

UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

Title of your master thesis

Author: Your name

Supervisors: Name of supervisors



UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2024

Abstract

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

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

Acknowledgements

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

Your name
Monday 20th May, 2024

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Figures	1
2	Background	2
2.1	Proposals	2
3	Collecting User Feedback for Syntactic Proposals	3
3.1	The core idea	3
3.1.1	Applying a proposal	4
3.2	Applicable proposals	4
3.2.1	Syntactic Proposals	5
3.2.2	Simple example of a syntactic proposal	5
3.2.3	[7]Discard Bindings	5
3.2.4	Pipeline Proposal	8
3.2.5	Do proposal	10
3.2.6	Await to Promise	10
3.3	Searching user code for applicable snippets	11
3.3.1	JSTQL	11
3.3.2	Transforming	12
3.3.3	Structure of JSTQL	13
3.4	Using the JSTQL with an actual syntactic proposal	15
3.4.1	Pipeline Proposal	15
3.4.2	Do Proposal	16
3.4.3	Await to Promises evaluation proposal	16
4	Implementation	18
4.1	Architecture	18
4.2	Parsing JSTQL using Langium	18
4.2.1	Langium	20
4.3	Pre-parsing	22
4.4	Using Babel to parse	25
4.5	Matching	27
4.6	Transforming	31
	Bibliography	33

List of Figures

4.1	Tool architecture	19
-----	-----------------------------	----

List of Tables

Listings

3.1	Example of imaginary proposal optional let to int for declaring numerical literal variables	5
3.2	Example of unpacking Object	6
3.3	Example of unpacking Array	6
3.4	Example discard binding with variable discard	6
3.5	Example Object binding and assignment pattern	6
3.6	Example Array binding and assignment pattern. It is not clear to the reader that in line 8 we are consuming 2 or 3 elements of the iterator. In the example on line 13 we see that is it more explicit how many elements of the iterator is consumed	6
3.7	Example discard binding with function parameters. This avoids needlessly naming parameters of a callback function that will remain unused.	7
3.8	Grammar of Discard Binding	7
3.9	Example of section containing the pipeline proposal	13
3.10	Example of pair section	13
3.11	Example of applicable to section	14
3.12	Example of transform to section	14
3.13	JSTQL definition of a proposal	14
3.14	Example of Pipeline Proposal definition in JSTQL	15
3.15	Definition of Do Proposal in JSTQL	16
3.16	Definition of Await to Promise evaluation proposal in JSTQL	17
4.1	Definition of JSTQL in Langium	20
4.2	Grammar of type expressions	24
4.3	Simple definition of a Tree structure in TypeScript	26
A.1	TypesScript types of Type Expression AST	34

Chapter 1

Introduction

Intro goes here

1.1 Background

1.1.1 Figures

Chapter 2

Background

2.1 Proposals

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

Chapter 3

Collecting User Feedback for Syntactic Proposals

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

3.1 The core idea

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

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

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

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

3.1.1 Applying a proposal

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

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

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

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

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

3.2 Applicable proposals

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

3.2.1 Syntactic Proposals

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

3.2.2 Simple example of a syntactic proposal

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

This proposal will look something like this:

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

Listing 3.1: Example of imaginary proposal **optional let to int for declaring numerical literal variables**

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

3.2.3 [7]Discard Bindings

The proposal Discard Bindings is classified as a Syntactic Proposal, as it contains no change to the semantics of EcmaScript. This proposal is created to allow for discarding objects when using the feature of unpacking objects/arrays on the left side of an assignment. The whole idea of this proposal is to avoid declaring unused temporary variables.

Unpacking when doing an assignment refers to assigning internal fields of an object/array directly in the assignment rather than using a temporary variable. See 3.2 for an example of unpacking an object and 3.3.

```

1 // previous
2 let temp = { a:1, b:2, c:3, d:4 };
3 let a = temp.a;
4 let b = temp.b;
5
6 // unpacking
7 let {a,b ...rest} = { a:1, b:2, c:3, d:4 };
8 rest; // { c:3, d:4 }

```

Listing 3.2: Example of unpacking Object

```

1 // previous
2 let tempArr = [ 0, 2, 3, 4 ];
3 let a = tempArr[0]; // 0
4 let b = tempArr[1] // 2
5
6 //unpacking
7 let [a, b, _1, _2] = [ 0, 2, 3, 4 ]; // a = 0, b = 2, _1 = 3, _2 = 4

```

Listing 3.3: Example of unpacking Array

In EcmaScripts current form, it is required to assign every part of an unpacked object/array to some identifier. The current status quo is to use `_` as a sign it is meant to be discarded. This proposal suggests a specific keyword *void* to be used as a signifier whatever is at that location should be discarded.

This feature is present in other languages, such as Rust wildcards, Python wildcards and C# using statement and discards. In most of these other languages, the concept of discard is a single `_`. In EcmaScript the `_` token is a valid identifier, therefore this proposal suggests the use of the keyword *void*. This keyword is already reserved as part of function definitions where a function is meant to have no return value.

