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 congue ius at, pro suas meis habeo no.

Acknowledgements

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

`Your name`
Tuesday 14th May, 2024

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Intro goes here

1.1 Background

1.1.1 Figures

Chapter 2

Background

2.1 Proposals

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

Chapter 3

Collecting User Feedback for Syntactic Proposals

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

3.1 The core idea

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 [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.

```
// previous
let temp = { a:1, b:2, c:3, d:4 };
let a = temp.a;
let b = temp.b;

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

Listing 3.2: Example of unpacking Object

```
// previous
let tempArr = [ 0, 2, 3, 4 ];
let a = tempArr[0]; // 0
let b = tempArr[1] // 2

//unpacking
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 is 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.

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

Listing 3.4: Example discard binding with variable discard

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

Listing 3.5: Example Object binding and assignment pattern

```
function* gen() {
  for (let i = 0; i < Number.MAX_SAFE_INTEGER; i++) {
    console.log(i);
    yield i;
  }
}

const iter = gen();
const [a, , ] = iter;
```

```
// prints:
//  0
//  1

const [a, void] = iter; // author intends to consume two elements
// vs.
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

```
// project an array values into an array of indices
const indices = array.map((void, i) => i);

// passing a callback to `Map.prototype.forEach` that only cares about
// keys
map.forEach((void, key) => { });

// watching a specific known file for events
fs.watchFile(fileName, (void, kind) => { });

// ignoring unused parameters in an overridden method
class Logger {
  log(timestamp, message) {
    console.log(`${timestamp}: ${message}`);
  }
}

class CustomLogger extends Logger {
  log(void, message) {
    // this logger doesn't use the timestamp...
  }
}

// Can also be utilized for more trivial examples where _ becomes
// cumbersome due to multiple discarded parameters.
doWork((_, a, _1, _2, b) => {});
// vs.
doWork((void, a, void, void, b) => {
});
```

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.

```
var [void] = x;          // via: BindingPattern :: `void`
var {x:void};            // via: BindingPattern :: `void`

let [void] = x;          // via: BindingPattern :: `void`
let {x:void};            // via: BindingPattern :: `void`

const [void] = x;        // via: BindingPattern :: `void`
const {x:void} = x;      // via: BindingPattern :: `void`

function f(void) {}      // via: BindingPattern :: `void`
function f([void]) {}    // via: BindingPattern :: `void`
function f({x:void}) {} // via: BindingPattern :: `void`

((void) => {});          // via: BindingPattern :: `void`
(([void]) => {});        // via: BindingPattern :: `void`
(({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

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

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

Listing 3.14: Example from ramda.js

3.2.6 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 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.

,

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

// Using a immediately invoked function
let x = function(){
    let tmp = f();
    return tmp + tmp + 1;
}();

// Using do expression
```

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

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.

```
// Code containing await

async function a(){
    let something = await asyncFunction();
    let c = something + 100;
    return c + 1;
}

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

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.

```
proposal PROPOSAL_NAME {
    pair PAIR_NAME {
        applicable to {

        }
        transform to {

        }
    }
    pair PAIR_NAME {
        applicable to .....
    }

    pair ....
}
```

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.

```
proposal Pipeline{
    pair SingleArgument {
        applicable to {
            <<someFunctionIdent>>(<<someFunctionParam: Expression |
                ↪ Identifier>>);
        }

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

    case MultiArgument {
```

