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

Package Template Script

An Implementation of Package Templates in TypeScript

Petter Sæther Moen

Package Template Script

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

Printed: Reprosentralen, University of Oslo

Abstract

In this thesis we will explore how TypeScript can be extended with an additional language mechanism for re-use and adaptation, namely Package Templates. We will look at how Package Templates, which was initially designed for a nominally typed language, will work in a structurally typed language like TypeScript, and what differences this makes for its usage.

Package Templates, or as it was originally called, Generic Packages, is a language mechanism first proposed by Krogdahl in 2001. The language mechanism gives the programmer the opportunity to create collections of classes, interfaces and enums which can later be re-used and adapted. These collections can be instantiated inside new collections, where the mechanism allows for renaming classes and its attributes, as well as merging members of the instantiated collections. This enables the programmer to write general collections for concepts such as graphs and lists, and later adapt these to new domains with additional concepts forming collections for domains such as road systems between cities.

The result of the work done in this thesis is the Package Templates Script programming language, or just PTS for short, and an easily accessible compiler for the language. This contribution will hopefully make the language mechanism, Package Templates, more accessible for newcomers, and potentially spark further research in the field.

Acknowledgements

I would like to thank my supervisor, Associate Professor Eyvind Wærsted Axelsen, who has made me a more critical thinker through his thorough and pedagogic feedback and has helped me gain insights I would likely have lacked without his help.

I would also like to thank my co-supervisor, Professor Stein Krogdahl, who unfortunately fell ill and passed away. It was a true inspiration to work with someone with such vast knowledge and experience in the field of programming languages.

Finally, I would like to thank my parents who have supported and encouraged me throughout my education, and my friends who have motivated me and brightened my days in these rather challenging times.

Contents

Abstract i
Acknowledgments iii

I Introduction and Background 1

1 Introduction 3
1.1 Research Questions 3
1.2 Contributions 4
1.3 Chapter Overview 4
1.4 Project Source Code 5

2 Background 7
2.1 Package Templates 7
2.1.1 Basics of Package Templates 7
2.1.2 Concepts of PT 15
2.2 TypeScript 16
2.2.1 JavaScript 16
2.2.2 What is TypeScript? 19
2.3 Structural and Nominal Type Systems 19

II The project 23

3 The Language - PTS 25
3.1 Syntax 25
3.2 The PTS Grammar 26
3.3 Example Program 27

4 Planning the Project 31
4.1 TypeScript vs. JavaScript 31
4.1.1 Type-checking Templates 31
4.1.2 Renaming 32
4.1.3 Language Choice Conclusion 32
4.2 What Do We Need? 32
4.3 Approach 33
4.3.1 Implementing PT as an internal DSL 33
4.3.2 Preprocessor for the TypeScript Compiler 37

4.3.3 TypeScript Compiler Plugin/Transform 37
4.3.4 Babel plugin 38
4.3.5 TypeScript Compiler Fork 38
4.3.6 Making a Custom Compiler 39
4.4 Conclusion 39

5 Implementation **43**
5.1 Methodology 43
5.2 Compiler Architecture 44
5.3 Lexer and Parser 44
5.3.1 Parser Generator 44
5.4 Transforming Parse Tree to AST 47
5.4.1 The AST Nodes 47
5.4.2 Transforming 48
5.5 Closing Templates 48
5.5.1 Create a Correctly Scoped AST 49
5.5.2 Transforming Nodes to Reference Nodes 50
5.5.3 Perform the Rename 52
5.5.4 Go Back to the Original AST 54
5.5.5 Merge Classes 54
5.6 Type-checking of Templates 56
5.7 Code Generation 56
5.8 Testing 57
5.8.1 Lexer and Parser 57
5.8.2 Compiler 58
5.9 Completing the Implementation 59
5.9.1 `addto`-statements 59
5.9.2 Supporting All Attribute Declarations 60
5.9.3 Supporting All References 60
5.9.4 Supporting Multi File Programs 60

6 Using PTS **61**
6.1 Installing and Using PTS Globally 61
6.2 Creating a PTS Project 62
6.3 A "Real World" Example 63

III Results **67**

7 Evaluation and Discussion **69**
7.1 Does PTS Fulfill The Requirements of PT? 69
7.1.1 The Requirements of PT 69
7.1.2 PTS' Implementation of the Requirements 72
7.1.3 Conclusion 83
7.2 Nominal vs. Structural Typing in PT 83
7.2.1 Advantages of Nominal Type Systems 83
7.2.2 Advantages of Structural Type Systems 84
7.2.3 Disadvantage of Structural Type Systems 85

7.2.4 Which Better Fits PT? 85

8 Concluding Remarks **89**
8.1 Addressing Research Questions 89
8.2 In Retrospect 90
8.2.1 Approach 90
8.3 Future Work 91
8.3.1 Finishing the PTS Compiler 91
8.3.2 Improve the Compilers Error Messages 91
8.3.3 Making Syntax Highlighting for the PTS Language . 91

List of Listings

2.1 Defining a package `P` and a template `T` 8
2.2 Instantiating template `T` in package `P` 9
2.3 Example of renaming classes during instantiation. This could be used to make the classes fit the domain of the project better. 10
2.4 Example of instantiating the same template twice solved by renaming. 11
2.5 Adding new attributes to the instantiated class `A` in package `P` 12
2.6 Adding the `Loggable` interface to the `Graph` class from listing 2.3 on page 10, making it compatible with our logger implementation. 12
2.7 Instantiation with class merging through renaming 13
2.8 Example of a nominally typed program in a Java-like language 21
2.9 Example of a structurally typed program in a Java-like language 21
3.1 An example program with instantiation, renaming, and addition-classes in PTS vs. PT 29
4.1 Example of defining a template in a library implementation. 35
4.2 Example of renaming a template class 35
4.3 Example showcasing the problems of renaming classes in a library implementation. 36
4.4 Example of instantiating a template in a library implementation 36
5.1 Snippet from the PTS grammar, where we override the `_declaration` rule from the TypeScript grammar, and adding two additional declarations. 47
5.2 AST of a class declaration of class A before and after transforming the references. The values surrounded by angle brackets are references to `Scope/Class` instances. . . . 53
5.3 Example of Tree-sitter grammar test 57
5.4 Example of a test for the PTS compiler 58
7.1 Example of parallel extension in PT. 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. 70

7.2 Modified example from [13] where type parameterization is used to create a list implementation. 72
7.3 An example program that should fail during compilation, where we are trying to reference a non-existent attribute, `h`, in an addition to class `A`. 74
7.4 Example showcasing the preservation of super-/subclass relations 75
7.5 Example of renaming in PTS 77
7.6 Example showcasing the problem of having renaming in a structural language. In class `B` we have an attribute, `a`, that expects an object that contains an attribute `i`. The attribute is initialized with an `A` object. This is fine in template `T` as `A` contains an attribute `i`, however when class `A`'s attribute is renamed in the instantiation in package `P` then an object of `A` is no longer valid as a value, since it no longer contains an attribute `i`. This is an instance where we can't just rename the references to `i`, since this reference isn't explicitly related to `A`. 78
7.7 Example showing how a renaming of an interfaces' attributes could result in an invalid program. 79
7.8 A program showcasing multiple uses in PTS, and the resulting program in TypeScript at the bottom. 80
7.9 Example of a similar list implementation as in listing 7.2 on page 72, without the use of required types. Instead of giving a type for the required type we will have to merge the class `E` with the "actual parameter". 81
7.10 Example of class merging in PTS, where we merge two classes, `A`, with attributes, `i` and `j`, respectively 82
7.11 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. 84
7.12 Example of spurious subsumption in TypeScript 86
7.13 Example of how using renaming in PTS might break a program. After renaming the field `i` to `j` the class `Consumable` is no longer consumable by function `f` in class `Consumer`. . . . 87
7.14 Example of how using renaming in PT might break a program. After renaming the method `f` to `g` the class, `A`, no longer conform to the implementing interface `I`. 88

Part I

Introduction and Background

Chapter 1

Introduction

Package Templates is a language mechanism created at the University of Oslo, at the Department for Informatics. The language mechanism is a mechanism for re-use and adaptation, where you are able to define collections of classes, interfaces and enums. These collections can then be instantiated at a later time, in a different context, where we can tailor the collections' content to fit its use. Package Templates was first proposed by Krogdahl in 2001 [12], and was at the time called Generic Packages. Since then further proposals have been made to the language mechanism, and it is now known as Package Templates, or PT for short.

TypeScript is a superset of JavaScript, the programming language of the web. It extends JavaScript with the addition of static type definitions. These type definitions are used for type-checking the program at compilation, as well as serving as documentation for the program [25].

This thesis will explore how Package Templates can be implemented in TypeScript. Here we will discuss the different approaches that can be taken when working with a project such as this, and how such an implementation can be carried out.

The purpose of implementing PT in TypeScript is to look at how this language mechanism would it into a language like TypeScript. Most interesting is probably TypeScript's structural type system, and how this mechanism will work in this context, where other implementations in statically typed languages so far have only been conducted in nominally typed languages. It will also be interesting to see how PT can be used in the context of the web, with its vast variety of frameworks and libraries.

1.1 Research Questions

As we briefly touched upon in the introduction, an implementation of Package Templates in a language like TypeScript gives rise to some interesting research questions:

- **RQ1**: How does the language mechanism Package Templates fit into TypeScript?
- **RQ2**: Does structural typing change how the core of Package Templates works?
- **RQ3**: Will having PT in a structurally typed language have any notable advantages or disadvantages over having it in a nominally typed language?

1.2 Contributions

This thesis' main contribution is the PTS compiler. It is easily accessible through the Node Package Manager, henceforth referred to as *npm*. This makes it easy to try out the PTS language, but more importantly the PT language mechanism. Having easy access to a language with PT might make adoption of the language mechanism greater, and spark new research within the field.

By making the parser for the language separate from the compiler this also contributes to making creations of tools for the language more accessible. While we have in this project used the parser solely for producing a parse tree for our compiler, this parser could also be used to make other tooling, such as syntax highlighting or a language server.

The final contribution this thesis makes is conveying how to approach related projects. We show in this thesis how someone can approach extending a language by utilizing the grammar extending capabilities of the general-purpose parser generator, Tree-sitter, and how Tree-sitter can be used as the parser for a compiler.

1.3 Chapter Overview

Chapter 2 will give the reader an introduction to the Package Templates language mechanism and the programming language TypeScript. We will also look into TypeScript's underlying language JavaScript, and its ecosystem.

Chapter 3 will present the programming language PTS, which is a superset of TypeScript with the addition of Package Templates. Here we will look at the grammar of the language as well as an example program.

Chapter 4 is focused around the planning phase of the project. This includes a discussion about whether we will need to go for TypeScript as our host language, or if we could opt for the simpler underlying language, JavaScript. We will look at the requirements for our project and look at the different approaches we could use to implement PTS, as well as making a decision for which approach is the most beneficial for our project.

Chapter 5 is all about the implementation of our compiler for the PTS programming language. Here we will look at the methodology used during the implementation phase, as well as going into detail about how the compiler was implemented. As the compiler is not fully implemented we will also talk about what remains to be implemented, and how this could be implemented to complete the implementation.

Chapter 6 presents how our compiler for the PTS programming language can be installed and used. We will present the two main ways of installing the compiler, either as a project dependency or a global installation. With an understanding of how to get the compiler up and running we will look at a real world example of how PTS can be used.