This proposal allows for the *void* keyword to be used in a variety of contexts. Some simpler than others but all following the same pattern of allowing discarding of bindings to an identifier. It is allowed anywhere the *BindingPattern*, *LexicalBinding* or *DestructuringAssignmentTarget* features are used in EcmaScript. This means it can be applied to unpacking of objects/arrays, in callback parameters and class methods.

```

1 using void = new UniqueLock(mutex);
2 // Not allowed on top level of var/let/const declarations
3 const void = bar(); // Illegal

```

Listing 3.4: Example discard binding with variable discard

```

1 let {b:void, ...rest} = {a:1, b:2, c:3, d:4}
2 rest; // {a:1, c:3, d:4};

```

Listing 3.5: Example Object binding and assignment pattern

```

1 function* gen() {
2   for (let i = 0; i < Number.MAX_SAFE_INTEGER; i++) {
3     console.log(i);
4     yield i;
5   }
6 }
7
8 const iter = gen();
9 const [a, , ] = iter;

```

```

10 // prints:
11 // 0
12 // 1
13
14 const [a, void] = iter; // author intends to consume two elements
15 // vs.
16 const [a, void, void] = iter; // author intends to consume three
    ↪ elements

```

Listing 3.6: Example Array binding and assignment pattern. It is not clear to the reader that in line 8 we are consuming 2 or 3 elements of the iterator. In the example on line 13 we see that is it more explicit how many elements of the iterator is consumed

```

1 // project an array values into an array of indices
2 const indices = array.map((void, i) => i);
3
4 // passing a callback to 'Map.prototype.forEach' that only cares about
5 // keys
6 map.forEach((void, key) => { });
7
8 // watching a specific known file for events
9 fs.watchFile(fileName, (void, kind) => { });
10
11 // ignoring unused parameters in an overridden method
12 class Logger {
13   log(timestamp, message) {
14     console.log(`${timestamp}: ${message}`);
15   }
16 }
17
18 class CustomLogger extends Logger {
19   log(void, message) {
20     // this logger doesn't use the timestamp...
21   }
22 }
23
24 // Can also be utilized for more trivial examples where _ becomes
25 // cumbersome due to multiple discarded parameters.
26 doWork((_, a, _1, _2, b) => {});
27 // vs.
28 doWork((void, a, void, void, b) => {
29 });

```

Listing 3.7: Example discard binding with function parameters. This avoids needlessly naming parameters of a callback function that will remain unused.

The grammar of this proposal is precisely specified in the specification found in the proposal definition on github.

```

1 var [void] = x;           // via: BindingPattern :: 'void'
2 var {x:void};            // via: BindingPattern :: 'void'
3
4 let [void] = x;          // via: BindingPattern :: 'void'
5 let {x:void};            // via: BindingPattern :: 'void'
6
7 const [void] = x;         // via: BindingPattern :: 'void'
8 const {x:void} = x;       // via: BindingPattern :: 'void'
9
10 function f(void) {}       // via: BindingPattern :: 'void'
11 function f([void]) {}     // via: BindingPattern :: 'void'
12 function f({x:void}) {}   // via: BindingPattern :: 'void'
13
14 ((void) => {});            // via: BindingPattern :: 'void'
15 (([void]) => {});         // via: BindingPattern :: 'void'
16 (({x:void}) => {});       // via: BindingPattern :: 'void'

```

```

17
18 using void = x;           // via: LexicalBinding : 'void' Initializer
19 await using void = x;     // via: LexicalBinding : 'void' Initializer
20
21 [void] = x;               // via: DestructuringAssignmentTarget : 'void'
22 ({x:void} = x);          // via: DestructuringAssignmentTarget : 'void'

```

Listing 3.8: Grammar of Discard Binding

3.2.4 Pipeline Proposal

The pipeline proposal is a Syntactic proposal with no change to functionality of EcmaScript, it focuses solely on solving problems related to nesting of function calls and other expressions that allow for a topic reference. A topic reference is a reference to some value based on the current context/topic.

The pipeline 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, these two 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.

Nested calls has some specific challenges when relating to readability when used. The order of calls go from right to left, which is opposite of the natural reading direction a lot of the users of EcmaScript are used to day to day, this means it is difficult switch 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, this is also a problem with readability of nested calls, which is the main challenge this proposal is trying to solve. Nested calls are not all bad however, one of the main good points of nested calls is 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, a library developer does not have to design their library around this specific call style.

```

1 // Deeply nested call with single arguments
2 function1(function2(function3(function4(value))));
3
4 // Deeply nested call with multi argument functions
5 function1(function2(function3(value2, function4)), value1);

```

Chaining solves some of the issues relating to nesting, 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 respoective function call, and it provides a way of untangling deep nesting. However, solving consecutive operations using chaining has its own set of challenges when used. In order to use chaining, the api of the code you

are trying to call has to be designed to allow for chaining. This is not always the case, making using chaining when it is not been designed specifically for very difficult. There are also concepts in JavaScript not supported when using chaining, such as arithmetic operations, literals, await, yield and so on. This is actually the biggest downside of chaining, as it only allows for function calls when used, and if one wants to allow for use of other concepts temporary variables have to be used.

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

The pipeline proposal aims to combine the benefits of these two styles without all the challenges each method faces.

The main benefit of pipeline is to allow for a similar style to chaining when chaining has not been specifically designed to be applicable. The idea uses syntactic sugar to change the order of writing the calls without influencing the API of the functions. Doing this allows each call to come in the direction of left to right, while still maintaining the modularity of deeply nested function calls.

The way the pipeline proposal aims to solve this is to introduce 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.

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

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

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

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

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

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

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

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

3.2.5 Do proposal

The [8, Do Proposal] is a proposal meant to bring *expression oriented* programming to EcmaScript. Expression oriented programming is a concept taken from functional programming which allows for combining expressions in a very free manor allowing for a highly malleable programming experience.