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

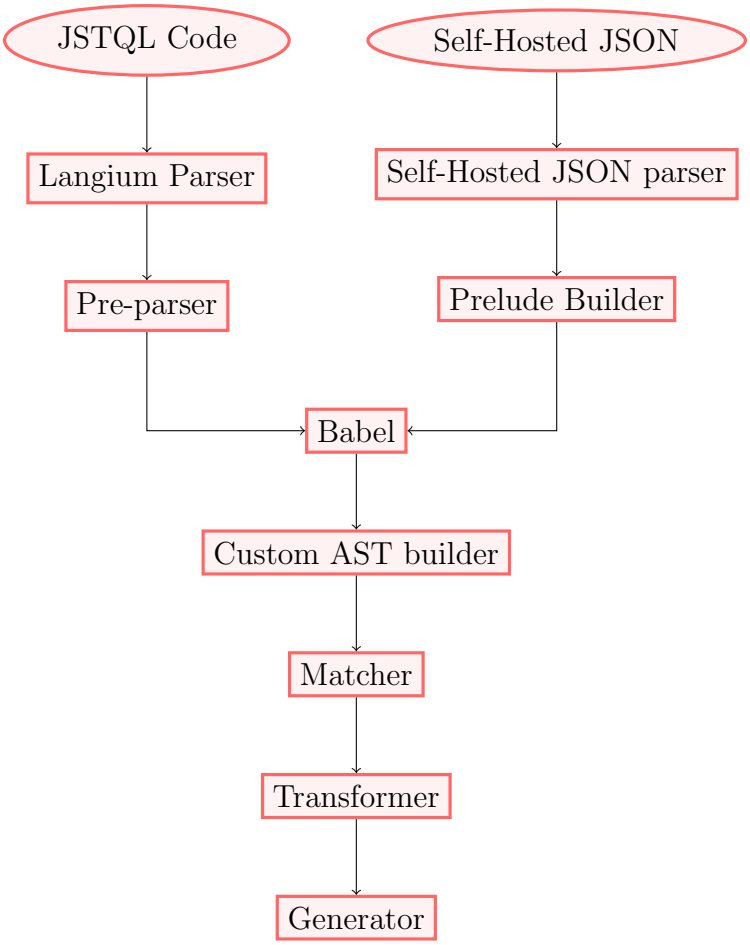

Figure 4.1: Overview of tool architecture

4.2.1 Langium

Langium [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 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 do 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):
           ↪ void {
3          try {
4              if (validationResultAplTo.errors.length != 0) {
5                  accept("error",
                       ↪ validationResultAplTo.errors.join("\n"), {
```

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

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`

```
export function parseInternal(code: string): InternalParseResult {
    let cleanedJS = "";
    let temp = "";
    let flag = false;
    let prelude: InternalDSLVariable = {};

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

```
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 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.

```
for (let stmt of stmts) {
    // Error if not variableDeclaration
    if (stmt.type == "VariableDeclaration") {
        // If defined multiple valid types
        if (stmt.init == "ArrayExpression") {
            prelude[stmt.name] = []; // Empty array on declared
            for (let elem of stmt.init.elements) {
                // Add each type of the array def
                prelude[stmt.name].push(elem);
            }
        } else {
            // Single valid type
            prelude[stmt.name] = [stmt.init.name];
        }
    }
}
```

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 and 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.

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

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

Listing 4.2: 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 [10]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 [10]visitor pattern to visit each node of the AST in a *depth first* manner, the idea of this pattern is one implements a *visitor* for each of the nodes in the AST and when a specific node is visited, that visitor is then used. In the case of transferring the AST into our simple tree structure we simply have to use the same visitor for all nodes, and place that node into the tree.

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

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

            if (last == null) {
                first = node;
            }
            last = node;
```

```
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 it the most important step, as if no matching code is found the tool will do no transformations. Finding the matches will depend entirely on how well the definition of the proposal is written, and how well the proposal actually can be defined within the confines of JSTQL . In this chapter we will discuss how matching individual AST nodes to each other, and how wildcard matching is performed.

Matching singular Expression/Statement

The method of writing the `applicable to` section using a singular simple expression/statement 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 and simple as possible. Therefore only matching against a single expression/statement ensures the matcher tries to perform a match at every level of the AST. This of course relies on that the expression/statement used to match against is as simple as possible, in order to easily find matches in user code.

When the template of `applicable to` is a single expression, parsing it with [1]Babel will produce an AST containing a single node of the *Program* body, namely an *ExpressionStatement*. The reason for this is Babel will treat an expression not bound to a Statement as an ExpressionStatement, this is a problem because we cannot use an ExpressionStatement to match against any expression that are already part of some statement, for example a VariableDeclaration. In order to solve this we have to remove the AST node ExpressionStatement and only do the match against its Sub-Expression. This of course is not a requirement for matching against a single statement, as then it is expected that the user code will be *similar*.

In order to determine if we are matching against an expression or a statement we can verify the type of the first statement in the program body of the AST generated when using [2]babel generate `applicable to`. If the first statement is of type `ExpressionStatement`, we know the matcher is supposed to match against an expression, and we have to remove the `ExpressionStatement` AST node from the tree used for matching. This is done by simply using `applicableTo.children[0].children[0].element` as the AST to match the users code against.

