Endringer fra forrige møte

- Skrevet om historien til JavaScript. Section 2.2.1 on page 13.
- Skrev litt om "Code Generation". Section 5.6 on page 35.
- Skrevet seksjon "What better fits PT?". Section 7.1.5 on page 47.
- Skrevet seksjon "More General Functions/Classes". Section 7.1.2 on page 47

PTS (Package Template Script)

An Implementation of Package Templates in TypeScript

Petter Sæther Moen

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

PTS (Package Template Script)

An Implementation of Package Templates in TypeScript

Petter Sæther Moen

PTS (Package Template Script)

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Contents

Abstract

Acknowledgements

Part I

Introduction

Chapter 1

Introduction

1.1 What is PT?

1.2 Purpose of Implementing PT in TS

1.3 Structure of Thesis

Chapter 2

Background

2.1 Package Templates

Krogdahl proposed Generic Packages in 2001, which is a language mechanism aimed at "large scale code reuse in object oriented languages" [7]. The idea behind this mechanism is to make modules of classes, called *packages*, that could later be imported and instantiated. This would make "textual copies" of the package body, and would also allow for further expanding the classes of the packages. Modularizing through Generic Packages made the programming more flexible as you would easily be able to write modules with a certain functionality and be able to later import it several times when there is a need for the functionality.

Generic Packages was later extended, and the mechanism is now called Package Templates (while the textual program modules themselves are simply called templates). The system is not fully implemented and there exists a number of proposals for extending it.

2.1.1 Syntax

In this section we will look at the syntax of Package Templates (further referred to as *PT*) in a Java-like language as proposed in [8].

Defining packages and package templates

Packages are defined by a set of classes similar to a normal Java package. Package templates (later just templates for short), are defined in a similar manner except for using the keyword *template*. Listing 1 on the following page shows an example of defining packages and templates. The contents of a package can be used as you would with a normal Java package.

Instantiating templates

Instantiating is what really makes PT useful. When defining packages and templates, PT allows for including already defined templates through instantiating. Instantiation is done inside the body of a package (or a template) with the use of an `inst`-clause. Instantiating a template will make

```
package P {
    interface I { ... }
    class A extends I { ... }
}

template T {
    class B { ... }
}
```

Listing 1: Defining a package P and a template T

"textual copies" of the classes, interfaces and enums from the instantiated template and insert them replacing the instantiation statement at compile time. Note that the template itself still exists and that it can be instantiated again in the same program. Listing 2 on the next page shows an example of instantiating a template inside a package. The resulting package `P` will then have the classes `A` and `B` from template `T` and its own class `C`.

Renaming

During instantiation it is possible to rename classes (as well as interfaces and enums) and class attributes, both fields and methods. Renaming is a part of the instantiation of templates, and will only affect the copy made for this instantiation, and it is done for the copy before it replaces the `inst`-statement. Renaming is denoted by an optional `with`-clause at the end of the `inst`-statement. In the `with`-clause one can rename classes using the following fat arrow syntax, `A => B`, where class `A` is renamed to `B`, and rename class attributes with a similar arrow syntax, `i -> j`, where the attribute `i` is renamed to `j`. For method renaming you have to give the signature of the method so that it is possible to distinguish between overloaded versions, i.e. `m1(int) -> m2(int)`. On a more technical level the compiler will find the class or attribute declaration that is going to be renamed, and then find all name occurrences bound to this declaration and rename these.

Field renaming comes after the class renaming enclosed by a set of parentheses. Renaming classes will also affect the signatures of any methods using this class. Listing 3 on page 8 shows an often used example of renaming, where a graph template is renamed to better fit a domain, in this case a road map. When renaming the class `Node` the signature of the methods in `Edge` using this `Node` was also changed to reflect this, i.e. the method `Node getNodeFrom()` would become `City getNodeFrom()` with the class rename, and `City getStartingCity()` with the method renaming.

Renaming makes it possible to instantiate templates with conflicting names of classes, or even instantiate the same templates multiple times. Listing 4 on page 9 shows an example of this where we instantiate the same template T twice without any issues.

```
// Before compile time instantiation of T
template T {
    class A { ... }
    class B { ... }
}

package P {
    inst T;
    class C {
        A a;
        B b() {
            ...
        }
    }
}

// After compile time instantiation of T
package P {
    class A { ... }
    class B { ... }
    class C { ... }
}
```

Listing 2: Instantiating template T in package P. Note that class C can reference class A and B as if they were defined in the same package, which they essentially are after the instantiation.

```
template Graph {
    class Node {
        ...
    }

    class Edge {
        Node getNodeFrom() { ... }
        Node getNodeTo() { ... }
    }

    class Graph {
        ...
    }
}

package RoadMap {
    ...
    inst Graph with
        Node => City,
        Edge => Road
            (getNodeFrom() -> getStartingCity(),
            getNodeTo() -> getDestinationCity()),
        Graph => RoadSystem;
    ...
}
```

Listing 3: Example of renaming classes during instantiation. This could be used to make the classes fit the domain of the project better.

```
template T {
    class A {
        void m() { ... }
    }
}

package P {
    inst T;
    inst T with A => B;
}

// package P after compile time instantiation and renaming
package P {
    class A {
        void m() { ... }
    }

    class B {
        void m() { ... }
    }
}
```

Listing 4: Example of instantiating the same template twice solved by renaming.

```
template T {
    class A {
        void someMethod() { ... }
    }
}

package P {
    inst T;
    addto A {
        int i;
        void someOtherMethod() { ... }
    }
}
```

Listing 5: Adding new attributes to the instantiated class A in package P

Additions to a template

When instantiating a template you can also add attributes to the classes of the template, as well as extending the class' implemented interfaces and this will only apply to the current copy. These additions are written inside an `addto`-clause. Extending the class with additional attributes is done in the body of the clause, like you would in a normal Java class. If an addition has the same signature as an already existing method from the instantiated template class, then the addition will override the existing method, similarly to traditional inheritance. Listing 5 shows an example of adding attributes to an instantiated class. The resulting class `A` in package `P` would have the field `i` and the methods `someMethod` and `someOtherMethod`.

It is also possible to extend the list of implemented interfaces of a class by suffixing the `addto`-clause with a `implements`-clause containing the list of implementing interfaces. Having the possibility to add implementing interfaces to classes makes working with PT easier and enables the programmer to re-use template classes to a much larger degree. This feature's use is easier explained through an example.

Say we have implemented some class that will deal with logging. This class can log the state of a class given that the class implements some interface `Loggable`. If we want to be able to log the state of our Graph implementation, from 3 on page 8, then the Graph class would need to implement the `Loggable` interface. We can't do this at the declaration of the Graph template, as we do not have access to the interface at the time of declaration. By using `addto` however we are able to add the `Loggable` interface and the `log` method to the Graph class. You can also achieve the same functionality through class merging, which we will look at in section 2.1.1 on page 12.