The motivation of the do expression proposal is to create a feature that allows for local scoping of a code block that is treated as an expression. This allows for complex code requiring multiple statements to be confined inside its own scope and the resulting value is returned from the block as an expression. Similar to how a unnamed functions or arrow functions are currently used. The current status quo of how to achieve this behavior is to use unnamed functions and invoke them immediately, or use an arrow function, these two are equivalent to a do expression.

The codeblock of a do expression has one major difference from these equivalent functions, as it allows for implicit return of the final expression of the block, and is the resulting value of the entire do expression.

The local scoping of this feature allows for a cleaner environment in the parent scope of the do expression. What is meant by this is for temporary variables and other assignments used once can be enclosed inside a limited scope within the do block. Allowing for a cleaner environment inside the parent scope where the do block is defined.

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

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

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

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

This proposal has some limitations on its usage. Due to the implicit return of the final expression you cannot end a do expression with an `if` without an `else`, or a `loop`.

3.2.6 Await to Promise

This section covers an imaginary proposal that was used to evaluate the program developed in this thesis. This imaginary proposal is less of a proposal and more of just a pure JavaScript transformation example. What this proposal wants to achieve is transforming a function using `await`, into a function that uses and returns a promise.

In order to do this an equivalent way of writing code containing `await` in the syntax of `.then()` promise, an equivalent way of writing the same functionality has to be identified. In this case, the equivalent way of expressing this using a promise is consuming the rest of the scope after `await` was written, and place it inside a `then(() =>)` function. The variable the `await` was assigned to has to be used as the argument to the `.then()` function.

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

3.3 Searching user code for applicable snippets

In order to identify snippets of code in the users codebase where a proposal is applicable we need some way to define patterns of code where we can apply the proposal. To do this, a DSL titled JSTQL is used.

3.3.1 JSTQL

Showcasing a proposal using a users code requires some way of identifying applicable code sections to that proposal. To do this, we have designed a DSL called JSTQL , JavaScript Template Query Language. This DSL will contain the entire definition used to identify and transform user code in order to showcase a proposal.

Identifying applicable code

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

Wildcards are written into the template inside a block denoted by `<< >>`. Each wildcard has to start 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 all together can be transferred from the original user code into the transformation template. The second part of the wildcard contains a wildcard type expression. A wildcard type expression is a way of defining exactly what types of AST nodes a wildcard will produce a match against, these type expressions use boolean logic together with the AST node-types from [1]BabelJS to create a very strict way of defining wildcards.

Wildcard type expressions

Wildcard type expressions allow for writing complex boolean logic on the kinds of nodes a wildcard can be matched against, this means writing the type for a wildcard can be as simple as just `Expression || Statement`, or as complex as `((Statement && ! ReturnStatement) && ! VariableDeclaration)*`. The operators mean the following, `&&` is logical AND, this means both parts of the expression have to evaluate to true, `||` means logical OR, so either side of expression can be true for the entire expression to be true, `!` is the only unary expression, and is logical NOT, so `!Statement` is any node that is NOT a Statement. There is also this definition `+`, this is only valid at the top level of the expression, and the pluss operator means this wildcard can be used any number of times in order. This is useful for matching against a series of Statements, while not wanting to match an entire BlockStatement. The final part of a wildcard expression is the AST types used, an example of this would be the wildcard type expression `ReturnStatement`, which will only match against an AST node of type ReturnStatement. Using the power of the wildcards, we can create templates that can be used for querying a users code for specific code sections that a proposal is applicable to.

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

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

3.3.2 Transforming

When matching sections of the users code has been found, we need some way of defining how to transform those sections to showcase a proposal. This is done by a similar template to applicable to, namely *transform to*, this template describes the general structure of the newly transformed code.

A transformation template is used to define how the matches will be transformed after applicable code has been found. The transformation is a general template of the code once the match is replaced in the original AST. However, without transferring over the context from the match, it is just a template search and replace. So in order to transfer the context from the match, wildcards are defined in this template as well. These wildcards use the same block notation found in the applicable to template, however they do not

need to contain the types, as those are not needed in the transformation. The only section required in the wildcard is the identifier used in applicable to, this is done in order to know which wildcard match we are taking the context from, and where to place it in the transformation template.

```
1 // Example of transform to template
2 const variableName = <<expr1>>;
```

3.3.3 Structure of JSTQL

JSTQL is designed to mimic the examples already provided by a proposal champion in the proposals README. These examples can be seen in each of the proposals described in 3.2. The idea is to allow a similar kind of notation to the examples in order to define the transformations.

Define proposal

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

```
1 proposal Pipeline_Proposal{
2
3 }
```

Listing 3.9: Example of section containing the pipeline proposal

Defining a pair of template and transformation

Each proposal will have 1 or more definitions of a template for code to identify in the users codebase, and its corresponding transformation definition. These are grouped together in order to have a simple way of identifying the corresponding cases of matching and transformations. This section of the proposal is defined by the keyword *case* and a block to contain its related fields. A proposal will contain 1 or more of this section. This allows for matching many different code snippets and showcasing more of the proposal than a single concept the proposal has to offer.

