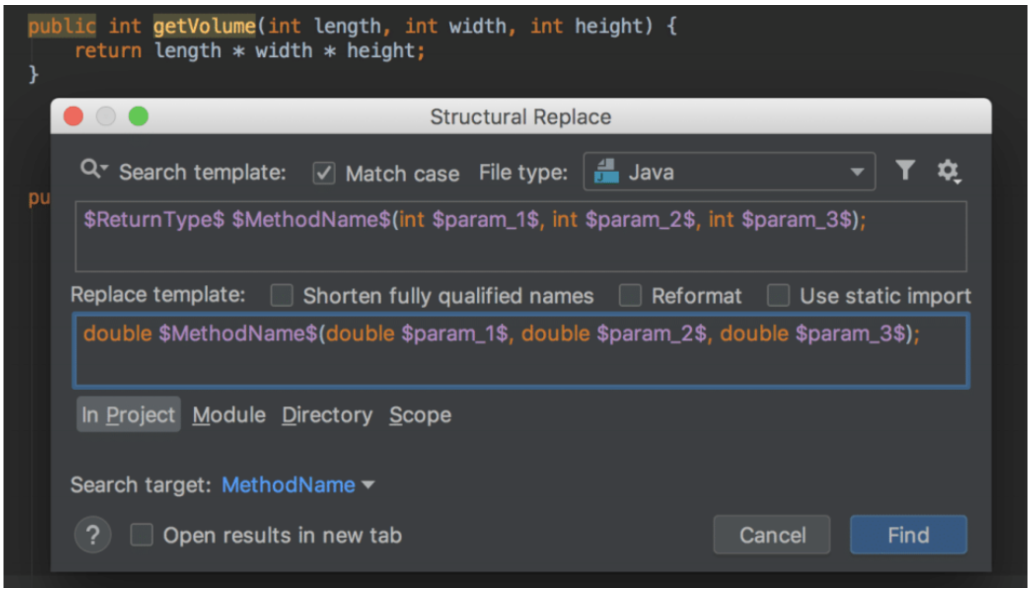
Jackpot is quite similar to JSTQL, as both languages define queries by using similar structure. In Jackpot, one defines a *pattern*, and then every match of that pattern can be re-written to a *fix-pattern*. Each fix-pattern can have a condition attached to it. This is quite similar to the *applicable to* and *transform to* sections of JSTQL. Jackpot also supports a feature which is similar to the wildcards in JSTQL—one can define variables in the *pattern* definition and transfer them over to the *fix-pattern* definition. In constant to JSTQL, wildcard type restrictions and notation for matching more than one AST node are not supported in Jackpot.

6.2 IntelliJ structural search

JetBrains IntelliJ-based Integrated Development Environments have a feature that allows for structural search and replace [?]. This feature is intended for large code bases where a developer wishes to perform a search and replace based on syntax and semantics, and not a (regular) text based search and replace.

When doing structural search in IntelliJ-based IDEs, 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.

In the figure below we perform a structured search for a method declaration with three parameters of type `int`, and replace it with a method declaration where all parameters are of type `double` and the return type is `double`.



```
public int getVolume(int length, int width, int height) {
    return length * width * height;
}
```

Search template: ☑ Match case File type: Java

```
$ReturnType$ $MethodName$(int $param_1$, int $param_2$, int $param_3$);
```

Replace template: ☐ Shorten fully qualified names ☐ Reformat ☐ Use static import

```
double $MethodName$(double $param_1$, double $param_2$, double $param_3$);
```

Search target: MethodName

Figure 6.1: Example of Intellij structural search and replace

This tool is interactive, and every match is showcased in the *Find* tool. In this tool, a 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 wishes so, they do not have to verify each match and can replace all matches at once.

IntelliJ structured search and replace and JSTQL have similarities: they both are template-based. In both approaches, templates can contain variables and wildcards to allow for matching against arbitrary code. Both tools also support matching multiple code parts against a single variable or a wildcard. A core difference between the two tools 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 IntelliJ.

6.3 JavaScript parsers

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

Speedy Web Compiler

Speedy Web Compiler [?] (SWC) is a library created for parsing and compiling JavaScript and other dialects (such as JSX and TypeScript). It is written in Rust and is known for its improved performance. SWC is used by large organizations creating applications and tooling for the web platform.

Speedy Web Compiler supports various features, 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* (that makes the bundle size of a project smaller, transforming for use with WebAssembly), as well as *custom plugins* (to change the specification of the languages parsed by SWC).