The extension of classes using the `addto`-clause is done independently for each class, ignoring any inheritance-patterns. What this means is that any additions made to a superclass will also be reflected in its subclasses,

```
template Logger {
    interface Loggable {
        String log();
    }

    class Logger {
        void log(Loggable loggable) {
            ...
        }
    }
}

package P {
    inst Graph;
    inst Logger;
    addto Graph implements Loggable {
        String log() {
            ...
        }
    }
}
```

Listing 6: Adding the `Loggable` interface to the Graph class from listing 3 on page 8, making it compatible with our logger implementation.

```
template T1 {
    class A {
        ...
    }
}

template T2 {
    class B {
        ...
    }
}

package P {
    inst T1 with A;
    inst T2 with B => A;
}
```

Listing 7: Instantiation with class merging through renaming

we do not have to worry about how the class is being used. If there is a class `A` with a subclass `B`, they can both get extensions independently of each other. Any extensions made to class `A` would of course still be inherited to class `B`, as with normal Java inheritance.

Merging classes

If two or more classes in the same or in different instantiations share the same name they will be merged into one class. Through this mechanism PT achieves a form of multiple inheritance. This form of inheritance is different from what you would normally find in Java, it acts more like *mixins*[1]. Here the subclasses are unaware of its superclasses, as this form of inheritance is merely merging of textual copies at compile time. We call this kind of inheritance static multiple inheritance.

If two classes don't share the same name, it is still possible to force a merge through renaming them to the same name. In listing 7 we see an example of renaming class `B` to `A` in order to merge them under the class name `A`. Merging the classes would lead to having a single class `A` with the attributes from both classes. The two classes `A` and `B`, from templates `T1` and `T2` respectively, no longer exists in package `P`, but have formed the new class `A`, which is a union of both. Any pointers typed with the old `A` or `B` will now be pointing to the new merged class `A`.

[1] A mixin is a language feature which enables you to inject code into a class.

2.2 TypeScript

Before we look at what TypeScript is we first need to understand JavaScript and the JavaScript ecosystem.

2.2.1 JavaScript

Back in the mid 90s web pages could only be static, however there was a desire to remove this limitation and make the web a more interactive platform, as it became increasingly more accessible to non-technical users. In order to remove this limitation, Netscape, with its Netscape Navigator browser, partnered up with Sun in order to bring the Java platform to the browser, and hired Brendan Eich to create a scripting language for the web. Eich was tasked to create a Scheme like language with syntax similar to Java, and the language was intended to be a companion language to Java. The language when it first released was called LiveScript, however it was later renamed to JavaScript. This has been characterized as a marketing ploy by Netscape in order to give the impression that it was a Java spin-off.

Microsoft, with its Internet Explorer, adopted the language and named it JScript. During this time Microsoft and Netscape would both ship new features to the language in order to increase the popularity for their respective browsers. Because of this war between browsers the language was later handed over to ECMA International as a starting point for a standard specification for the language. This ensured that users would get the same experience across different browsers, making the web more accessible [23].

A web page generally consists of three layers of technologies. The first layer is HTML, which is the markup language that is used to structure the web page. Second is CSS which gives our structured documents styling such as background colors and positioning. The third and final layer is JavaScript which enables web pages to have dynamic content. Whenever you visit a website that isn't just static information, but instead might have timely content updates, interactive maps, etc., then JavaScript is most likely involved [13].

JavaScript, or more precisely ECMAScript, is the programming language of the web. It is a language that conforms to the ECMAScript specification. ECMAScript is simply a JavaScript standard, created by Ecma International, made to ensure interoperability across different browsers. There is no official runtime or compiler for JavaScript as it is up to each browser to implement the languages runtime. When we create a JavaScript program/script for a web page we don't compile it and transfer a binary or bytecode file for the web page to execute, instead the browser takes the raw source code and interprets it[2].

JavaScript is a multi-paradigm language with dynamic typing...

[2]On a more technical level, JavaScript is generally just-in-time compiled in the browser.

ECMAScript Versions

ECMAScript versions are generally released on a yearly basis. This release is in the form of a detailed document describing the language, ECMAScript, at the time of release. New versions will most likely include some additions to the language, but never any breaking changes[3]. This is because the developer will not be able to control the environment on which the code will be executed[4] and you can not be sure which ECMAScript version the client browser will be using. Because of this lack of control over the runtime environment it is crucial that any pre-existing language features don't have breaking changes between versions.

Backwards Compatibility

With new ECMAScript versions comes new features, and it is up to each browser to implement these changes. As we mentioned earlier, we do not transfer a binary to the client browser, we transfer the source code. So when a JavaScript script uses a new ECMAScript feature it is not guaranteed to work with every client browser, since a lot of users might have older browsers installed, or the team behind the browser has not implemented the language feature yet. To deal with this a common practice in JavaScript development is to first transpile the source code before using it in a production environment. This transpilation step takes the source code and transpiles it into an older ECMAScript version. In doing this you ensure that more client browser will be able to run the script. This will rewrite the new language features, and often replace them with a function, called a *polyfill*. You can think of a polyfill as an implementation of a new language feature.

Some popular transpilers for JS to JS transpilation are Webpack and Babel, but you could also use the TypeScript compiler for this.

Node.js

As of the time of writing there are mainly two ways to execute JavaScript. You can run the program in the browser, as it was intended, or you can use a JavaScript runtime that runs on the backend, outside of the browser. Node.js (henceforth simply referred to as Node) is the mainstream solution for the latter. Node is a JavaScript runtime built on the JavaScript engine, V8, used by Chrome. It is independent of the browser and can be run through a *CLI* (Command-Line Interface). One major difference from the browser runtimes is that Node also supplies some libraries for IO, such as access to the file system and the possibility to listen to HTTP requests and WebSocket events. This makes Node a good choice for writing networking applications for instance.

[3] There has been occasions where there has been minor breaking changes between ECMAScript versions, but these changes happen very rarely and the affected areas are often insignificant.

[4] It is possible to ship chunks of code called *polyfills* with your code in order to regain some control over the environment. We will go more in-depth on this in section 2.2.1.

We will be using the Node runtime for our compiler since it gives us access to the file system, as well as enabling the compiler to be executed through a CLI, as is the norm for most compilers. The compiler will still also be available as a library.

2.2.2 What is TypeScript?

TypeScript is a superset of JavaScript. The language builds on JavaScript with the additions of static type definitions [20].

All valid JavaScript programs are also valid TypeScript programs.

Part II

The project

Chapter 3

The Language - PTS

While the majority of the semantics of PTS will be equal to that of PTj, we will have to change some syntax.