Chapter 7 is the first chapter of the results part of the thesis. Here we will discuss and evaluate the PTS programming language, checking whether it is a "true" implementation of PT, and how PT is affected by a structurally typed language.

Chapter 8 concludes this thesis. Here we will revisit the research questions we introduced previously, and answer them with the knowledge we have gained in the span of this work. We will conclude the chapter, and the thesis, by looking at what could have been done better in retrospect, and some proposals for future works within this field.

1.4 Project Source Code

The source code for the implementation of the PTS language is split up into two GitHub repositories, one for the parser of the project, and one for the compiler. The parser's source code can be found at https://github.com/petter/tree-sitter-pts. Source code for the compiler can be found at https://github.com/petter/pts-lang.

Chapter 2

Background

2.1 Package Templates

Krogdahl proposed Generic Packages in 2001, which is a language mechanism aimed at "large scale code re-use in object-oriented languages" [12]. 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 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 Basics of Package Templates

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 [13], with the extensions of required types as proposed in [3].

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 similarly except for using the keyword `template` instead of `package`. Listing 2.1 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.

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

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

Listing 2.1: Defining a package `P` and a template `T`

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 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 2.2 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`. 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.

Renaming

During instantiation it is possible to rename classes (as well as interfaces and enums) and *class attributes*. Here and henceforth we will be using the term class attributes to describe the union of both the fields and the methods of a class. 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 thin arrow syntax, `i -> j`, where the attribute `i` is renamed to `j`. For method renaming, the signature of the method has to be given, 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 2.3 on page 10 shows an often used

```
// 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 2.2: Instantiating template `T` in package `P`

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 2.4 on page 11 shows an example of this where we instantiate the same template, `T`, twice without any issues.

Additions to a class

When instantiating a template you can also add attributes to the classes of the template, as well as extending the class' implemented interfaces. These additions will only apply to the currently instantiated copy. 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 2.5 on page 12 shows an example of adding attributes to an

```
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 2.3: 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 2.4: Example of instantiating the same template twice solved by renaming.

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 2.3 on the preceding page, 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 the following section.

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

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

Listing 2.5: Adding new attributes to the instantiated class `A` in package `P`

```
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 2.6: Adding the `Loggable` interface to the `Graph` class from listing 2.3 on page 10, making it compatible with our logger implementation.

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

template T2 {
    class B {
        ...
    }
}

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

Listing 2.7: Instantiation with class merging through renaming

Merging classes

If two or more classes in the same or in different instantiations in one package 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* (a language feature for injecting code into a class, first introduced in the programming language Jigsaw [6]). The merging of the classes will not lead to a classic superclass-subclass relation, as the merged class is simply a concatenation of textual copies of the merging classes. 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 2.7 we see an example of renaming class `A` from template `T1` to `C` and class `B` from template `T2` to `C`. Renaming these two classes to the same name will force these classes to be merged in package `P`. The result of this merge is a new class `C` with the attributes of 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 `C`, which is a union of both. Any pointers typed with the old `A` or `B` will now be typed with the new merged class `C`.

Required Types

Required types in PT gives the programmer extra flexibility when declaring templates. They are generic types declared at the template level, which can be substituted at instantiation. If a template instantiates another template with a required type, but does not give an actual parameter for the required type, then the required type is propagated to the template it is being instantiated into. When a template with required types is instantiated in

a package, then all the required types needs actual parameters.

Required types can then be used throughout the template, similar to how you would use generics in a Java class. The most basic required type can be seen below.

```
template T { required type R { } }
```

Here `R` is a required type for which any class or interface can be substituted at instantiation. Required types can be constrained using both nominal types, such as classes and interfaces, and structural types, constraining the type to have certain attributes. Below we can see examples of declaring required types with different types of constraints, where the first has a simple nominal type constraint, the second having a structural constraint, and the third having both a nominal and structural constraint.

```
template T {
    required type R1 extends Runnable { }
    required type R2 { void f(); }
    required type R3 extends Runnable { void f(); }
}
```

We could then instantiate the template, `T`, giving classes or interfaces as actual parameters for the required types, as seen below.

```
package P {
    class A implements Runnable {
        void run() { ... }
    }

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

    interface C extends Runnable {
        void f();
    }

    inst T with
        R1 <= A,
        R2 <= B,
        R3 <= C;
}
```

Required types as presented above can not be used as classes or interfaces, that is you cannot create a new object of the type or implement the type as an interface for a class, as they can be substituted with both. They can only be used as type references, like in the simple `Tree` implementation example below.

```
template Tree {
    required type E { }
```

```
        class Tree {
            Node root;
            ...
        }

        class Node {
            E e;
            List<Node> children;
            ...
        }
    }
```

It is also possible to declare required classes and interfaces similarly to required types, which can be used as classes and interfaces respectively, however this will not be discussed in this thesis. If the reader wants to get a better understanding for required types, required classes and required interfaces I recommend reading [3].

2.1.2 Concepts of PT

With a firm understandings of the basics of PT we will now have to dig a bit deeper into some terminology and restrictions of the language mechanism.

Open and Closed Templates

A *closed template* is a template that does not contain any instantiation statements nor any additions to classes, it comprises only classes, interfaces and enums. The body of a closed template in Java is simply a Java program. Closed templates are self-contained units that can be separately type-checked [5]. An *open template* on the other hand is a template which do contain one or more instantiations or `addto`-statements in its body.

Open templates will be closed at compile-time. The task of closing a template is that of performing the contained instantiations and additions to classes. Open templates can instantiate open templates, as long as these instantiations are not cyclic. What this means is that a template `A` can instantiate a template `B` if template `B` does not contain any (transitive) instantiations of template `A`. A template `B` contains an instantiation of template `A` if it has an instantiation of template `A` in its body, or contains a nested instantiation of template `A`.

Packages can also be open and closed and work in the same manner as with templates, except that they can not be instantiated.

Avoiding Indirect Multiple Inheritance

While PT enables the programmer to merge classes together and giving us some form of static multiple inheritance, it is not intended to actually enable multiple inheritance. However, with class merging, it is not

uncommon that a class might end up with two or more different superclasses. To avoid this PT has some restrictions to stop this from happening.

The first restriction is that if an external class is used as a superclass, then it can only be merged with other classes with the same superclass. This restriction is necessary since we can't rename nor merge external classes.

The second restriction is that if two or more classes are merged in an instantiation, then their superclasses must also be merged in the same instantiation. This is to avoid the situation where merging two classes results in two or more different superclasses [13].

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 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 what we know it as today, JavaScript. This has been characterized as a marketing ploy by Netscape 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 of 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 [27].

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 [18].

JavaScript is a programming language conforming to the ECMAScript standard. ECMAScript is a JavaScript standard, created by Ecma International, made to standardize the JavaScript language and 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[1].

JavaScript is a multi-paradigm language, mainly consisting of object-orientation and functional programming, with a dynamic type system. It is object-oriented in the way that most data structures are represented through objects, and functional in the way that it has first-class functions, where functions can be free from a class and are treated as values that can be assigned to variables and sent around as parameters.

Where most object-oriented programming languages are class-based, like Java, JavaScript is *prototype-based*. What this means is that the objects are not class instances, but are rather "instances" of a prototype. What you would normally think of as a class instance in Java, is an object with a reference to a prototype object. These instances are created through constructor functions, which create an object and sets the prototype for the object.

An object in JavaScript is a "bag" of properties containing values, which are specified in the prototype constructor, and a reference to the prototype object it is an instance of. The prototype object is not special in any way, it is just another object that has contains values that can be commonly used by all objects with the same prototype, and can themselves have prototype objects. This is how inheritance works in JavaScript, chains of prototype objects, where the `Object` prototype is at the end of the chain, similar to how the `Object` class is at the top of the inheritance hierarchy in Java. The `Object` prototype has `null` as its prototype, and `null` does not have any prototype. When trying to access a member of an object the object itself is first checked, then its prototype, and the prototype's prototype and so on, following the chain of prototypes, until a match is found [10], or until there are no more prototypes to follow. Since prototypes are just objects they can as with any other object be changed at runtime or replaced by other prototypes.

In ECMAScript 2015 there was introduced a class-syntax, however, this is just syntactic sugar for creating the prototype object, and the associated constructor. Extending a class with this syntax is as you would expect just defining the prototype for the prototype object.

[1] 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[2]. This is because the developer will not be able to control the environment on which the code will be executed since you can not be sure which ECMAScript version the client browser is 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 browsers 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 that you ship with your code. These polyfills help the developer regain some control over the runtime environment on which the code will be run, and ensure that the code will run on almost any browser as expected.

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 originally intended, or you can use a JavaScript runtime that runs on the backend, outside 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

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

WebSocket events. This makes Node a good choice for writing networking applications for instance.

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. TypeScript's type system is structural, which means that the type of an object is not bound to a name, such as with nominal typing, but rather the structure of the object, such as having an attribute `i`, which may be restricted to a number. The type system also offers some more advanced type features such as union types, where you can combine types into a new type. The new union type represents values that can be any one of the combined types. There are also similarly intersection types. These types combine other types into a new type, which is the intersection of the combined types.

All valid JavaScript programs are also valid TypeScript programs. Types in TypeScript can be optional, as the type inference is powerful enough to infer most types without writing extra code. The type-checking can be tailored to be stricter or leaner, where you can for instance disable features such as usage of `any`-types, which are a way for the programmer to bypass the type-check for certain values. TypeScript has full interoperability with JavaScript, so you can adopt the language without needing to rewrite your entire code base. If you are working with a JavaScript library, but you want the safety of types, there can often be found type declaration files written by the community in the DefinitelyTyped project [25].

2.3 Structural and Nominal Type Systems

Throughout this thesis we will have a major focus on the underlying type systems of traditional PT in Java, and our implementation of PT in TypeScript. Java has what we call a *nominal* type system, while TypeScript has a *structural* type system.

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 are 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 and TypeScript, 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 }`.

In listing 2.8 on the next page we can see an example of a nominally typed program in a Java-like language. Here `B` is a subtype of `A`, while `C` is not. This is due to nominally typed programs having the requirement of explicitly naming its subtype relations, through e.g. a subclass-relation. Because of this we can see that at the bottom of the listing the first two statements pass, since both `A` and `B` are of type `A`, while the last statement fails (typically at compile time), as `C` is not of type `A`.

In listing 2.9 on the facing page we see a structurally typed program. This program also has the exact same declarations as in listing 2.8 on the next page, that is classes `A`, `B`, and `C` and the function `g`. In this program both type `B` and type `C` are a subtype of type `A`, since they both contain all members of type `A`. Not necessarily the same implementation as in class `A`, but the same types as in type `A`. This is one of the major differences between nominal and structural typing, types can conform to other types without having to explicitly state that they should. Type `C` is an example of this, while it does not have a subclass relation to class `A`, nor implement any common nominal interface, it still conforms to the type of `A`. The result of this is that all three usages of function `g` are valid in a structural type system, while consuming `C` was illegal in the nominal example.