SWC was considered to be used in this project, however due to SWC only supporting proposals when they reach stage 3, it was not possible to use this parser.

Acorn

Acorn [?] is parser written in JavaScript to parse JavaScript and related languages. Acorn focuses on plugin support to support extending and redefinition of how its internal parser works. Acorn focuses on being a small and performant JavaScript parser, and has a custom tree traversal library Acorn Walk. Babel is originally a fork of Acorn, and while Babel has since had a full rewrite, Babel is still heavily based on Acorn [?].

Acorn was considered as a parser in this project, however it does not have the same wide community as Babel, and does not have the same recommendation from TC39 as Babel does [?]. Even though it supports plugins and the plugin system is powerful, there does not exist the same amount of pre-made plugins for early stage proposals as Babel has.

6.4 Model-to-Model Transformations

Model-to-Model transformations are an integral part of model-driven engineering (MDE), which is a methodology that focuses on the creation and modification of abstract models rather than focusing on executable code [?]. This methodology provides a higher-level approach to developing large software systems.

The process of performing a model-to-model transformation is to convert one model into another, while preserving or adapting its underlying semantics and structure [?]. This is usually done by traversing its structure, and extracting data and transforming its format to fit the model it should be transformed into. This allows a model described within one domain to be transformed into another automatically.

6.5 Aspect-Oriented Programming

Aspect-Oriented Programming [?] (AOP) s a programming paradigm that enables modularity by allowing for a high degree of separation of concerns, specifically focusing on cross-cutting concerns. Cross-cutting concerns are aspects of a software program or a system that have an effect at multiple levels, cutting across the main functional requirements. Such aspects are often related to security, logging, or error handling, but could be any concern that are shared across an application.

In AOP, one creates an *aspect*, which is a module that contains some cross-cutting concern the developer wants to achieve. An aspect contains *advices*, which are the specific code fragments executed when certain conditions of the program are met (for example, a *before advice* is executed before a method executes, an *after advice* is executed after a method regardless of the methods outcome, an *around advice* 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 (these can be at specific methods or when specific constructors are called, and so on).

One can see a similarity between JSTQL and aspect-oriented programming: to define where *pointcuts* are placed, we have to define some structure and the AOP weaver has to search the code execution for events triggering the pointcut and run the advice defined within the aspect of that given pointcut.

Chapter 7

Conclusion and Future Work

In this thesis, we have explored an approach to define transformations of JavaScript code based on formal specifications of syntactic proposals. The goal of such transformations is to gather (early) feedback for (contentious) syntactic ECMAScript language proposals discussed by the TC39 committee. Our tool opens a possibility for the users to "preview" proposals on their own codebases: it can be conjectured that users' familiarity with the code shall improve the quality of feedback.

The work presented in this thesis is an initial step in developing a language workbench-like tool for supporting design of widely adopted programming languages. While this thesis adequately implements the machinery of the core of such a tool, future work is required. A major next step is to **integrate a feedback gathering mechanism in an IDE**. This shall give users a way to apply proposals to fragments of their code *and* to be able to give feedback on every such application. This could be implemented, for example, using a rating scale (e.g., Likert scale) to quantify user's preferences. The user would also be able to submit their code (in an obfuscated form) directly to the TC39 committee.

We have also identified several directions on how the **expressiveness of JSTQL can be improved.** For example, *parameterized specifications* can be introduced to enable reuse of (parts of) proposal specifications. Another example is to *support a richer syntax for wildcards*—this would allow for more power matching and transformations of the AST structures. Currently, our tool relies heavily on abstract syntax trees produced by Babel. While this can be considered as an advantage for the TC39's use case, introducing **support for arbitrary JavaScript parsers** can be beneficial.

Ultimately, **supporting other programming languages** in our tool could help in performing corpus analysis when designing new features for both ECMAScript and those other languages. In addition, this could enable exploring *co-evolution* of programming languages.