3.1 Syntax

For the implementation of PT we need syntax for the following:

- Defining packages (`package` in PTj)
- Defining templates (`template` in PTj)
- Instantiating templates (`inst` in PTj)
- Specifying renaming for an instantiation (`with` in PTj)
- Renaming classes (`=>` in PTj)
- Renaming class attributes (`->` in PTj)
- Additions to classes (`addto` in PTj)

`template`, `addto`, and `inst` are all not in use nor reserved in the ECMAScript standard or in TypeScript, and can therefore be used in Package Template Script without any issues.

The keyword `package` in TS/ES is, as of yet, not in use, however the ECMAScript standard has reserved it for future use. In order to "future proof" our implementation we should avoid using this reserved keyword, as it could have some conflicts with a potential future implementation of packages in ECMAScript. It could also be beneficial to not share the keyword in order to avoid creating confusion between the future ES packages and PT Packages. `module` is also a keyword that could be used to describe a PT package, however this is already used in the ES standard, and should therefore also be avoided in order to avoid confusion. We will therefore use `pack` instead.

Renaming in PTj uses `=>`(fat-arrow) for renaming classes, and `->`(thin-arrow) for renaming class attributes. PTj, for historical purposes, used two

different operators for renaming classes and methods, however in more recent PT implementations, such as [6], a single common operator is used for both. We will do as the latter, and only use a single common operator for renaming. Another reason for rethinking the renaming syntax is the fact that the =>(fat-arrow) operator is already in use in arrow functions [2], and reusing it for renaming could potentially produce an ambiguous grammar, or the very least be confusing to the programmer. JavaScript currently supports renaming of destructured attributes using the :(colon) operator and aliasing imports using the keyword `as`. We could opt to choose one of these for renaming in PTS as well, however in order to keep the concepts separated, as well as making the syntax more familiar for Package Template users, we will go for the ->(thin-arrow) operator.

The `with` keyword is currently in use in JavaScript for `with`-statements [10]. With it being a statement, we could still use it and not end up with an ambiguous grammar, however as with previous keywords, we will avoid using it in order to minimize concept confusion. Instead of this we will contain our instantiation renamings inside a block-scope (`{ }`). Field renamings for a class will remain the same as in PTj, being enclosed in parentheses (`( )`).

Another change we will make to renaming is to remove the requirement of having to specify the signature of the method being renamed. This was necessary in PTj as Java supports overloading, which means that several methods could have the same name, or a method and a field. Method overloading is not supported in JavaScript/TypeScript, and we do therefore not need this constraint.

3.2 The PTS Grammar

Now that we have made our choices for keywords and operators we can look at the grammar of the language.

PTS is an extension of TypeScript, and the grammar is therefore also an extension of the TypeScript grammar. There is no published official TypeScript grammar (other than interpreting it from the implementation of the TypeScript compiler), however up until recently there used to be a TypeScript specification [12]. This TypeScript specification was deprecated as it proved a to great a task to keep updated with the ever-changing nature of the language. However, most of the essential parts are still the same. The PTS grammar is therefore based on the TypeScript specification, and on the ESTree Specification [14].

In figure 3.1 on the next page we can see the PTS BNF grammar. This is not the full grammar for PTS, as I have only included any additions or changes to the original TypeScript/JavaScript grammars. More specifically the non-terminal ⟨declaration⟩ is an extension of the original grammar, where we also include package and template declarations as legal declarations. The productions for non-terminals ⟨id⟩, ⟨class declaration⟩, ⟨interface declaration⟩, and ⟨class body⟩ are also from the original grammar.

⟨declaration⟩ ⊨ ... | ⟨package declaration⟩ | ⟨template declaration⟩
⟨package declaration⟩ ⊨ `pack` ⟨id⟩ ⟨PT body⟩
⟨template declaration⟩ ⊨ `template` ⟨id⟩ ⟨PT body⟩
⟨PT body⟩ ⊨ `{` ⟨PT body decls⟩ `}`
⟨PT body decls⟩ ⊨ ⟨PT body decls⟩ ⟨PT body decl⟩ | λ
⟨PT body decl⟩ ⊨ ⟨inst statement⟩ | ⟨addto statement⟩ |
⟨class declaration⟩ | ⟨interface declaration⟩
⟨inst statement⟩ ⊨ `inst` ⟨id⟩ ⟨inst rename block⟩
⟨inst rename block⟩ ⊨ `{` ⟨class renamings⟩ `}` | λ
⟨class renamings⟩ ⊨ ⟨class rename⟩ | ⟨class rename⟩ , ⟨class renamings⟩
⟨class rename⟩ ⊨ ⟨rename⟩ ⟨attribute rename block⟩
⟨attribute rename block⟩ ⊨ `(` ⟨attribute renamings⟩ `)` | λ
⟨attribute renamings⟩ ⊨ ⟨rename⟩ | ⟨rename⟩ , ⟨attribute renamings⟩
⟨rename⟩ ⊨ ⟨id⟩ `->` ⟨id⟩
⟨addto statement⟩ ⊨ `addto` ⟨id⟩ ⟨addto heritage⟩ ⟨class body⟩
⟨addto heritage⟩ ⊨ ⟨class heritage⟩ | λ

Figure 3.1: BNF grammar for PTS. The non-terminals ⟨declaration⟩, ⟨id⟩, ⟨class declaration⟩, ⟨interface declaration⟩, and ⟨class body⟩ are productions from the TypeScript grammar. The ellipsis in the declaration production means that we extend the TypeScript production with some extra choices.
Legend: Non-terminals are surrounded by ⟨angle brackets⟩. Terminals are in `typewriter font`. Meta-symbols are in regular font.

3.3 Example programs

Figure 8 on the facing page.

```
// PTS
template T {
    class A {
        function f() : string {
            ...
        }
    }
}

pack P {
    inst T { A -> A (f -> g) };
    addto A {
        i : number = 0;
    }
}

// PTj
template T {
    class A {
        String f() {
            ...
        }
    }
}

package P {
    inst T with A => A (f() -> g());
    addto A {
        int i = 0;
    }
}
```

Listing 8: An example program with instantiation, renaming, and addition-classes in PTS vs. PTj

Chapter 4

Planning the Project

Before we start the implementation of our language we first need to do some planning. We know we are going to be creating a programming language, a superset of TypeScript with the addition of Package Templates. However, we might want to look at if creating a superset of TypeScript is the way to go, or if keeping it simple and extending JavaScript is a better call. There are a lot of approaches we can take for implementing our language, so we will have map out the requirements for our approach. Lastly we will have to look at all the different approaches we can take, and discuss the pros and cons of each approach.

This planning phase is crucial for the success of the project, as starting off on the wrong approach for the wrong language would set us back immensely.