```
// 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 2.8: Example of a nominally typed program in a Java-like language

```
// Given the same class definitions and
// the same consumer as in the previous listing.
// 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 2.9: Example of a structurally typed program in a Java-like language

Part II

The project

Chapter 3

The Language - PTS

In this chapter we will introduce the programming language Package Template Script, henceforth just referred to as PTS. Here we will make decisions about the syntax of the language, whether we can keep most of the syntax of the original PT proposal, or if we will have to make some adjustments to avoid concept confusion and an ambiguous grammar.

3.1 Syntax

For the implementation of PT we need a way to express the following language constructs:

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

`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/JS 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 PT uses `=>`(fat-arrow) for renaming classes, and `->`(thin-arrow) for renaming class attributes. PT, for historical purposes, used two different operators for renaming classes and methods, however in more recent PT implementations, such as [11], 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 [15]. 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 PT, being enclosed in a set of 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 PT 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 [17]. This TypeScript specification was deprecated as it proved a too 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 [19].

In figure 3.1 on the facing page we can see the BNF grammar for our language. This is not the full grammar for PTS, as I have only included

⟨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.

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 non-terminals ⟨id⟩, ⟨class declaration⟩, ⟨interface declaration⟩, and ⟨class body⟩ are also from the original grammar.

3.3 Example Program

Listing 3.1 on page 29 shows an example of a program in PTS. This program showcases the basics of defining packages and templates, and how instantiation, renaming and additions can be applied in the language. We also have a similar program at the bottom, showing how this is done in PT. While both the basic instantiation and additions stay pretty much

the same, renaming does have some interesting differences. We can see that in the PT example we have to specify the signature of methods we are renaming, while in the PTS example it is enough to just specify the names of the methods.

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

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

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

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

Listing 3.1: An example program with instantiation, renaming, and addition-classes in PTS vs. PT

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 option. We might also want to see if it is needed to create a language at all, or if we are able to create a TypeScript library which can achieve the functionality of PT instead. There are a lot of approaches we can take for implementing our language, so we will have to map out the requirements for our desired approach. We will conclude the chapter by looking at the different approaches we can take, and see which approach is right for the project.

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 to go for TypeScript as the host language, or would it be 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 Type-checking 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 safe. Because of this trait TypeScript is the better candidate for our language.

Now it should be noted that due to TypeScript's 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 7.1.2 on page 82 discuss more in-depth to what extent this requirement is met.

4.1.2 Renaming

Renaming is a hard task. In order to perform a (safe) renaming we will need to find the declaration and all references to this declaration and rename these. Doing this at compile time would mean that we will have to implement a type system of sorts, since this will help us identify references. This is also one of the reasons for why TypeScript is a better candidate than JavaScript, as TypeScript is statically typed, meaning the type of a variable is known at compile-time, while JavaScript is dynamically typed, where the type of a variable is first known at run-time. While TypeScript generally allows us to determine the types of variables at compile time, this is not always the case, since it is possible for the programmer to explicitly type a variable with `any`, a catch-all type which effectively bypasses type-checking. This means that we can still run into the same issues as we would in a JavaScript program, and not be able to perform a safe renaming, however in cases such as these where the programmer has explicitly chosen to bypass the type-check, it might then also be acceptable to not offer renaming of `any`-typed variables.

4.1.3 Language Choice Conclusion

There has previously been done research into dynamic variants of PT, where the PT transformations have been done mostly at run-time, so it is certainly a possibility for us to also write a dynamic variant of PT for JavaScript. However, as we discussed above, Package Templates has a lot of properties that are designed around strong typing, and we would therefore benefit from hosting PT in a strongly typed language like TypeScript. This will likely also prove to be more interesting research, as we could rather focus on TypeScript's structural type system, than focusing on creating another dynamic variant of PT. Because of these reasons we will in this thesis look at how Package Templates can be implemented into TypeScript.

4.2 What Do We Need?

There are a lot of approaches one can take when extending 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

Having a loosely coupled implementation might mean different things in different approaches. Generally we want our PT specific part of the implementation to stay loosely coupled with the TypeScript specific parts. In a TypeScript compiler fork this would for instance be pretty much unachievable, since the PT implementation would likely have to change some of the TypeScript implementation. If we are able to simply write a preprocessor to the TypeScript compiler however, this would be fulfill the requirement, as we could transform the PT specific parts of the language before letting the TypeScript compiler deal with the rest of the program. Having the implementation as loosely coupled as possible would make our implementation cheaper to maintain, as updates to the TypeScript compiler would likely not break our implementation, as it should not be affected by the TypeScript specifics.

4.3 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:

- Implementing as an internal DSL
- Making a preprocessor for the TypeScript compiler
- Making a compiler plugin/transform
- Making a fork of the TypeScript compiler
- Making a custom compiler

4.3.1 Implementing PT as an internal DSL

One of the first approaches we need to check out is if we are able to achieve the functionality of PT, without having to create a compiler. Instead of creating a compiler, which would likely be a complicated and time-consuming task, we could potentially get away with implementing the features of PT within TypeScript itself through making a small internal DSL[1]. In [4], Axelsen and Krogdahl show how they were able to implement PT in Groovy by utilizing the languages' meta-programming capabilities.

[1] While most programming languages are made to be general purpose, DSLs, or Domain-Specific Languages, are languages created to solve very specific problems within certain domains. An internal DSL (sometimes referred to as embedded DSL) is a language based on an existing programming language, but "tailoring it [...] to the domain of interest" [9].

Stordahl also showed the possibility of implementing PT in Boo, through its meta-programming capabilities in [24]. We will in this section therefore see if we are able to achieve something similar in TypeScript.

Both Groovy and Boo have strong meta-programming capabilities, where they can perform transformations to the syntax tree during compilation. JavaScript and TypeScript do not have the same capabilities for meta-programming during compilation, as JavaScript is intended as an interpreted language, and TypeScript is only supposed to offer type declarations without changing any of the underlying functionality, so any compile-time transformations are not an option for our DSL. Besides the compilation transformations used for some custom syntax, Groovy gives the programmer access to the *meta-object protocol*, where each object has a reference to its *meta-class*, where members of a class can be added or changed at runtime. This was utilized by Axelsen and Krogdahl during the implementation of GroovyPT, and we could seemingly achieve something similar in TypeScript. Similar to Groovy's meta-classes, objects in JavaScript have references to a prototype object. These prototypes can have members added or removed, or the entire prototype replaced, at runtime. Utilizing this we could potentially be able to implement PT's class merging, renaming and additions. We will dive deeper into this in the following sections.

To implement PT we will need to handle the following:

- Defining templates
- Renaming classes and class attributes
- Instantiating templates
- Merging classes

Defining Templates

For defining templates we would like a construct that can wrap our template classes in a scope. We will also need to be able to reference the template. JavaScript has three options for this, an array, an object or a class. It should however also be possible to inherit from classes within the same template, which rules out both arrays and objects, as there is no way of referencing other members during definition of the array/object (without first defining them outside the construct). Templates could therefore be defined as classes, where each member of the template is an attribute of the template class. In listing 4.1 on the next page we see an example of how this could be done.

Renaming Classes and Class Attributes

Renaming of classes is possible to an extent. Since we made the classes static attributes of the template class we could easily just create a new static

```
class T1 {
  static A = class {
    i = 1;
  };

  static B = class extends T1.A {
    b = 2;
  };
}
```

Listing 4.1: Example of defining a template in a library implementation.

```
class T1 {
  static A = class {
    i = 0;
  };
}


const classRef = T1.A;
delete T1.A;
T1.B = classRef;
```

Listing 4.2: Example of renaming a template class

field on the template class and use the `delete`-op [2] to remove the old field. We can see an example of this in listing 4.2.

Even though we were able to give the class a "new name", this would still not actually rename the class. Any reference to the old names would be left unchanged, and thus we are not able to achieve renaming in TypeScript. Listing 4.3 on the next page shows how this can be a problem, where the function `f` of class `X` would fail at run-time due to it not being able to find class `A`.

Attribute renaming can be done similarly, where we could alter the prototype of the class to rename attributes, however this would also lead to the same problems as with class renaming, where references would not be changed.

Instantiating Templates

As with renaming, we are also able to instantiate templates to an extent. We are able to iterate over the attributes of the template class, and populate a package/template with references to the template. An example of this can be seen in listing 4.4 on the following page.

[2] An operator in JavaScript for removing a property of an object. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete.

```
    class T1 {
        static A = class {
            i = 0;
        }
        static X = class {
            f() {
                return new A();
            }
        }
    }

   // Renaming
    const classRef = T1.A;
    T1.B = classRef;
    delete T1.A;

    // Trying to use the template after renaming
    const x = new T1.X();
    x.f(); // ReferenceError: A is not defined
```

Listing 4.3: Example showcasing the problems of renaming classes in a library implementation.

