

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
Sunday 12th 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	[5, Discard Bindings]	5
3.2.4	Pipeline Proposal	8
3.2.5	Description of Pipeline proposal	9
3.2.6	Do proposal	10
3.2.7	Await to Promise	11
3.3	Searching user code for applicable snippets	11
3.3.1	JSTQL	12
3.3.2	Matching	12
3.3.3	JSTQL custom matching types	13
3.3.4	Transforming	13
3.3.5	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
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	24
4.5	Matching	26
4.6	Transforming	26

4.7 Generating	26
Bibliography	27
A Generated code from Protocol buffers	28

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 deeply nested call	8
3.10	Example of chaining calls	9
3.11	Example from jquery	9
3.12	Example from unpublish	9
3.13	Example from underscore.js	9
3.14	Example from ramda.js	10
3.15	Example of do expression	10
3.16	Example of await to promises	11
3.17	Example of a wildcard	12
3.18	See 3.17 contains identifier expr1, and we refer to the same in this example, the only transformation happening here is rewriting let to const.	13
3.19	Example of section containing the pipeline proposal	13
3.20	Example of pair section	14
3.21	Example of applicable to section	14
3.22	Example of transform to section	14
3.23	JSTQL definition of a proposal	15
3.24	Example of Pipeline Proposal definition in JSTQL	15
3.25	Definition of Do Proposal in JSTQL	16
4.1	Definition of JSTQL in Langium	20
4.2	Simple definition of a Tree structure in TypeScript	25
A.1	Source code of something	28

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

THIS IS TOO ABRUPT OF AN INTRODUCTION, MORE GENERAL ALMOST REPEAT OF BACKGGROUND GOES HERE

CURRENT VERSION vs FUTURE VERSION instead of old way

DO NOT DISCUSS TOOL HERE

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 *old* and *new* way of programming in 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 *new* way of solving said problems. The *old* way is the current status quo when the proposal is not part of EcmaScript, and the *new* way 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 will get feedback from users of the language earlier in the proposal process, this will ideally allow for a more efficient process of adding proposals to 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 tool has 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 the tool has matched all parts of the program that the proposal could be applied, the users code has to be transformed to use the feature/s the proposal is trying to implement. 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 bugs. 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 [5, 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.

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. See in 3.9.

Challenges with nested calls

- The order of calls go from right to left, which is opposite of the natural reading direction users of EcmaScript are used to
- When introduction functions with multiple arguments in the middle of the nested call, it is not intuitive to see what call it belongs to.

Benefits of nested calls

- Does not require special design thought to be used

```

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);

```

Listing 3.9: Example of deeply nested call

Nesting 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. However, solving consecutive operations using chaining has its own set of challenges when used

3.2.5 Description of Pipeline proposal

Challenges with chaining calls

- APIs has to be specifically designed with chaining in mind
- Might not even be possible due to external libraries
- Does not support other concepts such as arithmetic operations, array/object literals, await, yield, etc...

Benefits of chaining calls

- More natural direction of call order
- Arguments of functions are grouped with function name
- Untangles deep nesting

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

Listing 3.10: Example of chaining calls

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.

```
1 // Status quo
2 var minLoc = Object.keys( grunt.config( "uglify.all.files" ) )[ 0 ];
3
4 // With pipes
5 var minLoc = grunt.config('uglify.all.files') |> Object.keys(%) [0];
```

Listing 3.11: Example from jquery

```
1 // Status quo
2 const json = await npmFetch.json(npa(pkgs[0]).escapedName, opts);
3
4 // With pipes
5 const json = pkgs[0] |> npa(%).escapedName |> await
  ↪ npmFetch.json(%, opts);
```

Listing 3.12: Example from unpublish

```
1 // Status quo
2 return filter(obj, negate(cb(predicate)), context);
3
4 // With pipes
5 return cb(predicate) |> _.negate(%) |> _.filter(obj, %, context);
```

Listing 3.13: Example from underscore.js

```

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

```

Listing 3.14: Example from ramda.js

3.2.6 Do proposal

The [6, 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 function is used currently. 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 statement in the block. This only works if the statement does not contain a final line end (;).