4.1 TypeScript vs. JavaScript

When extending TypeScript you might be asking yourself if it is truly necessary, is it better to keep it simple and just extend JavaScript instead? This is something we need to find out before going any further with the planning of our project.

4.1.1 Typechecking templates

One of the requirements of PT is that it should be possible to type-check each template separately. There is no easy way to type-check JavaScript code without executing it and looking for runtime errors. Even if some JavaScript program successfully executes without throwing any errors, we can still not conclude that the program does not contain any type errors. TypeScript on the other hand, with the language being statically typed, we can, at least to a much larger extent, verify if some piece of code is type secure. Because of this trait TypeScript is the better candidate for our language.

Now it should be noted that due to TypeScripts type system being unsound one could argue that this requirement of PT is not met. While this is true it still outperforms JavaScript on this remark, and we will

later in section 6.2 on page 44 discuss more in-depth to what extent this requirement is met.

4.2 What Do We Need?

There are a lot of approaches one can take when working with TypeScript, however due to the nature of this project there are some restrictions we have to abide by. Our approach should allow the following:

- The ability to add custom syntax (access to the tokenizer / parser)
- Enable us to do semantic analysis.

In addition to these we would also like to look for some other desirable traits for our implementation:

- Loosely coupled implementation (So that new versions of typescript not necessarily breaks our implementation).
- Mer?

4.3 Choosing the right approach

Before jumping into a project of this magnitude it is important to find out what approach to use. The goal of this project is to extend TypeScript with the Package Templates language mechanism, this could be achieved by one of the following methods:

- Making a fork of the TypeScript compiler
- Making a preprocessor for the TypeScript compiler
- Making a compiler plugin / transform
- Making a custom compiler from scratch

4.3.1 Preprocessor for the TypeScript Compiler

More work than ex plugin / transformer.

4.3.2 TypeScript Compiler Plugin / Transform

At the time of writing the official TypeScript compiler does not support compile time plugins. The plugins for the TypeScript compiler is, as the TypeScript compiler wiki specifies, "for changing the editing experience only" [11]. However, there are alternatives that do enable compile time plugins / transformers;

- ts-loader [21], for the webpack ecosystem

- Awesome Typescript Loader [16], for the webpack ecosystem. Deprecated
- ts-node [22], REPL / runtime
- ttypescript [4], TypeScript tool TODO: Les mer på dette

Unfortunately ts-loader, Awesome Typescript Loader and ts-node does not support adding custom syntax, as it only transforms the AST produced by the TypeScript compiler. Because of this they are not a viable option for our use-case and will therefore be discarded.

4.3.3 Babel plugin

Babel isn't strictly for TypeScript, but for JavaScript as a whole, however we could write our plugin to be dependent on the TypeScript transformation plugin.

Making a Babel plugin will make it very accessible as most web-projects use Babel, and the upkeep is cheap, as plugins are loosely coupled with the core.

In order for a Babel plugin to support custom syntax it has to provide a custom parser, a fork of the Babel parser. Through this we can extend the TypeScript syntax with our syntax for PT. This is all hidden away from the user, as this custom parser is a dependency of our Babel plugin.

Seeing as we have to make a fork of the parser in order to solve our problem, the upkeep will not be as cheap as first anticipated. However, being able to have most of the logic loosely coupled with the compiler core it will still make it easier to keep updated than through a fork of the TypeScript compiler.

4.3.4 TypeScript Compiler Fork

Possible, however not as accessible as other alternatives and will make upkeep expensive.

The TypeScript compiler is a monolith. It has about 2.5 million lines of code, and therefore has a quite steep learning curve to get into. If we were to go with this route it would be quite hard to keep up with the TypeScript updates, as updates to the compiler might break our implementation. However, as we have seen, going the plugin / transform route also requires us to fork the underlying compiler and make changes to it, however with the majority of the implementation being loosely coupled it would still make it easier to keep up-to-date. That being said it will probably be a lot easier to do semantic analysis in a fork of the TypeScript compiler vs in a plugin / transform.

Chapter 5

Implementation

In this chapter we are going to look at the implementation of PTS.

5.1 Compiler Architecture

5.2 Lexer and Parser

5.2.1 Parser Generator

There are a lot of parser generators out there, but there is no one-size-fits-all solution. In order to navigate through the sea of options we need to set some requirements in terms of functionality, so that we can more easily find the right tool for the task.

As we talked about in section 4.2 on page 26, we set ourselves the goal to find an approach that would allow us to create an implementation that was loosely coupled with TypeScript. TypeScript is a large language that is constantly updated, and is getting new features fairly often. Because of this one of the requirements for our choice of parser generator is the possibility for extending grammars. This is important because we want to keep our grammar loosely coupled with the TypeScript grammar, and don't want to

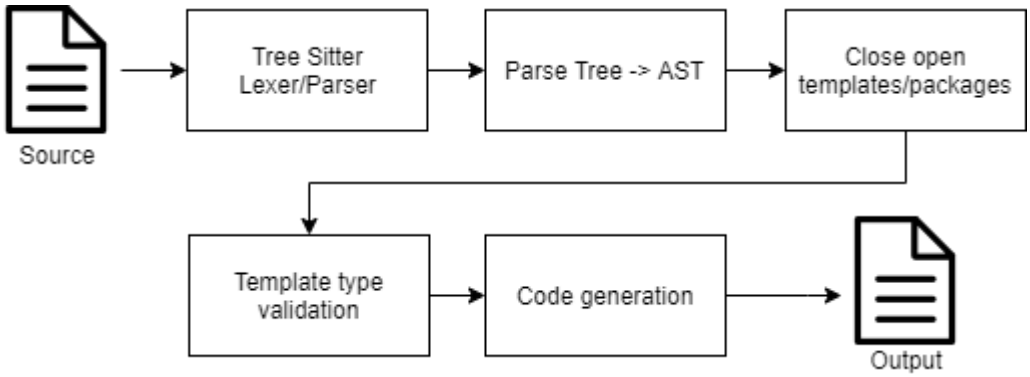

Figure 5.1: Overview of the compiler

be forced to rewrite the entire TypeScript grammar, as well as keeping it up-to-date.

Because our language will be extending TypeScript we would like to utilize the TypeScript compiler as much as we can. The TypeScript compiler will help us perform the type-checking for our compiler, as well as producing JavaScript output. Therefore we need to be able to interact with the TypeScript compiler somehow. The TypeScript compiler has two main interfaces for interaction, through the command-line interface or using the compiler API. Optimally we would like to use the compiler API as this is the easiest way for us to perform type-checking and compilation. The catch is however that the only supported languages for the compiler API is JavaScript and TypeScript. Therefore a desired attribute for our choice of parser generator is that it offers a runtime library in either JavaScript or TypeScript, so that all of our implementation can be done in the same language, and not have to work with command-line interfaces programmatically.