```
class T2 {
  static A = class {};
}

const P = class {};
for (let attr of Object.keys(T2)) {
  P[attr] = T2[attr];
}
```

Listing 4.4: Example of instantiating a template in a library implementation

The instantiation will only contain references to the instantiated templates classes, while PT instantiations make textual copies of the templates content. Only having references to the original template could mean that if a template that has been instantiated is later renamed, then the instantiated template might lose some of its references. We could possibly circumvent this by getting the textual representation of the class, through the class' `toString`, and then use `eval` to evaluate the class declaration.

Merging Classes

For merging of types we would use the built-in declaration merging [26]. Implementation merging is also possible because JavaScript has open classes. For implementation merging you would create an empty class which has the type of the merged declarations, and then assign the fields and methods from the merging classes to this class. There are several libraries that supports class merging, such as mixin-js[3].

4.3.2 Preprocessor for the TypeScript Compiler

Could we implement the PT specific features in a preprocessor? In order to understand this we need to understand what a preprocessor is. There are a lot of different definitions for preprocessors, but they are generally something that makes a source file ready for the compiler, through some simple transformations. I will here define a preprocessor as a "dumb" compiler. Where a compiler generally works on the source file as a tree, requiring knowledge of the underlying programming language, performing advanced tasks such as semantic analysis, a preprocessor works on the source file as a piece of text, without knowledge of the language, performing simple textual transformations such as removing comments, expanding macros (such as `#include` in C), etc.

So the question becomes, can we transform a PTS program to TypeScript by just doing textual transformations, and not having to rely on performing more advanced tasks such as semantic analysis. We would most likely be able to implement parts of PT with a preprocessor such as simple instantiation without renaming. However, as we mentioned in section 4.1 on page 31 we will need to do some type-checking in order to find the correct references when renaming, we can't just rename everything that is textually equal. This means that we will need an understanding of the underlying programming language, something more advanced than a preprocessor to implement the features of PT.

4.3.3 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

[3] https://www.npmjs.com/package/mixin-js

only" [16]. However, there are alternatives that do enable compile time plugins/transformers;

- ts-loader[4], for the webpack ecosystem
- Awesome Typescript Loader[5], for the webpack ecosystem.
- ts-node[6], REPL/runtime

Unfortunately all of the above do not support adding custom syntax, as they only work on 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.

4.3.4 Babel plugin

Babel isn't strictly for TypeScript, but for JavaScript, however there does exist a plugin for TypeScript in babel, and we could write a plugin that depend on this TypeScript 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.

4.3.5 TypeScript Compiler Fork

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 could prove a hard task 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 might presumably 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.

[4] https://github.com/TypeStrong/ts-loader
[5] https://github.com/s-panferov/awesome-typescript-loader
[6] https://github.com/TypeStrong/ts-node

4.3.6 Making a Custom Compiler

Making a custom compiler for PTS might seem like a hard task, but let us dig deeper into what this entails. Firstly we need to consider what the target should be. Normally a compiler would output some sort of byte code, like Java byte code in the Java compiler. Many compilers also produce native code. Producing native code is not an option for our implementation as we still want to stay in the same ecosystem, namely the browser. We could possibly produce WebAssembly byte code, however there are a lot of constructs in TypeScript/JavaScript that do not translate to WebAssembly, such as working with the DOM. Since neither of these are valid options we could either produce TypeScript or JavaScript. Producing TypeScript is possibly the easiest way to go, as most of PTS is already TypeScript. And producing TypeScript also means that we could run the resulting program through the TypeScript compiler to produce JavaScript.

Having TypeScript as the target for our compiler also means that we can ignore most parts of the language and mainly focus on the PT specifics. The rest of the language can be outputted pretty much as is, since our language will be a superset of TypeScript.

4.4 Conclusion

While it would be great to be able to implement Package Templates as an internal DSL in TypeScript, it would seem that this is not a suitable approach. Even though we were able to modify the prototype of the classes in the templates, and effectively achieve some form of renaming, we were not able to rename the references. This means that we won't be able to use the renaming to its fullest potential, and are thus not able to implement it as an internal DSL. On top of this, while we were able to reproduce certain PT functionality such as simple instantiations and class merging, the fact that we are not able to change the syntax of the language, and having to define templates as classes of classes leads to a quite ugly DSL, which could also potentially be hard for the programmer to grasp.

Making a preprocessor to the TypeScript compiler in order to implement the features of PT would presumably make the implementation time short. However, as we learned, in order to safely rename classes and attributes we need something more powerful than a simple preprocessor. If we were to look at the core of PT, without the renaming mechanism this would likely be the easiest approach, however, since we are aiming to implement the renaming mechanism this makes this approach not viable for the project.

Making a TypeScript compiler plugin would seemingly also be a good approach, in the future. However, as we discussed previously, the official TypeScript compiler (and its alternatives) does not have proper support for plugins that can alter the syntax of the language. Due to this we are not able to implement the features of PT, since these would require us to add extra syntax. For the time being this makes this approach not viable, however, if

in the future this would be supported it might prove a good approach for doing tasks such as these.

Implementing PT in a fork of the TypeScript compiler would likely lead to the most robust implementation, however the sheer size of the TypeScript compiler makes this approach undesirable. I fear that this approach would be too time costly for this project, and might lead to an incomplete implementation as a result of this. A similar project has been performed by Isene in [11], where they implemented PT in C# by extending the Roslyn compiler. Here Isene suggested that a project of this size was better fit for a group of two. To avoid re-discovering this we will therefore opt to go for another approach. Furthermore, an implementation of PT in the TypeScript compiler would not achieve our desired trait of having a loosely coupled implementation. This could result in a tedious process of dealing with merge conflicts when updates to the TypeScript compiler comes out.

Creating a plugin for babel might be a good approach, however since we have to implement our grammar as part of a fork of the babel parser, this makes the approach less desirable. As with the approach of implementing PT in a fork of the TypeScript compiler this would also lead to a tightly coupled implementation, at least for the parser part of our compiler. If we were able to write a plugin for the parsing step of Babel this might prove a viable option, however as of now there are no plans of supporting this.

Our last approach is creating a custom compiler. As we discussed, if our compiler can simply target TypeScript for code generation, then the custom compiler approach might be a valid option. Having TypeScript as the target means that most of the language's constructs can be ignored, since most of the PTS language is simply TypeScript code, we can output most of the code as is. The only transformations our compiler will need to do will be the PT specific transformations, such as replacing instantiation statements with the bodies of the instantiated templates. As it is a custom compiler we will be able to fulfill our requirements, since we do have access to the parser, so we can add new syntax, and we are able to perform semantic analysis. For the desired trait of being loosely coupled we could likely fulfill this as well with this approach. Since we will only be doing PT specific transformations in the compiler, this means that most of our compiler will stay loosely coupled with TypeScript. The one part of our compiler that might not be able to stay as loosely coupled as we would like is the lexer/parser. We will need parse the TypeScript parts of the PTS program in order to perform most of the PT specific transformations, such as renaming classes. However, some parser generators do allow for importing or extending grammars, so we might be able to at least partially stay loosely coupled. We will look more into this in chapter 5 on page 43.

It would seem that the most beneficial approach for our project is to write a custom compiler, since the other options were either not able to fulfill the requirements of our project, or ,in the case of implementing PT in a fork of the TypeScript compiler, too great a task for one person alone. With having made a choice for the approach, we can look at the implementation of our

compiler.

Chapter 5

Implementation

In this chapter we are going to look at the implementation of our compiler for PTS programming language, as described in chapter 3 on page 25. Before looking at the implementation we will first be discussing the methodology used during development.

5.1 Methodology

When tackling a project of this magnitude it is important to have a proper methodology for development. During the development phase of this project I have had a strong focus on using agile techniques, where I have filled the role as both product owner and developer. This agile software development has aided me in discovering new requirements as the project moves forward, and re-adjusting to these new requirements. I have actively used a Kanban board throughout development to help keep track of tasks and goals.

The compiler was made in an iterative manner. For each iteration I would start off by implementing a new feature, and then put on the product owner hat and test out the compiler. While working as product owner I try to understand how I would like to use the language and what requirements I have for the language. This often leads to re-adjusting the requirements.

I started off by creating a rough MVP (Minimum Viable Product), only implementing the most basic functionality, which comprised declaration of packages/templates and simple instantiation. This MVP made me understand the project and requirements better, and also gave the project some new requirements. After the initial iteration I decided to adopt a test-driven development approach. I made tests for the features I had already implemented and then continued to make tests for the next functionality goal. This was done in order to gain more confidence in the compiler, as well as helping me spot any erroneous code earlier rather than later, which makes fixing them less costly. All of this resulted in a better development cycle, making refactoring and implementation of new features a breeze.

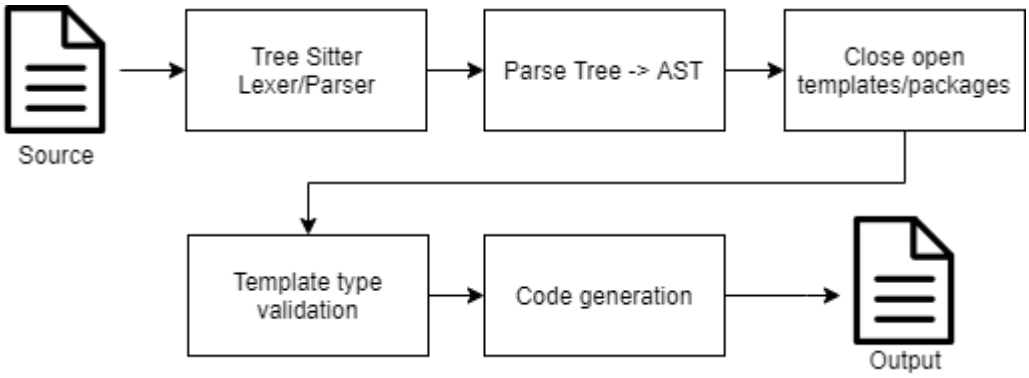

Figure 5.1: Overview of the compiler

When adding new features or refactoring some tests will undoubtedly fail, and before moving on I made sure that all the tests were passing again.

5.2 Compiler Architecture

Our compiler consists of the following parts:

- Lexing and parsing
- Parse tree transformation
- Type checking packages/templates
- Code generation

An overview of our architecture can be seen in figure 5.1. The first part of the compiler, namely the lexing and parsing will take a source file and transform it into a parse tree. Our compiler will then take this parse tree and transform it into a simpler abstract syntax tree (AST). This AST will then be used to perform the PT transformations. We will use this AST to close any open packages and templates, and finally use the transformed tree for code generation. Before code generation we will perform a type-check step, where we validate the type-safety of each package/template individually, to ensure an overall type-safe program. Given a valid type-safe program we can then move on to code generation. The target language for our code generation will mainly be TypeScript, however we will also offer to target JavaScript, as we can easily produce JavaScript code through using the TypeScript compiler.

5.3 Lexer and Parser

5.3.1 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 32, 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 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 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 are 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 interface 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 [21].

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 [21]. 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 [8].

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[1] and Jacob[2], however to my knowledge these also lack the functionality of extending grammars.

Tree-sitter

Tree-sitter[3] is a fairly new parser generator tool, compared to the others in this list. It aims to be general, fast, robust and dependency-free [7]. 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, and will move on with implementing our grammar in Tree-sitter.

[1] https://zaa.ch/jison/
[2] https://canna71.github.io/Jacob/
[3] https://tree-sitter.github.io/tree-sitter/

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

Listing 5.1: Snippet from the PTS grammar, where we override the `_declaration` rule from the TypeScript grammar, and adding two additional declarations.

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

5.4 Transforming Parse Tree to AST

5.4.1 The AST Nodes

Tree-sitter is a parser-generator written in Rust and C. Fortunately for us there does exist Node bindings for Tree-sitter. These Node bindings uses Node native addons[4] to interop with the Tree-sitter core. 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

[4]Node native addons are dynamically-linked shared objects written in C++ [20].

the parse tree and produce a new 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 extends B {"`), but also the 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.

5.4.2 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.

5.5 Closing Templates

The task of closing open packages and templates is what most of the implementation is focused around. It is the task of performing the declared instantiations and altering the declared classes through `addto`-statements.

This step is crucial as it will make each package/template a valid TypeScript program and make the program ready for code generation.

For the most basic instantiations that doesn't lead to any renaming, performing additions to classes, or merging of classes, the task is fairly simple. We merely have to find the referenced template, and replace the instantiation statement with the body of said template. However, to support proper instantiation, the task becomes a bit more advanced.

In order to support instantiation of templates, with all the concepts related to it, we will have to do some additional steps. For each instantiation statement we will have to perform the following steps on the body of the instantiated template.

1. Create a correctly scoped AST
2. Transform nodes to reference nodes
3. Perform the rename
4. Go back to the original AST
5. Merge classes

If the template that is being instantiated is not closed, we will perform the same steps to all the nested instantiations in a depth-first manner. 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.

5.5.1 Create a Correctly Scoped AST

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. We start off 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.

A scope is represented through the `Scope` class. 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 a variable (this covers both class attributes and other variables used throughout the program) or a class. Looking up symbols in the symbol table will always start in the called scope, looking for any references matching the given name. If we don't find any references with the given name we propagate the lookup to the parent scope. Given further

misses in the symbol table means that we will eventually reach the root scope, and if the root scope also doesn't contain any references then we fail the compilation and inform the programmer that there is a reference error in the program.

Having several layers of scope enables us to correctly handle shadowed variables, or parallel declarations with equal naming.

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`

We start of with a root scope which is given to the root node of the template body. We then traverse the tree and give every node a reference to the scope. When we reach one of the aforementioned nodes that should have its own scope, we create a new scope, with the current scope as the parent scope. This new scope is then given to all nodes beneath this node, until we reach another node in the list above.

At the end of this traversal we have an AST where every node has a reference to the scope its in.

5.5.2 Transforming Nodes to Reference Nodes

In order to rename a class or an attribute we also need to find all references to the class or attribute and rename these. To make this easier we have in this implementation instead transformed all references to *reference nodes*. 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. A reference can be either a variable reference or a class reference.

A variable reference can be a class attribute or any other variable declaration. Even though we call it a variable reference this does also cover functions. We don't need a different representation for functions, since functions in JavaScript are values which are assigned to a variable, and therefore share the same namespace, unlike Java where methods and variables can have the same name. These references are represented by a `Variable` class. The class contains the name of the attribute, and optionally what type it is. The type of a variable in our implementation is set through a very simple type-inference (relative to TypeScript's type system), where

we only look at if there is a `new`-expression assigned to the variable at declaration. If a variable is initialized with a class instantiation then the type of the variable is an object of that class. However, this simple type inference might not always correctly type variables, as variables might have explicitly declared types which are not typed with the class' type, however for simplicity, and in order to not reimplement TypeScript's entire type system, we currently ignore these in the implementation (though we will still perform a full type-check at a later stage in the compiler).

Class references are all references made to a class. These references are references such as class declarations, instantiations of a class, usage of a class as a type, etc. The `Class` class is a representation for class references. The class is an extension of the `Variable` class, so it can store the name of the class, however they also have references to all the variables that are instances of the class, and optionally a superclass, which is a reference to another `Class` instance.

With an understanding of what a reference is, let us look at how we can transform the AST nodes that are references. Transforming nodes into references mainly consists of two steps:

1. Transforming declarations
2. Transforming references

Transforming Declarations

When transforming declarations we both create the `Variable` or `Class` instance, register them in the scope they were found, and transform the identifier in the declaration to a reference. This reference is a special AST node. It is represented by the `RefNode` class. This class contains the same fields as other scoped AST nodes, `type` (which is always `"variable"` for reference nodes), `text`, `children` and `scope`. In addition to these fields, the `RefNode` class contains two additional fields for dealing with references. The first field is a reference to the `Variable` or `Class` instance that the `RefNode` is a reference to. The second is a field containing the original type of the node, which is used when transforming the `RefNodes` back to the original AST.

When we are transforming both the declarations and the references we need to pass through the AST multiple times. A pass through the AST will visit every node and perform a transformation. The task of transforming declarations requires us to do three passes through the AST:

1. Class declarations
2. Class heritage
3. Class attribute declarations

During the first pass through the AST we register and transform all class declarations, creating the `Class` instances. While creating the `Class`

instances we also register the associated `this` as a `Variable` in the class' body scope. Listing 5.2 on the next page shows an example of how a class declaration will be transformed in this step. In the listing we can see that the `"type_identifier"` node has been transformed to a `RefNode` instance.

For the second pass we register class heritage. Here we will visit all the class heritages and update the identifiers contained in the extends clause with `RefNodes` containing references to the appropriate `Class` instances found in the scope. For class declarations in which we found the class heritage nodes we will update the associated `Class` instance's superclasses. Setting these superclasses will ensure that later references to inherited attributes can be referenced and renamed properly.

The final step takes care of any class attribute declarations. This could have been done in the same pass as with class heritage, however it was separated to make the code more readable and the flow of the transformations more understandable. During this transformation we also perform a very simple form of type inference by checking if there is a `new`-expression in the assignment. For the cases where there is a `new`-expression we assign the type of the variable to be an instance of the constructed class. It is worth noting that the current implementation only transforms public field definitions. Other declaration types could be supported in the same manner as with public field definitions, however due to a lack of time these were not implemented.

Transforming References

Transforming references also has several passes, more precisely two passes.

1. Transforming `this` keyword
2. Transforming the rest

While transforming `this` references does not need to be done in a separate pass, it does simplify the process a little for next pass, as we do not have to worry about the `this` keyword as a special case for the other transformations. The `this`-nodes are transformed to a `RefNode` like all other references, and are simply a reference to the `Variable` instances for `this`, which we created in the class declaration pass.

In the second pass the rest of the references are transformed. Currently, the implementation will transform `new` expressions, member expressions, and type annotations. These are all transformed in a pretty similar manner, where the identifier is found and looked up in the attached scope. If we can't find it in the scope this usually means that it is something we can't rename, and is therefore not of interest, such as `console.log`, `number`, etc.

5.5.3 Perform the Rename

With all declarations and references transformed into reference nodes we can now perform the rename. For each class rename we can simply look