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 };
6
7 // Using a immediately invoked function
8 let x = function(){
9   let tmp = f();
10  return tmp + tmp + 1;
11 }();
12
13 // Using do expression

```

```

14 let x = do {
15     let tmp = f();
16     tmp + tmp + 1
17 }

```

Listing 3.15: Example of do expression

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

3.2.7 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 re-writing from using `await` so use promises.

In order to do this an equivalent way of writing code containing `await` in the syntax of `promises` had to be identified. In this case, the equivalent way of expressing this is consuming the rest of the scope `await` was written in and place it inside a `then(() =>)` function.

```

1 // Code containing await
2
3 async function a(){
4     let something = await asyncFunction();
5     let c = something + 100;
6     return c + 1;
7 }
8
9 // Re-written using promises
10 function a(){
11     return asyncFunction().then((something) => {
12         let c = something + 100;
13         return c;
14     })
15 }
16
17 In the example \ref*{ex:awaitToPromise} we change \texttt{a} from
    ↪ async to synchronous, but we still return a promise which
    ↪ ensures everything using the function \texttt{a} to still get
    ↪ the expected value.

```

Listing 3.16: Example of await to promises

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

In order to allow for the utilization of the users code. We have to identify snippets of the users code that some proposal is applicable to. In order 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.

3.3.2 Matching

In order to identify snippets of code a proposal is applicable to, we use templates of JavaScript. These templates allow for *wildcard* sections where it can match against specific AST nodes. These *wildcard* sections are also used to transfer the context of the code matched into the transformation.

A template containing none of these *wildcards* is matched exactly. This essentially means the match will be a direct code search for snippets where the AST of the users code match the template exactly.

The *wildcards* are written inside a block denoted by `;; WILDCARD ;;`. Each wildcard has to have a DSL identifier, a way of referring to that wildcard in the definition of the transformation, and a wildcard type

Each wildcard has to have some form of type. These types can be node-types inherited from Babels AST definition. This means if you want a wildcard to match any *CallExpression* then that wildcard should be of type *CallExpression*. In order to allow for multiple node-types to match against a single wildcard, JSTQL allows for sum types for wildcards, allowing multiple AST node-types to be allowed to a single wildcard definition.

The wildcard type can also be a custom type with special functionality. Some examples of this is **anyRest**, which allows for the matcher to match it against multiple expressions/statements defined within an AST node as a list. As an example this type could match against any number of statements within a codeblock.

This type definition is also used to define specific behavior the program using this DSL should perform. One example of this can be found in 3.24, where the DSL function *anyRest* is used to allow for any amount of child nodes found together with the wildcard. This means it is feasible to match against any number of function parameters for example.

```
1 let variableName = << expr1: CallExpression | Identifier >>;
```

Listing 3.17: Example of a wildcard

In 3.17 a wildcard section is defined on the right hand side of an assignment statement. This wildcard will match against any AST node classified as a *CallExpression* or an *Identifier*.

3.3.3 JSTQL custom matching types

`anyNExprs` is a custom DSL matching type. This type allows the matcher to match a specific section of the JavaScript template against any number of elements stored within a list on the AST node Object it is currently trying to match. Using this allows for transferring any number of expression from the match into the transformed code. This custom type is used in 3.24.

`anyNStatements` is a custom DSL matching type. This type allows the matcher to match against any number of Statements within a section of JavaScript. This custom type is used in 3.25

3.3.4 Transforming

Observe that once the a matching template has been defined, a definition of transformation has to be created. This transformation has to transfer over the code matched to a wildcard. This means a way to refer to the wildcard is needed. We do this in a very similar manner as defining the wildcard, since we have an internal DSL identifier previously defined in the definition of the matching, all that is needed is to refer to that identifier. This is done with a similar block definition `||` containing the identifier.