ANTLR4

ANTLR, ANother Tool for Language Recognition, is a very powerful and versatile tool, used by many, such as Twitter for query parsing in their search engine [17].

ANTLR supports extending grammars, or more specifically importing them. Importing a grammar works much like a "smart include". It will include all rules that are not already defined in the grammar. Through this you can extend a grammar with new rules or replacing them. It does not however support extending rules, as in referencing the imported rule while overriding [17]. This isn't a major issue however as you could easily rewrite the rule with the additions.

The only supported runtime library in ANTLR is in Java. This does not mean that you won't be able to use it in any other language, as you could simply invoke the runtime library through command line, however it is worth keeping in mind.

Overall ANTLR seems like a good option for our project, but the lack of a runtime library in TypeScript is a hurdle we would rather get a round if we can.

Bison

Bison is a general-purpose parser generator. It is one of many successors to Yacc, and is upwards compatible with Yacc [5].

Bison does not support extending grammars. The tool works on a single grammar file and produces a C/C++ program. There is a possibility to include files, like with any other C/C++ program, in the grammar files prologue, however this will not allow us to include another grammar, as it only inserts the prologue into the generated parser. In order to extend a grammar we would have to change the produced parser to include some

extra rules. Although this could possibly be automated by a script, it seems too hacky of a solution to consider.

On top of this Bison does not have a runtime library in JavaScript/-TypeScript. There do exist some ports/clones of Bison for JavaScript, such as Jison and Jacob, however to my knowledge these also lack the functionality of extending grammars.

Tree-sitter

Tree-sitter is a fairly new parser generator tool, compared to the others in this list. It aims to be general, fast, robust and dependency-free [3]. The tool has been garnering a lot of traction the last couple of years, and is being used by Github, VS Code and Atom to name a few. It has mainly been used in language servers and syntax highlighting, however it should still work fine for our compiler since it does produce a parse tree.

Although it isn't a documented feature, Tree-sitter does allow for extending grammars. Extending a grammar works much like in ANTLR, where you get almost a superclass relation to the grammar. One difference from ANTLR though is that it does allow for referencing the grammar we are extending during rule overriding. This makes it easier and more robust to extend rules than in ANTLR.

Tree-sitter also has a runtime library for TypeScript, which makes it easier for us to use it in our implementation than the previous candidates.

Another cherry on top is that Tree-sitter is becoming one of the mainstream ways of syntax highlighting in modern editors and IDEs, which means that we could utilize the same grammar to get syntax highlighting for our language.

All this makes Tree-sitter stand out as the best candidate for our project.

Implementing Our Grammar in Tree-sitter

Tree-sitter uses the term rule instead of production, and I will therefore also refer to productions as rules here.

Extending a grammar in Tree-sitter works much like extending a class in an object-oriented language. A "sub grammar" inherits all the rules from the "super grammar", so an empty ruleset would effectively work the same as the super grammar. Just like most object-oriented languages have access to the super class, we also have access to the super grammar in Tree-sitter. All of this enables us to add, override, and extend rules in an existing grammar, all while staying loosely coupled with the super grammar. By extending the grammar, and not forking it, we are able to simply update our dependency on the TypeScript grammar, minimizing the possibility for conflicts.

As mentioned, Tree-sitter allows for referencing the super grammar during rule overriding, effectively making it possible to combine the old rule and the new. A good example of overriding and combining rules can be found in the grammar of PTS, see listing 9 on the next page, where we

```
    _declaration: ($, previous) =>
        choice(
            previous,
            $.template_declaration,
            $.package_declaration
        )
```

Listing 9: Snippet from the PTS grammar, where we override the `_declaration` rule from the TypeScript grammar, and adding two additional declarations.

override the `_declaration` rule from the TypeScript grammar, to include the possibility for package and template declarations.

5.3 Transforming Parse Tree to AST

5.3.1 The AST Nodes

Tree-sitter is a parser-generator written in Rust. Fortunately for us we can still use tree-sitter in Node, due to Node supporting native addons[1]. Native addons in Node are fairly new, and at the time of writing the tree-sitter Node bindings are still using the older unstable *nan* (Native Abstractions for Node) instead of the newer and more stable *Node-API*. For the most part it does work, however I did meet some difficulties with the produced parse tree, more specifically the spread operator was not behaving properly on the native produced objects. To get around this we will be walking through the parse tree and produce an AST. What this means in practice is that we are going to be ignoring some parsing specific properties. One of the changes we are going to make is that we are going to ignore if a node is named or unnamed, we will be keeping all nodes. This will help us later in code generation.

For each AST node we picked out the following properties from the parse tree:

- Node type
- Text
- Children

The node type is a string representing the rule which produced the node. An AST node with a node type value of `"class_declaration"` for instance is a class declaration node.

The text field of an AST node contains the textual representation/code for the node and its children. A class declaration node for instance would of course contain the class declaration (`"class A extend B {"`), but also the

[1] Node native addons are dynamically-linked shared objects written in C++ [15].

entire body of the class. This text field is really only useful for leaf nodes, as this would for instance contain the value of a number, string, etc.

Finally, the children field is, as the name would suggest, a list of all the children of the node. For a class declaration node this would contain a leaf node containing the keyword `class`, a type identifier for the class name, and the class body. Optionally it could also contain a class heritage node, which again contains either an extends clause, an implements clause or both.

We could have also opted to get the start position and end position of each node, so that we could use this to produce better error messages. This was however not a priority in this thesis.

5.3.2 Transforming

I chose to do the transforming immutable, and in order to do this we have to traverse the parse tree depth first and create nodes postfix. Tree-sitter provides pretty nice functionality for traversing the parse tree through cursors. With a tree cursor we are able to go to the parent, siblings, and children easily. Using this we visit each node and produce an AST node as described in the section above.

5.4 Closing Templates

A closed template is a template that comprises only class, interface or enum declarations. An open template on the other hand is a template that contains at least one instantiation statement.

The task of closing open packages and templates is what most of the implementation is focused around. It is the task of performing the instantiations declared in the body of a package/template.

For instantiations without renaming the task is fairly simple. We merely have to find the referenced template, and replace the instantiation statement with the body of said template. Renaming on the other hand requires a bit more work.

In order to perform renaming on an instantiation we will have to perform the following tasks.

1. Create a scope for the template body, and give all nodes of the body access to this scope

2. Find all identifiers, member expressions, class declarations, etc., and replace them with *reference nodes*[2]

3. Perform the rename.

[2]Reference nodes are AST nodes that contain a pointer to the class or attribute they are supposed to represent. This makes the task of renaming easier as we only have to worry about changing the name in one place. We will go into more detail about this in section 5.4.2 on the following page.

4. Transform the scoped AST with reference nodes back to the original AST.

5. Merge class declarations and apply `addto` statements.

If the template body is also open we would have to close it as well. We want to close the nested templates before closing the upper templates, as renaming at the top level should affect all members from the nested instantiations.