```
if (applicableTo.children[0].element.type === "ExpressionStatement") {
            let matcher = new Matcher(
                internals,
                applicableTo.children[0].children[0].element
            );
            matcher.singleExprMatcher(
                code,
                applicableTo.children[0].children[0]
            );
            return matcher.matches;
}
```

In Figure 4.3 is the full definition of the expression matcher, it is based upon a Depth-First Search in order to perform matching, and searches from the top of the code definition. In the first part of the function, if we are currently at the first AST node of `applicable to`, we recursively try to match every child of the current code node against the full `applicable to` definition, this ensures no matches are undiscovered while not performing unnecessary searching using a partial `applicable to` definition. If a child node returns one or more matches, they are placed into a temporary array, once all children have been searched, the partial matches are filtered out and all full matches are stored. 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 collisions exist.

From line 22 in Figure 4.3 we are evaluating if the current node is a full match. First the current node is checked against the node of `applicable to`, if said node is a match, and it contains enough children to be matched against `applicable to`, we know we can perform the search. Since the list of elements in the child arrays are ordered, we can do a search on each of the nodes by using the same index in the `node.children` array. If any of the children do not return as a match, we know this is not a full match and perform an early return of `undefined`. Only if at least all the children of `applicable to` return as a match can we determine this is a match. That match is then returned, and is stored by the caller of this current iteration of the recursion.

```
singleExprMatcher(
        code: TreeNode<t.Node>,
        aplTo: TreeNode<t.Node>
    ): TreeNode<PairedNodes> | undefined {
        // If we are at start of ApplicableTo, start a new search on
           ↪ each of the child nodes
        if (aplTo.element === this.aplToFull) {
            // Perform a new search on all child nodes before trying
               ↪ to verify current node
            let temp = [];
            // If any matches bubble up from child nodes, we have to
               ↪ store it
            for (let code_child of code.children) {
                let maybeChildMatch =
                    ↪ this.singleExprMatcher(code_child, aplTo);
                if (maybeChildMatch) {
                    temp.push(maybeChildMatch);
                }
            }
            this.matches.push(...temp);
        }
```

```
18
19
20          let curMatches = this.checkCodeNode(code.element,
                ↪ aplTo.element);
21          curMatches =
22              curMatches && code.children.length >=
                    ↪ aplTo.children.length;
23          if (!curMatches) {
24              return;
25          }
26          // At this point current does match
27          // Perform a search on each of the children of both AplTo and
                ↪ Code.
28          let pairedCurrent: TreeNode<PairedNodes> = new TreeNode(null, {
29              codeNode: code.element,
30              aplToNode: aplTo.element,
31          });
32          for (let i = 0; i < aplTo.children.length; i++) {
33              let childSearch = this.singleExprMatcher(
34                  code.children[i],
35                  aplTo.children[i]
36              );
37              if (childSearch === undefined) {
38                  // Failed to get a full match, so early return here
39                  return;
40              }
41              childSearch.parent = pairedCurrent;
42              pairedCurrent.children.push(childSearch);
43          }
44
45          // If we are here, a full match has been found
46          return pairedCurrent;
47      }
```

Listing 4.3: Recursive definition of expression matcher

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

```
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 [9]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.

```
    multiStatementMatcher(code: TreeNode<t.Node>, aplTo:
        ↪ TreeNode<t.Node>) {
        if (
            code.element.type === "Program" ||
            code.element.type === "BlockStatement"
        ) {
            this.matchMultiHead(code.children, aplTo.children);
        }
        // Recursively search the tree for Program || BlockStatement
        for (let code_child of code.children) {
            this.multiStatementMatcher(code_child, aplTo);
        }
    }
```

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 [9]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 each and 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, which is quite similar to the second part of matching a single Statement/Expr, however the main 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.

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.

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

4.6 Transforming

To perform the transformation and replacement on each of the matches, the tool uses the function `transformer`, this function takes all the matches found with the matcher,

the code from `transform to` parsed by [3]babel/parser and built into our custom tree structure, the original user code AST, and the full direct output of `transform to` parsed by babel/parser.

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

The first step of transforming is done by taking the wildcards used by the matcher and place them into the AST generated from `transform to`, in our case that means searching `transform to` and the paired output of the matcher for *Identifiers* with the same name, and inserting the AST node matched against that specific `applicable to` node into the tree of `transform to` we are transforming. The version of `transform to` we are applying the transformation to is not in the custom tree structure used by the matcher, therefore we have to use [4]babel/traverse to traverse it with a custom [10]visitor only applying to AST nodes of type *Identifier*. Once the correct identifier is found while traversing, the node is simply replaced with the node the wildcard was matched against in the matcher.

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 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. Now we simply have to remove the next n-1 Statements, where n is the length of the list of Statements in the current match.

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] 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.

[10] 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