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

Listing 3.10: Example of pair section

Template used for matching

In order to define the template used to match, we have another section defined by the keyword *applicable to*. This section will contain the template defined using JavaScript with specific DSL keywords defined inside the template.

```
1 applicable to {  
2  
3 }
```

Listing 3.11: Example of applicable to section

Defining the transformation

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

```
1 transform to{  
2  
3 }
```

Listing 3.12: Example of transform to section

All sections together

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

```
1 proposal PROPOSAL_NAME {  
2   case PAIR_NAME {  
3     applicable to {  
4  
5     }  
6     transform to {  
7  
8     }  
9   }  
10  pair PAIR_NAME {  
11    applicable to .....  
12  }  
13  
14  pair .....  
15 }
```

Listing 3.13: JSTQL definition of a proposal

3.4 Using the JSTQL with an actual syntactic proposal

This section contains the definitions of the proposals used to evaluate the tool created in this thesis. These definitions do not have to cover every single case where the proposal might be applicable, as they just have to be general enough to create some amount of examples that will give a representative number of matches when the transformations are applied to some relatively long user code.

3.4.1 Pipeline Proposal

The Pipeline Proposal is the easiest to define of the proposals presented in 3.2. This is due to the proposal being applicable to a very wide array of expressions, and the main problem this proposal is trying to solve is deep nesting of function calls.

```
1 proposal Pipeline{
2
3   case SingleArgument {
4     applicable to {
5       "<<someFunctionIdent:Identifier ||
6         ↳ MemberExpression>>(<<someFunctionParam:
7         ↳ Expression>>);"
8     }
9     transform to {
10      "<<someFunctionParam>> |> <<someFunctionIdent>>(%);"
11    }
12  }
13  case DualArgument{
14    applicable to {
15      "<<someFunctionIdent: Identifier ||
16        ↳ MemberExpression>>(<<someFunctionParam:
17        ↳ Expression>>, <<moreFunctionParam: Expression>>)"
18    }
19    transform to {
20      "<<someFunctionParam>> |> <<someFunctionIdent>>(%,"
21      ↳ <<moreFunctionParam>>)"
22    }
23  }
24 }
```

Listing 3.14: Example of Pipeline Proposal definition in JSTQL

This first pair definition **SingleArgument** of the Pipeline proposal will apply to any *CallExpression* with a single argument. And it will be applied to each of the deeply nested callExpressions in a nested call. The second pair definition **DualArgument** will apply to any *CallExpression* with 2 arguments. One can in theory define any number of cases to cover a higher amount of arguments in the function call, however we only need to cover enough cases to produce at least some matches, so a smaller definition is better in this case

3.4.2 Do Proposal

The [8, Do Proposal] can also be defined with this tool. This definition will never catch all the applicable sections of the users code, and is very limited in where it might discover this proposal is applicable. This is due to the Do Proposal introducing an entirely new way to write JavaScript (Expression-oriented programming). If the user running this tool has not used the current status-quo way of doing expression-oriented programming in JavaScript, JSTQL will probably not find any applicable snippets in the users code. However, in a reasonably large codebase, some examples will probably be discovered.

```
1 proposal DoExpression{
2   case arrowFunction{
3     applicable to {
4       "let <<ident:Identifier>> = () => {
5         <<statements: (Statement && !ReturnStatement)*>>
6         return <<returnVal : Expression>>;
7       }
8     }
9   }
10  transform to {
11    "let <<ident>> = do {
12      <<statements>>
13      <<returnVal>>
14    }"
15  }
16 }
17
18 case immediatelyInvokedUnnamedFunction {
19   applicable to {
20     "let <<ident:Identifier>> = function(){
21       <<statements: (Statement && !ReturnStatement)*>>
22       return <<returnVal : Expression>>;
23     }();"
24   }
25
26   transform to {
27     "let <<ident>> = do {
28       <<statements>>
29       <<returnVal>>
30     }"
31   }
32 }
33 }
```

Listing 3.15: Definition of Do Proposal in JSTQL

3.4.3 Await to Promises evaluation proposal

This section will cover the evaluation proposal we created in order to evaluate this tool described in 3.2.

This proposal was created in order to evaluate the tool, as it is quite difficult to define applicable code in this current template form. This definition is limited, and can only apply if the function only contains a single await expression. This actually highlights some of the issues with the current design of JSTQL that will be described in Future Work.


```

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

```

Listing 3.16: Definition of Await to Promise evaluation proposal in JSTQL

Chapter 4

Implementation

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

4.1 Architecture

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

In this tool, there exists two multiple ways to define a proposal, and each provide the same functionality, they only differ in syntax and writing-method. One can either write the definition in JSTQL which utilizes Langium to parse the language, or one can use a JSON definition, which is more friendly as an API or people more familiar with JSON definitions.

4.2 Parsing JSTQL using Langium

In this section, the implementation of the parser for JSTQL will be described. This section will outline the tool Langium, used as a parser-generator to create the AST used by the tool later to perform the transformations.