Finally, once all templates have been closed we will have to perform class merging and apply any additions to classes.

In order to get a better understanding of this we will go through each step of closing a template in more detail.

5.4.1 Scoping

This step of closing templates works on the body of a copy of the instantiated template. In order to be able to rename classes and class attributes we first need to create correct scopes in which the renaming can be applied to. We start of with a list of normal AST nodes and will transform these nodes into nodes that has a reference to the scope they are part of.

Scopes in the compiler mainly consists of three different classes. The `Scope`, `Variable`, and `Class` classes.

The `Scope` class is essentially a symbol table that optionally extends a parent scope. A scope without a reference to a parent scope is the root scope. The symbol table is implemented as a map from the original attribute or class name, to a reference to either `Variable` or `Class` instance. Looking up symbols in the symbol table will always start in the called scope, and work its way upwards until we reach the root scope. This allows us to correctly handle shadowed variables.

The `Variable` class is a representation for class attributes. The class contains the name of the attribute.

For creating scopes I chose the following node types for "making new scopes".

- `class_body`
- `statement_block`
- `enum_body`
- `if_statement`
- `else_statement`
- `for_statement`
- `for_in_statement`
- `while_statement`
- `do_statement`
- `try_statement`
- `with_statement`

5.4.2 Transforming Nodes to References

Transforming nodes into references mainly consists of two steps.

- Transform declarations
- Transform references

Transform Declarations

The task of transforming declarations requires several passes through the AST. This is because in order to register som declarations we need to have others registered. To understand this we can look at the difference between the first and second pass.

During the first pass through the AST we register and transform all class declarations. Now for the second pass we register class heritage. The second pass needs to update the `Class` instances from the first pass with what class they are extending. If we try to do this in a single pass we would have to register classes

Transform References

5.4.3 Performing the Rename

With all declarations and references transformed into reference nodes we can now perform the rename. For each class rename we can simply lookup the old class name in the root scope and change the reference to the new class name. For class attribute renames we do the same just for the enclosing classes scope.

5.4.4 Going back to the original AST

reference -> original with new names remove scope

5.4.5 Merging Class Declarations

Check if can be merged, non overlapping members. Override and merge addto

5.5 Verification of Templates

ts api

5.6 Code Generation

After performing these steps we can finally produce the output. Producing TypeScript output is a pretty simple task. By traversing the AST we can concatenate the text from each leaf node and insert some whitespace between them. This will produce quite ugly, unformatted code, but as long as the closed packages and templates are valid typescript programs this will also produce a valid typescript program. In order to make it a bit more clean we perform an extra step before writing the output to the

specified file, a formatting step. There are a lot of TypeScript formatters out there, but we will be using *Prettier* for our implementation. Running our produced source code through the prettier formatter produces a nicely formatted, readable output.

Since browsers only support JavaScript we will also be implementing this as a target for code generation. This is fortunately also a simple task, as we already dependent on the TypeScript compiler, and we are able to produce TypeScript source code, we can use this to produce JavaScript output.

5.7 Notes on Performance

Very slow compiler/PP because of the chosen implementation, with tree traverser for every step.

5.8 Testing

5.8.1 Lexer and Parser

Tree-sitter tests are simple `.txt` files split up into three sections, the name of the test, the code that should be parsed, and the expected parse tree in *S-expressions*[3].

5.8.2 Transpiler

Started with jest, and used some time to get it to work with typescript files, however had to switch because jest doesn't handle native libraries (tree-sitter) too well. It `requires` the same native library several times, making the wrapping around the native program to break.

[3] S-expressions are textual representations for tree-structured data. See [19] for additional information and examples

```
==========================
Closed template declaration
==========================

template T {
    class A {
        i = 0;
    }
}

---
(program
    (template_declaration
        name: (identifier)
        body: (package_template_body
                (class_declaration
                    name: (type_identifier)
                    body: (class_body
                        (public_field_definition
                            name: (property_identifier)
                            value: (number)))))))
```

Listing 10: Example of tree-sitter grammar test

Part III

Result

Chapter 6

Does PTS Fulfill The Requirements of PT?

This thesis is concerned about implementing Package Templates in TypeScript. However, in order to determine to what degree we have actually implemented PT or just created something that looks like it, we have to understand what the requirements of PT are, and if we are meeting those requirements. We will therefore in this chapter look at the requirements as described in [8](kanskje en hybrid med required types?). After getting an understanding of the requirements we are going to look at how our implementation holds up to them.

6.1 The Requirements of PT

In [8] the authors discuss requirements of a desired language mechanism for re-use and adaptation through collections of classes. They then present a proposal for Package Templates, which to a large extent fulfills all the desired requirements. In order for us to be satisfied to call our implementation of the language mechanism Package Templates, we will also have to meet these requirements.

The requirements presented in the paper were the following:

- Parallel extension
- Hierarchy preservation
- Renaming
- Multiple uses
- Type parameterization
- Class merging
- Collection-level type-checking

In order to get a better understanding of what these requirements entail we will have to dive a bit deeper into each requirement.

6.1.1 Parallel Extension

The parallel extension requirement is about making additions to classes in a package/template, and being able to make use of them in the same

```
template T {
    class A {
        ...
    }

    class B {
        ...
    }
}

package P {
    inst T;

    addto A {
        void someMethod() {
            ...
        }
    }

    addto B {
        void someOtherMethod() {
            A a = new A();
            a.someMethod();
        }
    }
}
```

Listing 11: Example of parallel extension in PTj. Here we make additions to both `A` and `B` in our instantiation in package `P`, and we are able to reference the additions done to `A` in our addition to `B`. This is done without the need to cast `A`, as if the additions were present at the time of declaration.

collection. What this means is that if we are making additions to a class, then we should be able to reference these additions in a declaration or in an addition to a separate class withing the same package/template, without needing to cast it. We can see an example of this in listing 11, where an addition to class `B` is referencing the added method of class `A`. The order of additions does not affect the parallel extension. We could just as easily switched the positions of the additions around in this example.

6.1.2 Hierarchy Preservation

PT should never break the inheritance hierarchy of its contents. If we have a template with classes `A` and `B`, and class `B` a subclass of class `A`, then this relation should not be affected by any additions or merges done to either of the classes. That is if we make additions to class `B` it should still be a subclass of class `A`, and any additions made to class `A` should be inherited

to class `B`. Even if we make additions to both class `A` and `B`, then `B` with additions should still be a subclass of class `A` with additions.

6.1.3 Renaming

The renaming requirement states that PT should enable us to rename the names of any class, and its attributes, so that they better fit their use case.