```
// Before transformation
{
    type: 'class_declaration',
    text: 'class A ...',
    scope: <Scope 1>,
    children: [
        {
            type: 'class',
            text: 'class',
            scope: <Scope 1>,
            children: []
        }, {
            type: 'type_identifier',
            text: 'A',
            scope: <Scope 1>,
            children: [],
        }
        ...
    ]
}

// After transformation
{
    type: 'class_declaration',
    text: 'class A ...'
    scope: <Scope 1>,
    children: [
        {
            type: 'class',
            text: 'class',
            scope: <Scope 1>,
            children: []
        }, {                                    //
            type: 'variable',                   // This node
            text: '',                           // has been
            scope: <Scope 1>,                   // transformed
            children: [],                       // to a RefNode
            origType: 'type_identifier',        //
            ref: <Class 1>                      //
        },
        ...
    ]
}
```

Listing 5.2: AST of a class declaration of class A before and after transforming the references. The values surrounded by angle brackets are references to `Scope`/`Class` instances.

up the old class name in the root scope and change the reference to the new class name. For class attribute renames we do a lookup on the specified class' scope. If the attribute can't be found in the class' scope we do a lookup in the optional superclass' scope. If either the superclass is missing or we can't find it in its scope either we throw an error, informing the programmer that the attempted renaming can't be performed.

5.5.4 Go Back to the Original AST

After we have performed the renaming we can go back to the original, simpler, AST. For most nodes this is as simple as removing the scope property. For `RefNodes` we have to create new AST nodes. Fortunately we stored the original type of the `RefNode`, which means we only need to fill in the `children` and `text` properties. The `children` property will always be an empty array, since all `RefNodes` are also leaf nodes. The `text` property will be filled with the name of the variable or class it is a reference to. This name can be found through the reference stored in the `RefNode` to either a `Variable` or `Class` instance.

Once we have traversed the tree and done the transformations as described we will end up with an AST tree similar to what we had initially, where all instantiation statements have been replaced by the renamed bodies of the templates they were declared to instantiate.

5.5.5 Merge Classes

After we have performed the instantiation and have returned to the original AST format we can finish off the last task necessary to close a template, namely class merging. At this stage we will work on a package/template body which will mostly consist of a (potentially) valid TypeScript program, with the exception of possibly `addto`-statements and duplicate class declarations. These `addto`-statements and duplicate class declarations will in this step be merged to form new classes, and remove the final construct of PT before the body is a TypeScript program. Merging of classes mainly consists of performing the following tasks:

- Grouping class declarations and `addto`-statements
- Verify the validity of the groups
- Merging the bodies of the classes
- Merging the class declarations
- Replace the old classes with the new

Grouping Class Declarations and `addto`-statements

Before we can start merging the classes we need to first group the classes and `addto`-statements which are going to be merged together. We will do this through collecting all classes and `addto`-statements in the first layer of

the package/template body. These will then be split up into groups based on the class name.

Verify the Validity of the Groups

Before we can merge these groups together, we first need to check that doing so is valid. This consists of checking that there is at least one class declaration in each group, as we can not perform additions to a non-existent class.

As we discussed earlier in section 2.1.2 on page 15 another requirement we should check before merging these classes is that if some classes in a group have external superclasses, then these superclasses need to be the same in order to avoid indirect multiple-inheritance. However, we will not trouble ourselves by checking this, as this will be picked up in the type-check stage of our compiler anyways.

Merging the Bodies of the Classes

Now that we have groups of classes that should be merged, and these groups are valid, we can then proceed to composing new classes from these. The first step to this is to create the new bodies of the new classes. This task is relatively simple as we just combine the AST nodes contained within each class declaration's or `addto`-statement's body.

The `addto`-statements would require some more advanced merging in order to support overriding methods, however this was not implemented due to the restricted time of the project. This will be discussed further in section 5.9 on page 59.

Merging the Class Declarations

Merging of class declarations is similar to merging the bodies, however we here instead merge the class heritage. That is if there are classes with class heritage we will merge the optional `extends` and `implements` clauses, inserting comma nodes between each member. This step might produce class declarations with multiple superclass, however this will be picked up by the type-check later on.

Replace the Old Classes With the New

Now that we have composed new classes from the merging of the old classes we can insert these back into the AST. In order to somewhat resemble the original program we replace the first class declaration within the group with the new class, while all other members of the group are removed from the AST.

5.6 Type-checking of Templates

After the previous step we have a program where all packages and templates are closed, meaning that the bodies of these should contain plain TypeScript. Because of this we can relatively easily type-check each package/template individually by using the TypeScript compiler and its compiler API. In order for us to type-check our pacakges/templates we will have to transform the bodies of the packages/templates into a textual format. This will be done by applying our code generation implementation, which we will discuss in section 5.7, to the body of the package/template we are currently working on. Running code generation on the package/template body will give us a TypeScript program. This program can then be passed on to the TypeScript compiler for type-checking. We will make the TypeScript compiler transpile the program to JavaScript without emitting any output. This will effectively type-check the program.

If the TypeScript compiler throws any errors we can log this for the user of our compiler to fix, and inform in which package/template this error occurred. If no errors were thrown we have a type-safe package/template, and we can then proceed to the next step in our compilation.

5.7 Code Generation

After performing these steps we can finally produce the output. Producing TypeScript output is a relatively simple task. By traversing the AST we can concatenate the text from each leaf node with whitespace between each leaf node's resulting textual representation. This will produce quite ugly, unformatted code, but as long as the contents of the closed packages and templates are valid TypeScript programs, then the generated code will be so as well. In order to make it more readable we perform an extra step before writing the output to the specified file, a formatting step. For formatting our generated code we will be using *Prettier*[5] for our implementation as it is relatively simple to use. Running our produced source code through the Prettier formatter produces a nicely formatted, readable output.

The TypeScript output is probably the best target for understanding what the PT mechanism does, however it might not be the best output for production use. Since the only officially supported language for the web is JavaScript we will also be implementing this as a target for code generation. This is fortunately also a relatively simple task, as we already depend on the TypeScript compiler, and since we are able to produce TypeScript source code, we can use this to produce JavaScript output.

[5] A code formatter for the web ecosystem. See https://prettier.io/.

```
============================
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 5.3: Example of Tree-sitter grammar test

5.8 Testing

Testing has been an essential tool throughout the development of the compiler. After the initial prototype of the compiler was running I continued onward with test driven development. This allows me to write up tests for all the features of the language, and run them concurrently as I make the changes to the implementation.

5.8.1 Lexer and Parser

For testing the lexer and parser I used the built in testing framework. 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*[6]. These files are placed in the `corpus` folder, and will be automatically executed when running `tree-sitter test`.

Listing 5.3 shows an example of a Tree-sitter grammar test. This example is taken from the source code of the PTS parser, where we test template declarations.

[6]S-expressions are textual representations for tree-structured data. See [23] for additional information and examples

```
import test from 'ava';
import transpile from '../src';

test('Transpiles closed templates to nothing', (t) => {
    const program = `
template P {
    class A {
        i = 0;
    }
}
`;

    const expected = ``;
    const result = transpile(
        program,
        { emitFile: false, targetLanguage: 'ts' }
    );

    t.is(result, expected);
});
```

Listing 5.4: Example of a test for the PTS compiler

5.8.2 Compiler

For testing the implementation of our compiler I chose to make similar tests to those used in the parser. I used the test framework AVA[7] to run my tests. I chose to structure my tests into separate files under the `__test__` folder. This is not a strict requirement for AVA, but helps structure the project. All my tests for the compiler are implementation tests, where I transpile a program and check that the output is equal to the expected output. AVA has helper methods for this where I can use the `is` function to check that two strings are equal to each other. On differing strings an informative error is logged, with the differences between the actual results and the expected results.