Appendix A

TypeScript types of wildcard type expressions

```
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

Appendix B

Examples of transformations performed in Evaluation

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

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

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

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

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

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

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

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

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

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

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

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

```
if (repo && repo.onDidDestroy) {
    repo.onDidDestroy(() =>
    this.repositoryPromisesByPath.delete(pathForDirectory)
    );
}
```

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

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

```
Lensflare.Geometry = do {
    const geometry = new BufferGeometry();
    const float32Array = new Float32Array([
        -1, -1, 0, 0, 0, 1, -1, 0, 1, 0, 1, 1, 0, 1, 1, -1, 1, 0, 0, 1,
    ]);
    const interleavedBuffer = new InterleavedBuffer(float32Array, 5);
    geometry.setIndex([0, 1, 2, 0, 2, 3]);
    geometry.setAttribute(
        "position",
        new InterleavedBufferAttribute(interleavedBuffer, 3, 0, false)
    );
    geometry.setAttribute(
        "uv",
        new InterleavedBufferAttribute(interleavedBuffer, 2, 3, false)
    );
    geometry;
};
```

```
Lensflare.Geometry = do {
    const geometry = new BufferGeometry();
    const float32Array = new Float32Array([
        -1, -1, 0, 0, 0, 1, -1, 0, 1, 0, 1, 1, 0, 1, 1, -1, 1, 0, 0, 1,
    ]);
    const interleavedBuffer = new InterleavedBuffer(float32Array, 5);
    geometry.setIndex([0, 1, 2, 0, 2, 3]);
    geometry.setAttribute(
        "position",
        new InterleavedBufferAttribute(interleavedBuffer, 3, 0, false)
    );
    geometry.setAttribute(
        "uv",
        new InterleavedBufferAttribute(interleavedBuffer, 2, 3, false)
    );
    geometry;
};
```

"Do expression" transformation, taken from `three.js/examples/jsm/objects/Lensflare.js`

```
const panLeft = (function () {
    const v = new Vector3();

    return function panLeft(distance, objectMatrix) {
        v.setFromMatrixColumn(objectMatrix, 0); // get X column of
            ↪ objectMatrix
        v.multiplyScalar(-distance);

        panOffset.add(v);
    };
})();

const panUp = (function () {
    const v = new Vector3();

    return function panUp(distance, objectMatrix) {
        if (scope.screenSpacePanning === true) {
            v.setFromMatrixColumn(objectMatrix, 1);
        } else {
            v.setFromMatrixColumn(objectMatrix, 0);
            v.crossVectors(scope.object.up, v);
        }

        v.multiplyScalar(distance);

        panOffset.add(v);
    };
})();
```

```
const panLeft = do {
    const v = new Vector3();
    function panLeft(distance, objectMatrix) {
        v.setFromMatrixColumn(objectMatrix, 0); // get X column of
            ↪ objectMatrix
        v.multiplyScalar(-distance);
        panOffset.add(v);
    }
};
const panUp = do {
    const v = new Vector3();
    function panUp(distance, objectMatrix) {
        if (scope.screenSpacePanning === true) {
            v.setFromMatrixColumn(objectMatrix, 1);
        } else {
            v.setFromMatrixColumn(objectMatrix, 0);
            v.crossVectors(scope.object.up, v);
        }
        v.multiplyScalar(distance);
        panOffset.add(v);
    }
};
```

"Do expression" transformation, taken from `three.js/examples/jsm/objects/Lensflare.js`

```
async loadAsync(url, onProgress) {
    const scope = this;
    const path =
        this.path === "" ? LoaderUtils.extractUrlBase(url) : this.path;
    this.resourcePath = this.resourcePath || path;
    const loader = new FileLoader(this.manager);
    loader.setPath(this.path);
    loader.setRequestHeader(this.requestHeader);
    loader.setWithCredentials(this.withCredentials);
    return loader.loadAsync(url, onProgress).then(async (text) => {
        const json = JSON.parse(text);
        const metadata = json.metadata;
        if (
            metadata === undefined ||
            metadata.type === undefined ||
            metadata.type.toLowerCase() === "geometry"
        ) {
            throw new Error("THREE.ObjectLoader: Can't load " + url);
        }
        return await scope.parseAsync(json);
    });
}
```

```
async loadAsync(url, onProgress) {
    const scope = this;

    const path = this.path === "" ? LoaderUtils.extractUrlBase(url) :
       ↪ this.path;
    this.resourcePath = this.resourcePath || path;

    const loader = new FileLoader(this.manager);
    loader.setPath(this.path);
    loader.setRequestHeader(this.requestHeader);
    loader.setWithCredentials(this.withCredentials);

    const text = await loader.loadAsync(url, onProgress);

    const json = JSON.parse(text);

    const metadata = json.metadata;

    if (
        metadata === undefined ||
        metadata.type === undefined ||
        metadata.type.toLowerCase() === "geometry"
    ) {
        throw new Error("THREE.ObjectLoader: Can't load " + url);
    }

    return await scope.parseAsync(json);
}
```

"Await to promise" transformation, taken from `three.js/src/loaders/ObjectLoader.js`