6.1.4 Multiple Uses

PT should be allow us to use packages/templates multiple times for different purposes in the same program, and any additions or renamings should not affect any of the other uses. Each use should be kept independent of each other. This is an important requirement of PT as when we create a package or a template it is often designed to be reused. An example of this is the graph template we created in listing 3 on page 8. Here we bundled the minimal needed classes in order to have a working implementation for graphs. We then used this graph implementation to model a road systems, however we might later also want to reuse the graph implementation for modelling the sewer systems of each city, and this should not be affected by any changes we made to the graph template for our road system.

6.1.5 Type Parameterization

Dette kan kanskje heller være en del av PT background. Skrev dette basert på JOT, men er det slik at PT nå bruker required types istedenfor? Da burde dette kanskje endres noe

The requirement of type parameterization of templates works similar to how type parameterization for classes works in Java. Type parameterization in Java enables the programmer to assume the existence of a class during declaration, and the class can later be given at the time of instantiation. Type parameters in Java can have a constraint that it must extend another class. Similarly, type parameterization in PT also enables the programmer to assume the existence of a class, however here the type parameter is accessible to the whole template. PT type parameters can also be constrained, however in PT type constraining uses structural typing instead of nominal typing. What this means is that instead of declaring what the type parameter must extend it instead declares what structure the type parameter must conform to.

In listing 12 on the next page we see an example of how type parameterization can be used to implement a list. This example might not look too different from what you would do with type parameterization in Java, however having the type parameter at the template level does have some advantages. One advantage of having the type parameter at template level is that you don't need to specify the actual parameter again after the instantiation. At instantiation of the `ListsOf` template we can give i.e. a `Person` class, containing some information about a persons name, date of

```
template ListsOf<E> {
    class List {
        AuxElem first, last;
        void insertAsLast(E e) { ... }
        E removeFirst() { ... }
    }
    class AuxElem {
        AuxElem next;
        E e; // Reference to the real element
    }
}
```

Listing 12: Example from [8] where type parameterization is used to create a list implementation.

birth, etc., as the actual parameter and then we would not have to keep respecifying the actual parameter for every reference. Another advantage of using type parameters at the template level is that the type parameter can be used by all classes in the template. If we wanted to implement this in Java we would either have to have type parameters for both classes, or the `AuxElem` class would need to be a inner class of the `List` class.

6.1.6 Class Merging

PT should allow for merge two or more classes. When merging classes the result should be a union of their attributes. If we merge two classes `A` and `B`, it should be possible to reach all `B` attributes from an `A`-variable, and vice versa.

6.1.7 Collection-Level Type-Checking

The final requirement is collection-level type-checking. This requirement is there to ensure that each separate package/template can be independently type-checked. By having the possibility to type-check each pacakge/template we can also verify that the produced program is also type-safe, as long as the instantiation is conflict-free.

6.2 Verifying Templates

Talk about unsoudness of TypeScript. Talk about unsoudness of Java [1] Talk about since the requirement is met with Java we assume it is adequately met with TypeScript as well.

Chapter 7

Difference between PTS and PTj

7.1 Nominal vs. Structural Typing

One of the most notable differences between PTS and PTj are the underlying languages type systems. PTS, as an extension of TypeScript, has structural typing, while PTj on the other hand, an extension of Java, has nominal typing.

Nominal and structural are two major categories of type systems. Nominal is defined as "being something in name only, and not in reality" in the Oxford dictionary. Nominal types are as the name suggest, types in name only, and not in the structure of the object. They are the norm in mainstream programming languages, such as Java, C, and C++. A type could be `A` or `Tree`, and checking whether an object conforms to a type restriction, is to check that the type restriction is referring to the same named type, or a subtype. Structural types on the other hand is not tied to the name of the type, but to the structure of the object. These are not as common in mainstream programming languages, but are very prominent in research literature. However, in more recent (mainstream) programming languages, such as Go, TypeScript and Julia (at least for implicit typing), structural typing is becoming more and more common. A type in a structurally typed programming language is often defined as a record, and could for example be `{ name: string }`.

7.1.1 Advantages of Nominal Types

Subtypes

In nominal type systems it is trivial to check if a type is a subtype of another, as this has to be explicitly stated, while in structural type systems this has to be structurally checked, by checking that all members of the super type, are also present in the subtype. Because of this each subtype relation only has to be checked once for each type, which makes it easier to make a more performant type checker for nominal type systems. However, it is also possible to achieve similar performance in structurally typed languages

```
// Given class A
class A {
    void f() { ... }
 }

// A subtype, B, in nominal typing
class B extends A { ... }

// A subtype, C, in structural typing
class C {
    void f() { ... }
    int g() { ... }
}
```

Listing 13: Example of subtype relations in nominal and structural typing, in a Java-like language. In the example of the nominal subtype we have to explicitly state the subtype relation, while in the structural subtype example the subtype relation is inferred from the common attributes.

through some clever representation techniques [18]. We can see an example of subtype relations in both nominal and structural type systems, in a Java-like language, in listing 13. It is important to note that even though `C` is a *subtype* of `A` in a structural language, it is not a *subclass* of `A`.

Runtime Type Checking

Often runtime-objects in nominally typed languages are tagged with the types (a pointer to the "type") of the object. This makes it cheap and easy to do runtime type checks, like in upcasting or doing a `instanceof` check in Java. It is also easier to check sub-type relations in nominal type systems, even though you might still have to do a structural comparison, you only have to perform this once per type.

7.1.2 Advantages of Structural Types

Tidier and More Elegant

Structural types carry with it all the information needed to understand its meaning. This is often seen as an advantage over nominal typing as the programmer only has to look at the type to understand it, while in nominal typing you would often have to look at the implementation or documentation to understand the type, as the type itself is part of a global collection of names.

Advanced Features

Type abstractions (parametric polymorphism, ADTs, user-defined type operators, functors, etc), these do not fit nice into nominal type systems.

More General Functions/Classes

Malayeri and Aldrich performed a study (see [9]) on the usefulness of structural subtyping. The study was mainly focused around two characteristics of nominally-typed programs that would indicate that they would benefit from a structurally typed program. The first characteristic was that a program is systematically making use of a subset of methods of a type, in which there is no nominal type corresponding to the subset. The second characteristic was that two different classes might have methods which are equal in name and perform the same operation, but are not contained in a common nominal supertype. 29 open-source Java projects were examined for these characteristics.

For the first characteristic the authors ran structural type inference over the projects and found that on average the inferred structural type consisted of 3.5 methods, while the nominal types consisted of 37.8 methods. While for the second characteristic the authors looked for types with more than one common methods and found that every 2.9 classes would have a common method without a common nominal supertype. We can see that from both of these characteristics that the projects could have benefited from a structural type system, as this would make the programs more generalized, and could therefore support easier re-use of code.