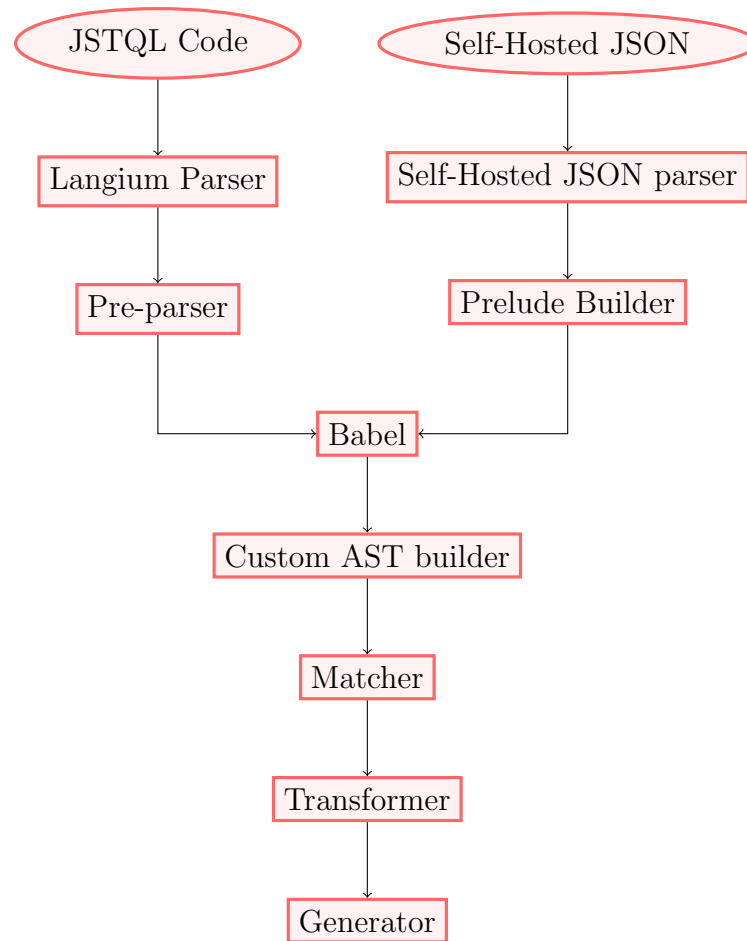


Figure 4.1: Overview of tool architecture

4.2.1 Langium

Langium [6] is primarily used to create parsers for Domain Specific Language, these kinds of parsers output an Abstract Syntax Tree that is later used to create interpreters or other tooling. In the case of JSTQL we use Langium to generate an AST definition in the form of TypeScript Objects, these objects and their relation are used as definitions for the tool to do matching and transformation of user code.

In order to generate this parser, Langium required a definition of a Grammar. A grammar is a set of instructions that describe a valid program. In our case this is a definition of describing a proposal, and its applicable to, transform to, descriptions. A grammar in Langium starts by describing the **Model**. The model is the top entry of the grammar, this is where the description of all valid top level statements.

In JSTQL the only valid top level statement is the definition of a proposal. This means our language grammar model contains only one list, which is a list of 0 or many **Proposal** definitions. A Proposal definition is denoted by a block, which is denoted by `{...}` containing some valid definition. In the case of JSTQL this block contains 1 or many definitions of **Case**.

Case is defined very similarly to **Proposal**, as it contains only a block containing a definition of a **Section**

The **Section** is where a single case of some applicable code and its corresponding transformation is defined. This definition contains specific keywords to describe each of them, **applicable to** denotes a definition of some template JSTQL uses to perform the matching algorithm. **transform to** contains the definition of code used to perform the transformation.

In order to define exactly what characters/tokens are legal in a specific definition, Langium uses terminals defined using Regular Expressions, these allow for a very specific character-set to be legal in specific keys of the AST generated by the parser generated by Langium. In the definition of **Proposal** and **Pair** the terminal **ID** is used, this terminal is limited to allow for only words and can only begin with a character of the alphabet or an underscore. In **Section** the terminal **TEXT** is used, this terminal is meant to allow any valid JavaScript code and the custom DSL language described in 3.3.1. Both these terminals defined allows Langium to determine exactly what characters are legal in each location.

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

```

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

```

Listing 4.1: Definition of JSTQL in Langium

In the case of JSTQL , we are not actually implementing a programming language meant to be executed. We are using Langium in order to generate an AST that will be used as a markup language, similar to YAML, JSON or TOML. The main reason for using Langium in such an unconventional way is Langium provides support for Visual Studio Code integration, and it solves the issue of parsing the definition of each proposal manually. However with only the grammar we cannot actually verify the wildcards placed in `apl_to_code` and `transform_to_code` are correctly written. This is done by using a feature of Langium called `Validator`.

Langium Validator

A Langium validator allows for further checks on the templates written withing JSTQL , a validator allows for the implementation of specific checks on specific parts of the grammar.

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

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

```

1 export class JstqlValidator {
2     validateWildcardAplTo(pair: Pair, accept: ValidationAcceptor):
3         ↪ void {
4         try {
5             if (validationResultAplTo.errors.length != 0) {

```

```

5         accept("error",
6             ↪ validationResultAplTo.errors.join("\n"), {
7                 node: pair.aplTo,
8                 property: "apl_to_code",
9             });
10    }
11    if (validationResultTraTo.length != 0) {
12        accept("error", validationResultTraTo.join("\n"), {
13            node: pair.traTo,
14            property: "transform_to_code",
15        });
16    } catch (e) {}
17 }
18 }

```

Using Langium as a parser

[6]Langium is designed to automatically generate a lot of tooling for the language specified using its grammar. However, in our case we have to parse the JSTQL definition using Langium, and then extract the Abstract syntax tree generated in order to use the information it contains.

To use the parser generated by Langium, we created a custom function `parseDSLtoAST()` within our Langium project, this function takes a string as an input, the raw JSTQL code, and outputs the pure AST using the format described in the grammar described in Figure 4.1. This function is exposed as a custom API for our tool to interface with. This also means our tool is dependent on the implementation of the Langium parser to function with JSTQL. The implementation of JSTQL-SH is entirely independent.