```
1 const variableName = <<expr1>>;
```

Listing 3.18: See 3.17 contains identifier `expr1`, and we refer to the same in this example, the only transformation happening here is rewriting `let` to `const`.

3.3.5 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.

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 3.19: 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 pairs. This section of the proposal is defined by the keyword *pair* 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   pair PAIR_NAME {  
2  
3   }
```

Listing 3.20: 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 3.21: 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 keywords 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 3.22: 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   pair 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 3.23: JSTQL definition of a proposal

3.4 Using the JSTQL with an actual syntactic proposal

In this section some examples of how a JSTQL definition of each of the proposals discussed in 3.2 might look. 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 on any reasonably long code definition a user might use this tool with.

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   pair SingleArgument {  
3     applicable to {  
4       <<someFunctionId>>(<<someFunctionParam: Expression |  
5         ↪ Identifier>>);  
6     }  
7     transform to {  
8       <<someFunctionParam>> |> <<someFunctionId>>(%);  
9     }  
10  }  
11  
12  case MultiArgument {
```

```

13     applicable to {
14         <<someFunctionIdent>>(<<firstFunctionParam : Expression | Identifier>>,
15             <<restOfFunctionParams: anyRest>>
16             );
17     }
18
19
20     transform to {
21         <<firstFunctionParam>> |> <<someFunctionIdent>>(%,<
22             ↪ <<restOfFunctionParams>>);
23     }
24 }

```

Listing 3.24: 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. The second pair definition **MultiArgument** will apply to any *CallExpression* with 2 or more arguments. This is because we use the custom JSTQL type **anyRest** that allows to match against any number of elements in an array stored on an AST node.

3.4.2 Do Proposal

The [6, 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     pair arrowFunction{
3         applicable to {
4             () => {
5                 <<blockStatements: anyStatementList>>
6                 return << returnExpr: Expr >>
7             }
8         }
9         transform to {
10             do {
11                 << blockStatements >>
12                 << returnExpr >>
13             }
14         }
15     }
16
17     pair immediatelyInvokedUnnamedFunction {
18         applicable to {
19             function(){
20                 <<blockStatements: anyNStatements>>
21                 return << returnExpr: Expr >>
22             }();
23         }
24     }

```



```
25         transform to {
26             do {
27                 << blockStatements >>
28                 << returnExpr >>
29             }
30         }
31     }
32 }
```

Listing 3.25: Definition of Do 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.