7.1.3 Disadvantage of Structural Type Systems

It is worth noting that the advantage of types conforming to each other without explicitly stating it in structural type systems can also be a disadvantage. Structurally written programs can be prone to *spurious subsumption*, that is consuming a structurally equal type where it should not be consumed. An example of this can be seen in listing 14 on the next page.

7.1.4 What Difference Does This Make For PT?

We are not going to go further into comparing nominal and structural type systems and "crown a winner", as there are a lot useful scenarios for both nominal and structural type systems. We will instead look more closely into how a structural type system fits into PT, and what differences this makes to the features, and constraints, of this language mechanism.

7.1.5 Which Better Fits PT?

With the addition of required types in PT the language mechanism now has to utilize structural typing, independent of the underlying language's type system. Required types in PT already uses structural typing, even if the underlying language might be using a nominal type system. This is because PT doesn't allow using externally declared classes, interfaces, etc., so in order to be able to have constraints on generic types structural typing was needed. I would therefore argue that a structurally typed language is a better fit for PT in order to make it feel more like a first-class citizen.

```
function double(o: {calculate: () => number}) {
    return o.value() * 2
}

const vector = {
    x: 2,
    y: 3,
    calculate: () => 4
}

const unintended = {
    value: () => {
        doSomeSideEffect();
        return 1;
    }
}
```

Listing 14: Example of spurious subsumption in TypeScript

```
// Given the following class definitions for A, B and C:
class A {
    void f() {
        ...
    }
}

class B extends A {
    ...
}

class C {
    void f() {
        ...
    }
}

// And a consumer with the following type:
void g(A a) { ... }

// Would result in the following
g(new A()); // Ok
g(new B()); // Ok
g(new C()); // Error, C not of type A
```

Listing 15: Example of a nominally typed program in a Java-like language

```
// Given the same class definitions and
// the same consumer as in the example above.
// Would result in the following
g(new A()); // Ok
g(new B()); // Ok
g(new C()); // Ok, because C is structurally equal to A
```

Listing 16: Example of a structurally typed program in a Java-like language

Having two different type systems for different contexts might prove to be confusing for the programmer.

It is important to note that even though structural typing might seem like a better fit for the current proposal of PT, there will certainly be proposals in the future where nominal type systems will prove to be the victor. The reader should also note that an implementation of PT in a language should not aim to change the underlying type system of the host language, as this will certainly only lead to more confusion than what we have explored in this chapter.

Chapter 8

Results

8.1 Future Works

8.1.1 Diamond Problem

essay

8.1.2 Improve the Compilers Error Messages

As I mentioned shortly in subsection 5.3.1 on page 32 tree-sitter does have support for giving position of a syntax node, and this could be utilized to produce better error messages.

Bibliography

[1] Nada Amin and Ross Tate. 'Java and scala's type systems are unsound: the existential crisis of null pointers'. In: *ACM SIGPLAN Notices* 51.10 (Dec. 2016), pp. 838–848. ISSN: 0362-1340. DOI: 10.1145/3022671.2984004. URL: https://dl.acm.org/doi/10.1145/3022671.2984004.

[2] *Arrow function expressions - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow%7B%5C_%7Dfunctions (visited on 19/10/2020).

[3] Max Brunsfeld. *Tree-sitter*. URL: https://tree-sitter.github.io/tree-sitter/ (visited on 19/02/2021).

[4] Cevek. *ttypescript*. URL: https://github.com/cevek/ttypescript.

[5] Free Software Foundation. *Bison 3.7.1*. URL: https://www.gnu.org/software/bison/manual/bison.html (visited on 24/02/2021).

[6] Eirik Isene. *PT#-Package Templates in C# Extending the Roslyn Full-Scale Production Compiler with New Language Features*. Tech. rep. 2018.

[7] Stein Krogdahl. 'Generic Packages and Expandable Classes'. In: (2001).

[8] Stein Krogdahl, Birger Møller-Pedersen and Fredrik Sørensen. 'Exploring the use of package templates for flexible re-use of collections of related classes'. In: *Journal of Object Technology* 8.7 (2009), pp. 59–85. ISSN: 16601769. DOI: 10.5381/jot.2009.8.7.a1.

[9] Donna Malayeri and Jonathan Aldrich. 'Is Structural Subtyping Useful? An Empirical Study'. In: *Programming Languages and Systems, 18th European Symposium on Programming, {ESOP} 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, {ETAPS} 2009, York, UK, March 22-29, 2009. Proceedings*. 2009, pp. 95–111. DOI: 10.1007/978-3-642-00590-9_8. URL: https://doi.org/10.1007/978-3-642-00590-9%7B%5C_%7D8.

[10] MDN Web Docs. *with - JavaScript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with (visited on 24/02/2021).

[11] Microsoft. *microsoft/TypeScript*. URL: https://github.com/microsoft/TypeScript/wiki/Writing-a-Language-Service-Plugin.

[12] Microsoft Corporation. *Typescript Language Specification Version 1.8*. Tech. rep. Oct. 2012. URL: https://web.archive.org/web/20200808173225if%7B%5C_%7D/https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md.

[13] Mozilla. *What is JavaScript? - Learn web development | MDN*. 2019. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First%7B%5C_%7Dsteps/What%7B%5C_%7Dis%7B%5C_%7DJavaScript%20https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First%7B%5C_%7Dsteps/What%7B%5C_%7Dis%7B%5C_%7DJavaScript%7B%5C#%7DA%7B%5C_%7Dhigh-level%7B%5C_%7Ddefinition%7B%5C%%7D0Ahttps://developer.mozilla.org/en-US/docs/Learn/Java.

[14] Mozilla Contributors and ESTree Contributors. *The ESTree Spec*. Tech. rep. URL: https://github.com/estree/estree.

[15] Node. *C++ Addons | Node.js v11.3.0 Documentation*. URL: https://nodejs.org/api/addons.html (visited on 04/03/2021).

[16] Stanislav Panferov. *Awesome TypeScript Loader*. URL: https://github.com/s-panferov/awesome-typescript-loader.

[17] Terence (Terence John) Parr. *The definitive ANTLR 4 reference*. Dallas, Texas, 2012.

[18] Benjamin C Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 9780262162098.

[19] *S-expression - Wikipedia*. URL: https://en.wikipedia.org/wiki/S-expression (visited on 25/01/2021).

[20] TypeScript. *Typed JavaScript at Any Scale*. 2020. URL: https://www.typescriptlang.org/ (visited on 09/03/2021).

[21] TypeStrong. *ts-loader*. URL: https://github.com/TypeStrong/ts-loader.

[22] TypeStrong. *ts-node*. URL: https://github.com/TypeStrong/ts-node.

[23] Wikipedia. *JavaScript - Wikipedia*. URL: https://en.wikipedia.org/wiki/JavaScript (visited on 09/03/2021).