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

4.3 Pre-parsing

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

Why not use Langium?

Langium has support for creating a generator for generating an artifact, this actually suits the needs of JSTQL quite well and could be used to extract the wildcards from each `pair` and create the `TransformRecipe`. This would, as a consequence, make JSTQL-SH not be entirely independent, and the entire tool would rely on Langium. This is not preferred as that would mean both ways of defining a proposal both are reliant of Langium and not separated. The reason for using our own pre-parser is to allow for an independent way to define transformations using our tool.

Extracting wildcards from JSTQL

In order to allow the use of [1, Babel], the wildcards present in the blocks of `applicable to` and `transform to` have to be parsed and replaced with some valid JavaScript. This is done by using a pre-parser that extracts the information from the wildcards and inserts an `Identifier` in their place.

To pre-parse the text, we look at each and every character in the code section, when a start token of a wildcard is discovered, which is denoted by `<<`, everything after that until the closing token, which is denoted by `>>`, is then treated as an internal DSL variable and will be stored by the tool. A variable `flag` is used, so when the value of `flag` is false, we know we are currently not inside a wildcard block, this allows us to just pass the character through to the variable `cleanedJS`. When `flag` is true, we know we are currently inside a wildcard block and we collect every character of the wildcard block into `temp`. Once we hit the end of the wildcard block, when we have consumed the entirety of the wildcard, it is then passed to a tokenizer, then to a recursive descent parser.

```
1 export function parseInternal(code: string): InternalParseResult {
2   let cleanedJS = "";
3   let temp = "";
4   let flag = false;
5   let prelude: InternalDSLVariable = {};
6
7   for (let i = 0; i < code.length; i++) {
8     if (code[i] === "<<" && code[i + 1] === "<") {
9       // From now in we are inside of the DSL custom block
10      flag = true;
11      i += 1;
12      continue;
13    }
14
15    if (flag && code[i] === ">>" && code[i + 1] === ">") {
16      // We encountered a closing tag
17      flag = false;
18
19      let { identifier, types } = parseInternalString(temp);
20
21      cleanedJS += identifier;
22
23      prelude[identifier] = types;
24      i += 1;
25      temp = "";
26      continue;
27    }
28
29    cleanedJS += code[i];
30  }
31}
```

```

27     }
28
29     if (flag) {
30         temp += code[i];
31     } else {
32         cleanedJS += code[i];
33     }
34 }
35 return { prelude, cleanedJS };
36 }

```

Parsing wildcard

Once a wildcard has been extracted from the `pair` definitions inside JSTQL , they have to be parsed into a simple Tree to be used when matching against the wildcard. This is accomplished by using a simple tokenizer and a [9]Recursive Descent Parser.

Our tokenizer simply takes the raw stream of input characters extracted from the wildcard block within the template, and determines which part is what token. Due to the very simple nature of the type expressions, no ambiguity is present with the tokens, so determining what token is meant to come at what time is quite trivial. The tokenizer **I need to figure out what kind of tokenization algorithm i am actually using LOL**

A recursive descent parser is created to closely mimic the grammar of the language the parser is implemented for, where we define functions for handling each of the non-terminals and ways to determine what non terminal each of the tokens result in. In the case of this parser, the language is a very simple boolean expression language. We use boolean combinatorics to determine whether or not a specific AST nodetype of a [3]Babel parser AST node is a match against a specific wildcard. This means we have to create a very simple AST that can be evaluated using the AST nodetype as an input.

```

1 Wildcard:
2     Identifier ":" MultipleMatch
3
4 MultipleMatch:
5     GroupExpr "*"
6     | TypeExpr
7
8 TypeExpr:
9     BinaryExpr
10    | UnaryExpr
11    | PrimitiveExpr
12
13 BinaryExpr:
14     TypeExpr { Operator TypeExpr }*
15
16 UnaryExpr:
17     {UnaryOperator}? TypeExpr
18
19 PrimitiveExpr:
20     GroupExpr | Identifier
21
22 GroupExpr:
23     "(" TypeExpr ")"

```

Listing 4.2: Grammar of type expressions

The grammar of the type expressions used by the wildcards can be seen in Figure 4.2, the grammar is written in something similar to Extended Backus-Naur form, where we define the terminals and non-terminals in a way that makes the entire grammar *solvable* by the Recursive Descent parser.

Our recursive descent parser produces a very simple [11, 12]AST which is later used to determine when a wildcard can be matched against a specific AST node, the full definition of this AST can be seen in A.1. We use this AST by traversing it using a [13]visitor pattern and comparing each *Identifier* against the specific AST node we are currently checking, and evaluating all subsequent expressions and producing a boolean value, if this value is true, the node is matched against the wildcard, if not then we do not have a match.

Pre-parsing JSTQL-SH

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

In order to use JavaScript as the meta language to define JavaScript we define a **Prelude**. This prelude is required to consist of several **Declaration Statements** where the variable names are used as the internal DSL variables and right side expressions are used as the DSL types. In order to allow for multiple types to be allowed for a single internal DSL variable we re-use JavaScripts list definition.

We use Babel to generate the AST of the **prelude** definition, this allows us to get a JavaScript object structure. Since the structure is very strictly defined, we can expect every **stmt** of **stmts** to be a variable declaration, otherwise throw an error for invalid prelude. Continuing through the object we have to determine if the prelude definition supports multiple types, that is if it is either an **ArrayDeclaration** or just an **Identifier**. If it is an array we initialize the prelude with the name field of the **VariableDeclaration** to either an empty array and fill it with each element of the **ArrayDeclaration** or directly insert the single **Identifier**.