Listing 5.4 shows one of the tests from the source code of the compiler, where we similarly to the Tree-sitter test we saw before, check that a closed template is working as expected. Instead of providing a parse tree as the expected result, we instead provide an expected program as the string. Since templates are not supposed to produce any code after being transpiled, we therefore also expect this program to be an empty string.

[7] https://github.com/avajs/ava

5.9 Completing the Implementation

While we have a working compiler for most of the language, we did not have enough time to implement all the desired functionality. What still remains to be implemented are handling `addto`-statements properly, and handling all declaration and reference types. We are also currently not supporting splitting up the program into several files. In the following sections we discuss how implementing these features could be carried out.

5.9.1 `addto`-statements

`addto`-statements are one of the core features of PT I was not able to finish in time. As of now `addto`-statements for the most part work as intended, however the method overriding capabilities of `addto`-statements was not implemented.

There are mainly two ways for us to finish the implementation of `addto`-statements as of now, one hacky but cheap implementation, and one more robust but expensive implementation. The cheap and hacky way is to simply always merge the bodies of the `addto`-statement at the bottom of the formed merged class. This will work because of the JavaScript's prototype-based object-orientation. Since the class syntax is only syntactic sugar for creating a prototype object and a constructor function, this means that any attributes that are lower down in the prototype object will override any previously declared attributes. So if we in a `addto`-statement wish to override an attribute we will utilize overriding of properties in objects to achieve this. The hacky part of the implementation is that TypeScript gives errors for duplicate declarations inside the class syntax, as this is often a sign of an erroneous program. In order for us to still be able to pass the type-check we will have to bypass the TypeScript compilers errors for certain lines by prefixing all attribute declarations in the bodies of `addto`-statements with a `// @ts-ignore`. This will unfortunately have the drawback of not having any type-checks for these function-declarations.

The better more robust way to fix this will force us to make a quite severe refactor of the majority of the codebase. With this approach we will have to create more complicated datastructures for classes and class attributes. With our current implementation we are simply combining two AST trees when merging classes, not really worrying about what is contained within these trees. In order for us to be able to override attributes with the `addto`-statements we will have to merge the bodies of classes in a smarter way. I suggest that before class merging, the AST nodes representing classes or `addto`s are transformed, as well as their contained attribute declarations. Class nodes can be transformed to a node containing a body of attribute nodes, instead of general AST nodes, and the attribute declarations will be transformed to a pair of the name of the attribute, and the old AST representation of the attribute. This will enable us to perform smarter merges, as we have easy access to the contained attributes. For class declarations this will enable us to give better error messages when merging

classes resulting in duplicate attribute declarations, and for `addto` this enables us to replace the attributes that will be overridden.

This more robust implementation was attempted, but had to be discarded because I ran out of time. The attempted implementation can be found under the experimental branch in the GitHub repository, https://github.com/petter/pts/tree/experimental. This implementation also tries to achieve better representations for templates and packages to contain these new class representations.

5.9.2 Supporting All Attribute Declarations

As of now we only support public field definitions, such as `a = 2` for instance, and simple function declarations, such as `f() { ... }`. There are of course more ways that attributes can be declared, such as the function syntax, `function f() { ... }`, however this was not prioritized during implementation. I believe that the task of implementing the remaining attribute types will not be a too expensive task, as these could be implemented similarly as with the already implemented attribute declarations. If someone ought to finish this task the TypeScript grammar is very useful for finding all the different possibilities that an attribute could be declared.

5.9.3 Supporting All References

This task is probably the hardest to finish off. In order to support all reference types we would presumably have to do more advanced semantic analysis of the program than we are currently doing, in order to correctly identify references. One should then ask themselves if it is worth it to continue on with this approach, or if starting off fresh with a fork of the TypeScript compiler would be a better choice, as we would likely get the semantic analysis for free.

5.9.4 Supporting Multi File Programs

This could likely be implemented by reading all files that are declared in the import part of the main program. We could then replace the import statement of templates with the contained templates. This could however of course lead to duplicate template names, so a complete implementation might have to offer some template aliasing on imports to circumvent this.

Chapter 6

Using PTS

Now that we have a working implementation of our compiler for PTS, let us look into how we could install and use it. There are mainly two ways of using the PTS compiler:

- Installing it globally, or
- Creating a PTS project

In the following sections we will look at how you can install and use the compiler for both approaches.

The PTS compiler requires you to have Node and npm installed on your computer. For instructions on installing Node and npm I refer the reader to the npm documentations[1].

6.1 Installing and Using PTS Globally

Installing PTS globally will enable you to use PTS anywhere, and might be favorable if you are planning to create several smaller projects to test it out, or if you are not too experienced with the node ecosystem. If you want to install the compiler globally you can do the following:

```
$ npm install -g pts-lang
```

This will give you access to the PTS compiler CLI through the command `pts-lang`. By giving the `--help` flag you will get some useful information for how to use the compiler.

```
$ pts-lang --help
Options:
      --help                    Show help
      --version                 Show version number
  -i, --input                   Name of the input file
  -o, --output                  Name of the output file
```

[1] https://docs.npmjs.com/downloading-and-installing-node-js-and-npm

```
-v, --verbose                    Show extra information during
                                 transpilation
-t, --targetLanguage, --target   Target language for
                                 transpilation
-r, --run
```

6.2 Creating a PTS Project

If you are using PTS for a specific project it might be better to set it up as a project dependency in npm. When installed in an npm project the CLI is available to use through npm scripts or through accessing it directly from the `node_modules` folder in your project. The compiler can also be accessed through the API by importing it as with any other npm package.

Installing it inside an npm project will not require you to install it globally, as it will stay contained in the project. This also means that any contributors of the project will not have to worry about installing PTS, as it will be installed when the project is set up.

To initialize an npm project you can do the following:

```
$ mkdir <project name>
$ cd <project name>
$ npm init -y
```

With a project set up you can install the PTS compiler as following:

```
$ npm install pts-lang
```

With the PTS compiler installed in the project you can then set up some scripts in your project's `package.json` to start and/or build the project. Below you can see an example of a section of a `package.json` file with scripts for running and building a file:

```
{
  "scripts": {
    "start": "pts-lang -i src/index.pts --run",
    "build": "pts-lang -i src/index.pts -o build/index"
  }
}
```

The start script only runs the program, and does not emit any files, while the build script transpiles the `src/index.pts` file to JavaScript. If you would rather have TypeScript output you can use the `-t` flag to specify this:

```
pts-lang -i src/index.pts -o build/index -t ts
```

6.3 A "Real World" Example

Now that we understand how to get PTS set up, let us look at how it could be used in a real world example, and how PT enables the programmer to modularize the code base even further giving great flexibility. Note that the following example will not work with the current state of the compiler as it doesn't handle member expressions containing call expressions, such as `f().i`. Properly handling these types of member expressions would require us to analyse the function for its return type. The example serves as an example of how PTS could be useful given a more complete implementation for a real world problem.

The most common use of TypeScript is to create web applications. Let us look at how PTS can help make this task easier for the programmer. We will try to create a simple web application for displaying a Pokémon. To do this we will use one of the most popular web frameworks, React[2].

React is a web framework developed by Facebook[3]. It aims to make creating scalable web projects easier to handle, through enabling the programmer to modularize collections of elements into *components*. These components are often created to make re-use of common elements easier, such as creating a styled button with certain features, or we could create a component to represent the entire web application.

Displaying one specific Pokémon can be pretty simple, however we would like to create a React component that can display information and a picture of any Pokémon. We do not want to have to write down information about all Pokémon, so we will fetch this information from an API, more specifically the PokéAPI[4]. This API lets us fetch data about all Pokémon.

We will start off with the task of fetching data. As this is something you commonly want to do in web applications it might be a good idea to separate this logic into a separate template:

```
template FetchJSON {
    class FetchJSON extends Component {
        componentDidMount() {
            fetch(this.props.url)
                .then(response => response.json())
                .then(data =>
                    this.setState(state => ({...state, data}))
                ).catch(error =>
                    this.setState(state => ({...state, error}))
                );
        }
    }
}
```

[2] https://reactjs.org/
[3] https://www.facebook.com/
[4] https://pokeapi.co/

The component we see above will fetch whatever URL we pass to it in its props and update the state with the results of the fetch. If we for some reason should fail to fetch the data we will instead update the state with the error message we got.

In addition to fetching data, it might be useful to have a logger, which will log all state changes to the console. This is often useful when working with React components as we are able to see when they update, and what the state was at the time of the update. Such a logger could then also be separated into its own template, like the following:

```
template StateLogger {
    class StateLogger extends Component {
        componentDidUpdate() {
            console.log("State updated!", this.state);
        }
    }
}
```

Finally we would like to combine these into our Pokémon component, and add some logic for displaying the information. We will do this inside of a package, so that this will produce an output:

```
pack Pokemon {
    inst FetchJSON { FetchJSON -> Pokemon };
    inst StateLogger { StateLogger -> Pokemon };
    addto Pokemon {
        render() {
            if(this.state.error) {
                return (
                    <div>
                        <h1>An error occurred</h1>
                        <p>{this.state.error.message}</p>
                    </div>
                );
            }

            if(this.state.data === undefined) {
                return 'Loading...';
            }

            const name = this.state.data.name;
            const pokemonTypes = this.state.data.types;
            const image = this.state.data.sprites.front_default;
            return (
                <div>
                    <img src={image} />
                    <h1>{name}</h1>

                    <h2>Types</h2>
```

```
                    <ul>
                        {pokemonTypes.map(pokemonType => (
                            <li>{pokemonType.name}</li>
                        ))}
                    </ul>
                </div>
            )
        }
    }
}
```

We could then use our Pokémon component in our application by supplying a URL for the Pokémon to display, as seen below:

```
class App extends Component {
    render() {
        <Pokemon
            url="https://pokeapi.co/api/v2/pokemon/ditto" />
    }
}
```

In this example PT helped us split up our component into several smaller modules. This enables us to later re-use these common pieces of functionality, fetching data and logging, in other components, which will make our project more scalable, and aids in shortening development time. Modularizing these concepts also helped make the implementation of our Pokémon component less cluttered, which makes the readability of the code better.

Part III

Results

Chapter 7

Evaluation and Discussion

7.1 Does PTS Fulfill The Requirements of PT?

This thesis is concerned with 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 [13]. After getting an understanding of the requirements we are going to look at how our implementation holds up to them.

7.1.1 The Requirements of PT

In [13] 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. These requirements can therefore be used to evaluate our implementation and determine whether our implementation can be classified as a valid implementation of Package Templates.

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.

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

    class B {
        A a = new A();
        ...
    }
}

package P {
    inst T;

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

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

Listing 7.1: Example of parallel extension in PT. 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.

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 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 within the same package/template, without having to use casting to access the new members. We can see an example of this in listing 7.1, where an addition to class `B` is referencing the added method of class `A`. Contrast this to how an implementation with traditional extension through subclasses of `A` and `B` would work, and the casts etc. necessary to make the call `a.someMethod()` work in that context. The order of additions does not affect the parallel extension.

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` is 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.

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.

Multiple Uses

PT should 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 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 re-used. An example of this is the graph template we created in listing 2.3 on page 10. 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 system, however we might later also want to re-use 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.

Type Parameterization

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 type during declaration of a class, and the actual type can be given when a new object of the class is created. Type parameters in Java can have constraints where the concrete type must extend another class or interface. Similarly, type parameterization in PT also enables the programmer to assume the existence of a type, however here the type parameter is accessible to the whole template. PT type parameters can also be constrained similar to Java, by giving a nominal type it should extend, however PT also allows for type constraining through structural types, by giving a structure the type should conform to.