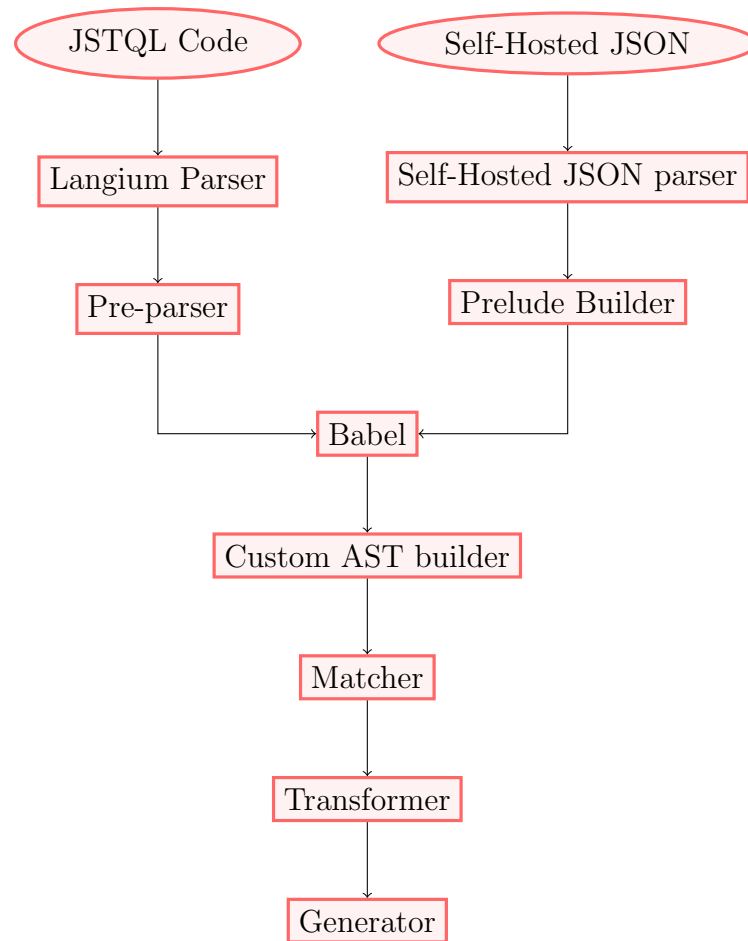


Figure 4.1: Overview of tool architecture

4.2.1 Langium

Langium [4] 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 TypeScript Objects that are later 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 **Pair**.

Pair 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         (pair+=Pair)+
9         "}";
10
11 Pair:
12     "pair" 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) {
6                 accept("error",
7                     ↪ validationResultAplTo.errors.join("\n"), {

```

```

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

```

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

Pre-parsing 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.

In order 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, we pass `temp` on to the function `parseInternalString`

```

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

```

Each wildcard will follow the exact same format, they begin with the opening token <<, followed by what name this variable will be referred by, this variable is called an internal DSL variable and will be used when transferring the matching AST node/s from the users code into the transform template. Following the internal DSL variable a : token is used to show we are moving onto the next part of the wildcard. Following this token is a list of DSL types, either 1 or many, that this wildcard can match against, separated by |. This is a very strict notation on how wildcards can be written, this avoids collision with the already reserved bit-shift operator in JavaScript, as it is highly unlikely any code using the bit-shift operator would fit into this format of a wildcard.

```

1 << Variable_Name : Type1 | Keyword | Type2 | Type3 >>

```

```

1 function parseInternalString(dslString: string) {
2     let [identifier, typeString] = dslString
3       .replace(/\s/g, "").split(":");
4
5     return {
6         identifier,
7         types: typeString.length > 0 ? typeString.split("|") : [],
8     };
9 }

```

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`.

```
1 for (let stmt of stmts) {
2   // Error if not variableDeclaration
3   if (stmt.type == "VariableDeclaration") {
4     // If defined multiple valid types
5     if (stmt.init == "ArrayExpression") {
6       prelude[stmt.name] = []; // Empty array on declared
7       for (let elem of stmt.init.elements) {
8         // Add each type of the array def
9         prelude[stmt.name].push(elem);
10      }
11    } else {
12      // Single valid type
13      prelude[stmt.name] = [stmt.init.name];
14    }
15  }
16 }
```

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

```

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

```

Listing 4.2: Simple definition of a Tree structure in TypeScript

Placing the AST generated by Babel into this structure means utilizing the library [3]Babel Traverse. Babel Traverse uses the [7]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.

[3]Babel Traverse uses the [7]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 individual AST nodes to each other, and how wildcard matching is performed.

Matching singular Expression

The method of writing the `applicable to` section using a singular expression is by far the most versatile way of defining a proposal, this is simply because there will be a much higher chance of discovering matches with a template that is as generic as possible. Therefore only matching against a single expression ensures the matcher tries to perform a match at every level of the AST.

Matching 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 sliding window of the amount of statements at every *BlockStatement*, as that is the only placement Statements can reside in JavaScript.

4.6 Transforming

4.7 Generating

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.

Bibliography

- [1] Babel · Babel, May 2024. [Online; accessed 10. May 2024].
- [2] @babel/generator · Babel, May 2024. [Online; accessed 12. May 2024].
- [3] @babel/traverse · Babel, May 2024. [Online; accessed 12. May 2024].
- [4] Langium, April 2024. [Online; accessed 10. May 2024].
- [5] proposal-discard-binding, April 2024. [Online; accessed 25. Apr. 2024].
- [6] proposal-do-expressions, May 2024. [Online; accessed 2. May 2024].
- [7] 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 System.out.println("Hello Mars");
```

Listing A.1: Source code of something