4.4 Using Babel to parse

Allowing the tool to perform transformations of code requires the generation of an Abstract Syntax Tree from the users code, **applicable to** and **transform to**. This means parsing JavaScript into an AST, in order to do this we use a tool [1, Babel].

The most important reason for choosing to use Babel for the purpose of generating the AST's used for transformation is due to the JavaScript community surrounding Babel. As this tool is dealing with proposals before they are part of JavaScript, a parser that supports early proposals for JavaScript is required. Babel supports most Stage 2 proposals through its plugin system, which allows the parsing of code not yet part of the language.

Custom Tree Structure

To allow for matching and transformations to be applied to each of the sections inside a `pair` definition, they have to be parsed into an AST in order to allow the tool to match and transform accordingly. To do this the tool uses the library [1, Babel] to generate an AST data structure. However, this structure does not suit traversing multiple trees at the same time, this is a requirement for matching and transforming. Therefore we use this Babel AST and transform it into a simple custom tree structure to allow for simple traversal of the tree.

As can be seen in Figure 4.3 we use a recursive definition of a `TreeNode` where a node's parent either exists or is null (it is top of tree), and a node can have any number of children elements. This definition allows for simple traversal both up and down the tree. Which means traversing two trees at the same time can be done in the matcher and transformer section of the tool.

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

Listing 4.3: Simple definition of a Tree structure in TypeScript

Placing the AST generated by Babel into this structure means utilizing the library [4]Babel Traverse. Babel Traverse uses the [13]visitor pattern to allow for traversal of the AST. While this method does not suit traversing multiple trees at the same time, it allows for very simple traversal of the tree in order to place it into our simple tree structure.

[4]Babel Traverse uses the [13]visitor pattern to visit each node of the AST in a *depth first* manner, the idea of this pattern is one implements a *visitor* for each of the nodes in the AST and when a specific node is visited, that visitor is then used. In the case of transferring the AST into our simple tree structure we simply have to use the same visitor for all nodes, and place that node into the tree.

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

```

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

```

4.5 Matching

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

Determining if AST nodes match

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

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

Since a wildcard is replaced by an `Identifier`, when matching a node in the template, we have to check if it is the *Identifier* or *ExpressionStatement* with an identifier contained within, if there is an identifier, we have to check if that identifier is a registered wildcard. If an `Identifier` shares a name with a wildcard, we have to compare the node against

the Type expression of that wildcard. When we do this, we traverse the entirety of the wildcard expression AST and compare each of the leaves against the type of the current code node. These resulting values are then passed through the type expression and the resulting value is whether or not that code node can be matched against the wildcard. We differentiate between if a node matched against a wildcard with the `+` notation, as if that is the case we have to keep using that wildcard until it returns false in the tree exploration algorithms.

When we are either matching against an Identifier that is not a registered wildcard, or any other AST node in the template, we have to perform an equality check, in the case of this template language, we can get away with just performing some preliminary checks, such as that names of Identifiers are the same. Otherwise it is sufficient to just perform an equality check of the types of the nodes we are currently trying to match. If the types are the same, they can be validly matched against each other. This is sufficient because we are currently trying to determine if a single node can be a match, and not the entire template structure is a match. Therefore false positives that are not equivalent are highly unlikely due to the fact the entire structure has to be a false positive match.

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

```
1 enum MatchResult {  
2     MatchedWithWildcard,  
3     MatchedWithPlussedWildcard,  
4     Matched,  
5     NoMatch,  
6 }
```

Matching a singular Expression/Statement template

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

Determining if we are currently trying to match with a template that is only a single expression/statement, we have to verify that the program body of the template has the

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

When matching an expression the first statement in the program body of the AST generated when using [2]babel generate will be of type **ExpressionStatement**, the reason for this is Babel will treat free floating expressions as a statement, and place them into an **ExpressionStatement**. This will miss many applicable sections in the case of trying to match against a users code because expressions within other statements are not inside an **ExpressionStatement**. This will give a template that is incompatible with a lot of otherwise applicable expressions. This means the statement **ExpressionStatement** has to be removed, and the search has to be done with the expression as the top node of the template.

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

Recursively discovering matches

The matcher used against single **Expression/Statement** templates is based upon a Depth-First Search in order to perform matching, and searches for matches from the top of the code definition. It is important we try to match against the template at all levels of the code AST, this is done by starting a new search on every child node of the code AST if the current node of the template tree is the top node of the template. This ensures we have tried to perform a match at any level of the tree, this also means we do not get any partial matches, as we only store matches that are returned at the recursive call when we do the search from the first node of the template tree. This is all done before ever checking the node we are currently on. The reason for this is to avoid missing matches that reside further down in the current branch, and also ensure matches further down are placed earlier in the full match array, which makes it easier to perform transformation when partial collisions exist.

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

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

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

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

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

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

```

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

```

Matching multiple Statements

Using multiple statements in the template of `applicable to` will result in a much stricter matcher, that will only try to perform an exact match using a [10]sliding window of the amount of statements at every *BlockStatement*, as that is the only placement Statements can reside in JavaScript[5].

The initial step of this algorithm is to search through the AST for ast nodes that contain a list of *Statements*, this can be done by searching for the AST nodes *Program* and *BlockStatement*, as these are the only valid places for a list of Statements to reside [5]. Searching the tree is quite simple, as all that is required is checking the type of every

node recursively, and once a node that can contain multiple Statements, we check it for matches.