In listing 7.2 on the next page we see an example of how type parameterization can be used to implement a list. Type parameterization in PT might feel similar to how you would use it in regular Java, however having the type parameter at the template level, where Java has it at class level, does have some advantages. One advantage of having the type parameter at

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

Listing 7.2: Modified example from [13] where type parameterization is used to create a list implementation.

template level is that you don't need to specify the actual parameter again after instantiation. At instantiation of the `ListsOf` template we can give i.e. a `Person` class, containing some information about a persons name, date of birth, etc., as the actual parameter, and then we would not have to keep specifying 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 an inner class of the `List` class.

Class Merging

PT should allow for merging 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 of `B`'s attributes from an `A`-variable, and vice versa.

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.

7.1.2 PTS' Implementation of the Requirements

With a proper understanding of the requirements of PT we can evaluate our implementation to check whether it fulfills these requirements.

Parallel Extension

Fulfilling the requirement of parallel extension requires us to be able to merge or make extensions to classes, and be able to reference these additions within the same package/template, in the same or in a different class. To understand how this requirement can be fulfilled it is important to understand how the requirement could fail to be fulfilled. A failure to fulfill the requirement would be that making additions in parallel would fail to compile, or create an otherwise incorrect program.

Failure to compile might of course not always be a bad thing, there are certain scenarios where we do want the compiler to throw an error. There are mainly two scenarios where we would like the compilation to fail for additions:

- trying to make an addition to a non-existent class, and
- trying to reference non-existent attributes in a class.

The first scenario where our compiler should fail is when we are making additions to a non-existent class. This will be caught in the class merging part of our compiler. In the class merging part of the implementation the compiler will group all class declarations and additions by the class name. If there is a group containing only additions then it will fail, as we have no class to make additions to.

The second scenario is when we are trying to reference non-existent attributes in a class. An example of this can be seen in listing 7.3 on the following page. This example will fail during the type-checking of our packages/templates, as discussed in 5.6 on page 56. Our approach for dealing with this is pretty much by not dealing with it, and instead assume that everything is okay at this stage of the compilation. We will then instead discover any inconsistencies in the type-checking stage of the compiler. In the aforementioned listing it is of course pretty easy to examine class `A` to see if it contains an attribute `h`, however it might not always be this easy. In a more complicated example where we are in the process of merging several classes and additions it might prove a tougher task to see if the addition would result in a type-safe class. So as long as we are able to perform the addition we can assume that it is working as intended and instead let the TypeScript compiler check if it is type-safe, after the addition has been performed.

Now that we understand when we want compilation to fail let us look at where we do not want it to fail, namely when we have a valid parallel extension. One way it could result in an unwanted failure is if we tried to check if the addition contains any invalid references or type errors. This could commonly happen if we are trying to check the addition's references to the declared class. However, as discussed above, checking if a reference to an attribute is valid is quite tricky, and in our implementation we instead leave this up to the TypeScript compiler in the type-check stage. By doing this we will not incorrectly throw any false-negatives when it comes to

```
template T {
    class A {
        function f() {
            return 1;
        }
    }
}

package P {
    inst T;
    addto A {
        function g() {
            this.h();
        }
    }
}
```

Listing 7.3: An example program that should fail during compilation, where we are trying to reference a non-existent attribute, `h`, in an addition to class `A`.

parallel extensions. This approach does unfortunately come with some downsides. By not addressing the issue at the addition stage it makes it harder to give informative error messages when invalid references do occur, however this was a tradeoff that was beneficial for this project.

Our implementation handles parallel extension as expected, it fails when it should fail and succeeds when it should succeed. It also doesn't produce any false-negative failures, failing where it shouldn't fail or succeeding when it should fail. Therefore our implementation fulfills the requirement of parallel extension.

Hierarchy Preservation

In order to fulfill the hierarchy preservation requirement we have to preserve all super-/subclass relations after additions and merges have been applied. Listing 7.4 on the facing page shows a program, and the resulting TypeScript program after compilation, which adheres to the requirement of hierarchy preservation. In the example class `B` is still a subclass of class `A` after both a merge and an addition is made to `B`, which is what we expect. However, this doesn't prove that our implementation will fulfill the requirement of hierarchy preservation, let us dig deeper into how our implementation handles these scenarios.

As we talked about briefly in section 5.5.5 on page 54 when we merge classes we make sure to also merge their class heritage, combining the extending classes and implementing interfaces of the different classes. This means that we might end up with instances where we are extending multiple different classes, however this will then be picked up in the type-

```
// PTS
template T1 {
    class A {
        i = 0;
    }

    class B extends A {
        f() {
            return this.i;
        }
    }
}

template T2 {
    class C {
        j = 0;
    }
}

package P {
    inst T1;
    inst T2 { C -> B };
    addto B {
        k = 0;
    }
}

// Resulting program
class A {
    i = 0;
}

class B extends A {
    f() {
        return this.i;
    }
    j = 0;
    k = 0;
}
```

Listing 7.4: Example showcasing the preservation of super-/subclass relations

checking stage of the compiler. If we had not merged class heritage, we could have ended up breaking the inheritance hierarchy in the aforementioned listing, as we could have for example ended up with class `C`'s heritage, which does not have a superclass. Because of the heritage merging we can with confidence say that we have fulfilled the requirement of hierarchy preservation, as we always preserve all super-/subclass relations.

Renaming

In order to be able to fulfill the renaming requirement our implementation should be able to rename classes and their attributes. This renaming should result in a program where not only the declarations have been renamed, but also all references. Listing 7.5 on the facing page shows a program with an example of renaming in PTS, where we are renaming a class, `A`, and the class' attribute, `i`. We can see in the resulting program that the identifier in the declaration of both the class and the attribute has changed, as well as the references to these in the constructor of the class and references in another class, `B`. The renaming has also not wrongly renamed other references that are similar in naming, such as the parameter of the constructor of class `A`.

Because of TypeScript's structural type system, renaming will have to work a bit differently than it would in a nominally typed language. In nominal PT, all references are always renamed. In structural PT we could end up with programs where some references can not be renamed, as some variables might not be directly tied to a class. Listing 7.6 on page 78 showcases this problem. The problem arises in the `a` attribute of class `B` in template `T`. This has been declared to be a variable expecting an object where there exists an attribute `i`. The variable `a` is initialised with an object of the class `A`. This is fine in template `T`, however when `T` is instantiated in package `P`, and `A`'s attribute `i` is renamed to `j`, this is no longer the case. The `a` variable in class `B` is not a direct reference to class `A`, and we can thus not rename the reference to the attribute `i`. Since an object of `A` no longer contains an attribute `i`, this is no longer a valid value for variable `a`.

The aforementioned listing shows how we would like the compilation result to look like, however this is not the result the current implementation produces. TypeScript's type system can be quite complicated, and due to a lack of time I chose to ignore most of the type declarations. The current implementation would have treated the attribute `a` as an `A`-variable, since it is being initialized with an object of `A`, and therefore have renamed later references of `a.i` to `a.j`. It was more important to get a working prototype, than support all scenarios with different type signatures. This is something I would of course have liked to take into consideration if I had more time to spend on the implementation. Deciding the type of variables is something that possibly would have come for cheaper if I had opted for a fork of the TypeScript compiler as my approach. This is something we will come back to in 8.2.1 on page 90.

```
// PTS
template T {
    class A {
        i = 0;
        constructor(i: number) {
            this.i = i;
        }
    }

    class B {
        a = new A();
        function f() {
            return a.i;
        }
    }
}

pack P {
    inst T { A -> X (i -> j) };
}

// Resulting program
class X {
    j = 0;
    constructor(i: number) {
        this.j = i;
    }
}

class B {
    a = new X();
    function f() {
        return a.j;
    }
}
```

Listing 7.5: Example of renaming in PTS

```
// PTS
template T {
    class A {
        i = 0;
    }

    class B {
        a : { i : number } = new A();
        i = a.i;
    }
}

pack P {
    inst T { A -> A (i -> j) };
}

// Expected result
class A {
    j = 0;
}

class B {
    a : { i : number } = new A();
    i = a.i;
}
```

Listing 7.6: Example showcasing the problem of having renaming in a structural language. In class `B` we have an attribute, `a`, that expects an object that contains an attribute `i`. The attribute is initialized with an `A` object. This is fine in template `T` as `A` contains an attribute `i`, however when class `A`'s attribute is renamed in the instantiation in package `P` then an object of `A` is no longer valid as a value, since it no longer contains an attribute `i`. This is an instance where we can't just rename the references to `i`, since this reference isn't explicitly related to `A`.

```
template T {
    interface I {
        void f();
    }

    class A implements I {
        void f() { ... }
    }
}

package P {
    inst T with I => I (f() -> g());
    // Error: A is not abstract and does not
    // override abstract method f() in I
}
```

Listing 7.7: Example showing how a renaming of an interfaces' attributes could result in an invalid program.

PT for the nominally typed language Java is also not perfect in this regard, as we can run into issues with renamings of interfaces' attributes. If a class implements an interface, and we in a later instantiation rename the interfaces' attributes, and not the attributes of the implementing class, we could end up with a program where the class no longer fulfills the requirement of the interface. In listing 7.7 we can see this problem in action, where after a rename of interface `I`'s method `f`, the class `A` no longer implements the interface properly. This is of course also a problem in structural PT, and can potentially be an even bigger problem as we would not necessarily have an explicit relation between the interface and the class, and could therefore also not properly describe the error to the programmer.

As we have discussed, renaming does work to some extent, but will also in a lot of cases not work because of the implementations' lack of handling explicit types. The renaming requirement is therefore only partially fulfilled.

Multiple Uses

In order for this requirement to be fulfilled we should be able to re-use a template several times, with different renamings and additions while the different instantiations stay independent of each other. This was something I paid extra attention to during implementation, not just to fulfill the requirement, but to avoid bugs. I solved this by making sure that while transforming the AST this would be done in an immutable fashion. In order to test this we will be creating a simple program where we instantiate the same template more than once and see if the resulting program is as expected. The program can be seen in listing 7.8 on the following page. The program comprises a template `T` with a single class, `A`, with an attribute

```
// PTS
template T {
    class A {
        i = 0;
    }
}

pack P {
    inst T { A -> B (i -> j) };
    inst T;
    inst T { A -> A (i -> x) };
}

// Resulting program
class B {
    j = 0;
}

class A {
    i = 0;
    x = 0;
}
```

Listing 7.8: A program showcasing multiple uses in PTS, and the resulting program in TypeScript at the bottom.

`i`. This template will then be instantiated three times, where we first will be renaming the class and field, then instantiate without renaming, and finally instantiate it with just an attribute renaming. The expected program should have two classes, one class `B`, with an attribute `j`, and a class `A` where the two bottom instantiations should have created a merged class with attributes `i` and `x`. We can see from the resulting program after a successful compilation that this is as expected.

From the aforementioned listing we can see that re-using templates will be kept independent of each other, and we therefore fulfill the requirement of multiple uses.