Once a list of *Statements* has been discovered, the function `matchMultiHead` can be executed with that block and the Statements of `applicable to`. This function will use the technique [10]sliding window to match multiple statements in order the same length as the list of statements are in `applicable to`. This sliding window will try to match every Statement against its corresponding Statement in the current *BlockStatement*. When matching a singular Statements in the sliding window, a simple DFS recursion algorithm is applied, similar to algorithm used for matching a single expression/statement template, however the difference is that we do not search the entire AST tree, and if it matches it has to match fully and immediately. If a match is not found, the current iteration of the sliding window is discarded and we move on to the next iteration by moving the window one further.

One important case here is we might not know the width of the sliding window, this is due to wildcards using the `+`, as they can match one or more nodes against each other. These wildcards might match against `(Statement)+`. Therefore, we have to use a two point technique when iterating through the statements of the users code. As we might have to use the same statement from the template multiple times.

Output of the matcher

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

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

4.6 Transforming

To perform the transformation and replacement on each of the matches, we take the resulting list of matches, the template from the `transform to` section of the current case of the proposal, and the AST version of original code parsed by Babel. All the transformations are then applied to the code and we use [2]Babel generate to generate JavaScript code from the transformed AST.

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

Preparing the transform to template

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

In order to perform the transformation, all the sections matched against a wildcard have to be transferred into the `transform to template`. We utilize the functionality from Babel here and traverse the generated AST of the transform to template using [4]Babel traverse, as this gives us utility functions to replace, replace with many, and remove nodes of the AST. We use custom visitors for *Identifier* and *ExpressionStatement* with an Identifier as expression, in order to determine where the wildcard matches have to be placed, as they have to be placed at the same location that shares a name with the wildcard. Once a shared identifier between the `transform to template` and the `applicable to template` is discovered, a babel traverse replace with multiple is performed and the node/s found in the match is inserted in place of the wildcard.

Inserting the template into the AST

Having a transformed version of the users code, it has to be inserted into the full AST definition of the users code, again we use [4]babel/traverse to traverse the entirety of the code AST using a visitor. This visitor does not apply to any node-type, as the matched section can be any type. Therefore we use a generic visitor, and use an equality check to find the exact part of the code this specific match comes from. Once we find where in the users code the match came from, we replace it with the transformed `transform to` nodes. This might be multiple Statements, therefore the function `replaceWithMultiple` is used, to insert every Statement from the `transform to` body, and we are careful to remove any following sibling nodes that were part of the original match. This is done by removing the $n-1$ next siblings from where we inserted the transform to template.

To generate JavaScript from the transformed AST created by this tool, we use a JavaScript library titled [2]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.

Bibliography

- [1] Babel · Babel, May 2024. [Online; accessed 10. May 2024].
- [2] @babel/generator · Babel, May 2024. [Online; accessed 12. May 2024].
- [3] @babel/parser · Babel, May 2024. [Online; accessed 14. May 2024].
- [4] @babel/traverse · Babel, May 2024. [Online; accessed 12. May 2024].
- [5] ECMAScript® 2025 Language Specification, April 2024. [Online; accessed 13. May 2024].
- [6] Langium, April 2024. [Online; accessed 10. May 2024].
- [7] proposal-discard-binding, April 2024. [Online; accessed 25. Apr. 2024].
- [8] proposal-do-expressions, May 2024. [Online; accessed 2. May 2024].
- [9] Matthew S. Davis. An object oriented approach to constructing recursive descent parsers. *SIGPLAN Not.*, 35(2):29–35, feb 2000.
- [10] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. Sliding-Window Aggregation Algorithms: Tutorial. In *DEBS '17: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 11–14. Association for Computing Machinery, New York, NY, USA, June 2017.
- [11] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery.
- [12] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, may 2005.
- [13] J. Palsberg and C.B. Jay. The essence of the visitor pattern. In *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)*, pages 9–15, 1998.

Appendix A

Generated code from Protocol buffers

```
1 export interface Identifier extends WildcardNode {
2   nodeType: "Identifier";
3   name: string;
4 }
5
6 export interface Wildcard {
7   nodeType: "Wildcard";
8   identifier: Identifier;
9   expr: TypeExpr;
10  star: boolean;
11 }
12
13 export interface WildcardNode {
14   nodeType: "BinaryExpr" | "UnaryExpr" | "GroupExpr" | "Identifier";
15 }
16
17 export type TypeExpr = BinaryExpr | UnaryExpr | PrimitiveExpr;
18
19 export type BinaryOperator = "||" | "&&";
20
21 export type UnaryOperator = "!";
22
23 export interface BinaryExpr extends WildcardNode {
24   nodeType: "BinaryExpr";
25   left: UnaryExpr | BinaryExpr | PrimitiveExpr;
26   op: BinaryOperator;
27   right: UnaryExpr | BinaryExpr | PrimitiveExpr;
28 }
29 export interface UnaryExpr extends WildcardNode {
30   nodeType: "UnaryExpr";
31   op: UnaryOperator;
32   expr: PrimitiveExpr;
33 }
34
35 export type PrimitiveExpr = GroupExpr | Identifier;
36
37 export interface GroupExpr extends WildcardNode {
38   nodeType: "GroupExpr";
39   expr: TypeExpr;
40 }
```

Listing A.1: TypeScript types of Type Expression AST