Type Parameterization

The type parameterization requirement is something the implementation does not fulfill. This was not implemented due to it not being prioritized. There is only so much time available during the span of a master thesis, and I chose to look at how the core of PT would fit into a structurally typed language like TypeScript, rather than on making sure it would be a fully fleshed out implementation of PT. Another reason for avoiding this is that much of type parameterization can be achieved through merging and renaming. Listing 7.9 on the next page shows an example of how you can

```
template ListsOf {
    class E { }
    class List {
        AuxElem first, last;
        void insertAsLast(E e) { ... }
        E removeFirst() { ... }
    }
    class AuxElem {
        AuxElem next;
        E e;
    }
}
```

Listing 7.9: Example of a similar list implementation as in listing 7.2 on page 72, without the use of required types. Instead of giving a type for the required type we will have to merge the class `E` with the "actual parameter".

use an empty class as a generic type implementation of lists, similar to the list implementation with required types in listing 7.2 on page 72. Required types do of course have a lot of advantages such as making it possible to constrain the type, and forcing the programmer to give an actual parameter for the type, which we are unable to do.

Class Merging

In order to fulfill the requirement of class merging we will have to be able to merge classes through instantiating two or more templates containing a class with a common name, or through renaming a class at instantiation leading to two classes having the same name. A failure to fulfill this requirement could be that a resulting program would contain two different class declarations, where these should have been merged. This could occur if we had for instance not respected the renaming of a class at the time class merging, that is if we had renamed a class to be merged, but this was not picked up during class merging. Our implementation of class merging has been made so that it is essentially unaware of the instantiation and renaming steps, it is done as a separate step after these actions have been performed. What this entails is that after performing all instantiations, with optional renamings, we can momentarily end up with several class declarations for the same class, which would be an invalid TypeScript program, however before the closing of templates step is done we will merge the class declarations and `addto`-statements representing the same class, forming new classes. The resulting program could of course still be an invalid TypeScript program, as the merge of classes could lead to erroneous code, such as duplicate function declarations, however this is not a result of an invalid implementation, but rather an error as a product of an invalid input program. The class merging step will merge classes without checking their bodies, as these will be picked up in the type-checking step later on. By separating the logic of instantiations (and renaming) and class

```
// PTS
template T {
    class A {
        i = 0;
    }
}

pack P {
    inst T;
    inst T { A -> A (i -> j) };
}

// Resulting program
class A {
    i = 0;
    j = 0;
}
```

Listing 7.10: Example of class merging in PTS, where we merge two classes, `A`, with attributes, `i` and `j`, respectively

merging, as we have done, we should minimize the potential for an invalid implementation of class merging.

Listing 7.10 shows a program with class merging, and the resulting program from a compilation of the PTS program. In the program we instantiate the same template twice, but rename the only attribute, so that the resulting program doesn't contain duplicate declarations. The instantiations will produce one class `A` with the attribute `i` and another class `A` with the attribute `j`. These are then merged together during compilation. The resulting program is as expected a single class `A` with both attributes.

Because of our separation of logic between the instantiations and the class merging we will always catch all merged classes, as long as the instantiation process was successful. The fulfillment of this requirement therefore relies heavily on the fulfillment of the renaming requirement. As we discussed earlier, the renaming requirement is only partially fulfilled, however the only important feature of renaming for this requirement is that the class declarations are always renamed correctly, which they are. Therefore the class merging requirement is fulfilled.

Collection-level Type-checking

To fulfill this requirement our implementation just needs to perform a type-check on each package/template individually. In the type-checking step of our compiler we do just this, and we therefore fulfill this requirement.

However, as we previously touched upon in section 4.1.1 on page 31, TypeScript's type system is unsound. What this means is that TypeScript's

type-system will not be able to pick up all type errors. With an unsound type-check, can we truly say we have fulfilled the requirement of collection-level type-checking?

In the proposal of PT, Java is used as the host language. Java has also been shown to have an unsound type system [1], so we could argue that if the requirement is fulfilled in PT, then our unsound type-check should also fulfill the requirement.

7.1.3 Conclusion

While we do not fulfill every requirement, we do fulfill most of them. The current implementation might not be a full implementation of PT, but we can confidently say we have at least made an implementation of the core of PT for TypeScript. Not having a full implementation does mean that we might not be able to examine all the differences between our implementation and PT, however we will be able to examine the common elements, which covers the most interesting parts. This allows us to explore how a mechanism like PT fits with the TypeScript language, and its potential utility.

7.2 Nominal vs. Structural Typing in PT

One of the most notable differences between PTS and PT are the underlying languages' type systems. PTS, as an extension of TypeScript, has structural typing, while PT on the other hand, an extension of Java, has nominal typing. We will in the following sections try to understand what advantages (and disadvantages) these type systems have over the other, and how this affects Package Templates.

7.2.1 Advantages of Nominal Type Systems

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 (modulo co-/contravariance). 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 through some clever representation techniques [22]. We can see an example of subtype relations in both nominal and structural type systems, in a Java-like language, in listing 7.11 on the next page. 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`.

```
// 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 7.11: 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.

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 [22].

7.2.2 Advantages of Structural Type Systems

Arguably 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 arguably only has to look at the type to understand its meaning, 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 [22].

More General Functions/Classes

Malayeri and Aldrich performed a study (see [14]) 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.

7.2.3 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 7.12 on the following page.

Here the function `double` will consume an object that has a `calculate` attribute. The intended use is to consume something that does a calculation on the object and returns a number which will be doubled, while the unintended use example in this case does some unexpected side effect and returns a number as a status code. The unintended object can be consumed by the `double` function as it is conforms to the signature of the function, while in a nominally typed system this can be avoided to a much larger extent.

7.2.4 Which Better Fits PT?

Now that we understand the advantages and disadvantages of both categories of type systems, let us look at which type system would be more beneficial for PT.

PT's type parameters uses structural typing, independent of the underlying language's type system. Using structural typing was seen as a necessity for required types as this would give the mechanism its required flexibility. One could therefore argue that a structural type system is a better fit for Package Templates as it would remove the confusion of dealing with two different styles of typing in a single program, and make the language mechanism feel more like a first class citizen of the host language.

Another advantage for having structural typing for PT is that it can help strengthen one of the main concepts of Package Templates, namely re-use. As we learned from the study of Malayeri and Aldrich, structural typing can make programs more general which makes them more prone to re-use.

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

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

// function calculate also returns number, but as a status code
const unintended = {
    calculate: () => {
        doSomeSideEffect();
        return 1;
    }
}

double(vector); // Ok, intended

double(unintended); // Not intended use,
                    // but it is type-safe to do so.
```

Listing 7.12: Example of spurious subsumption in TypeScript

However, there are also some quite significant problems with having PT in a structural language. Renaming is especially something that might not fit nicely into a structurally typed language. In listing 7.13 on the next page we see an example of a program that breaks after renaming an attribute. The renaming resulted in class `Consumable` no longer conforming to the signature of function `f` in class `Consumer`. PT does not support changing the signature of functions so there is no way for us to be able to make the `Consumable` class conform. In order to avoid running into this problem we might consider disallowing inline type declaration. This would force us to give an interface as the type for the formal parameter, `consumable`. We could then also rename the members of the interface in order to once again make the `Consumable` class conform to the signature of function `f`.

It is worth noting that the problem of renaming causing programs to break is not something unique to structural typing, this can also occur in nominally typed programs. Listing 7.14 on page 88 showcases a program that breaks after renaming. In this listing we see that after renaming method `f` of class `A` the class no longer fulfill the requirements of the implementing interfaces `I`, as `I` expects a method `f` to be present, which it no longer is. We could resolve this by also performing an equal renaming to interface `I`. Although it is a problem in nominal PT as well, it is less so than with structural PT, since it can be resolved with just an additional renaming, and due to the relation being explicitly stated we can also give

```
template T {
    class Consumable {
        i = 0;
    }

    class Consumer {
        function f(consumable : {i: number}) {
            ...
        }
    }
}

pack P {
    inst T { Consumable -> Consumable (i -> j) };
}
```

Listing 7.13: Example of how using renaming in PTS might break a program. After renaming the field `i` to `j` the class `Consumable` is no longer consumable by function `f` in class `Consumer`.

an error message during compilation notifying the programmer of this inconsistency. In structural PT we would have to disallow inline-types in order to make the problem more solvable for the programmer, but due to the relation between the class and interface not necessarily being explicitly stated it would be harder to give a sensible warning for the programmer.

With the discussed general advantages and disadvantages of structural and nominal type systems, and the points brought forward in this section we can see that both styles of typing have strong use cases with PT. A nominal type system in PT would seemingly lead to less problematic renaming scenarios, while a structural type system in PT would arguably fit better with the overall theme of PT, flexible re-use.

```
template T {
    interface I {
        void f();
    }

    class A implements I {
        void f() { ... }
    }
}

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

Listing 7.14: Example of how using renaming in PT might break a program. After renaming the method `f` to `g` the class, `A`, no longer conform to the implementing interface `I`.

Chapter 8

Concluding Remarks

In this thesis we have discussed the approach to the project and the implementation of the compiler for the PTS language, as well as evaluated the implementation and discussed the consequences of having a structurally typed language as the host of PT. We will conclude this work by returning to our initial research questions and trying to answer these with the knowledge we have accumulated through the span of this thesis. After this we will have a look at what could have been done differently in retrospect, and finish off with proposing future works within the field.

8.1 Addressing Research Questions

RQ1: How does the language mechanism, Package Templates, fit into TypeScript?

Tenker at dette spørsmålet blir litt for bredt, og at svaret fort blir litt for subjektivt, så tenker nok å fjerne dette spørsmålet. Web bruker jo for tiden også særdeles lite klasser, da de fleste heller velger å uttrykke seg gjennom den funksjonelle delen av JS/TS, så det passer kanskje egentlig ikke så bra inn heller.

RQ2: Does structural typing change how the core of Package Templates works?

Most of the functionality of Package Template stay the same in a structurally typed language as they do in a nominally typed language. The most notable deviation is the renaming mechanism. While we still rename all valid references, it is worth noting that what a reference is will differ in a structurally typed language.

In nominal PT, all variables that are instantiated with an object of a class, will also have some explicit relation to the class. A variable in nominal PT will likely have either the class (or a superclass) as the type, or an interface. For the case of the variable having the class as the type the relation to

itself is obviously there. For interface or superclass as the type the relation must explicitly be declared in the class' heritage. If the class' attributes are renamed we will either be able to rename the variables references to these attributes if the renaming was applied to the class or any of its superclasses, or at least give a detailed error message for when the renaming was applied to the interface.

In structural PT, variables instantiated with an object of a class might not have the same explicit relation to the constructed class. A variable in a structurally typed language might be typed with a structure that the class conforms to. However, this conformity might break after a renaming has been applied to the class. In these cases we will not be able to rename the variables type, nor any of the variables references to the possibly renamed attributes, since these do not have the same explicit relation to the class.

RQ3: Will having PT in a structurally typed language have any notable advantages or disadvantages over having it in a nominally typed language?

As we discussed in section 7.2.4 on page 85 there are some benefits of hosting PT in a structurally typed language. Structural typing does fit nicely in with the theme of Package Templates, namely re-use. A structural type system gives the programmer a flexibility that nominal type systems can not offer to the same extent. One of the strongest points for structural typing is how it enables us to easily use third-party libraries without necessarily having to alter our classes. Say we have implemented a graph library in PT and later on wanted to use some third-party graph utility library. This library might take a graph as input and do some calculations such as finding the shortest path between nodes in the graph. In structural typing, as long as our graph implementation is structurally equal to the type of the input we can pass it with no problems. In a nominally typed language we might have to alter our implementation to explicitly declare that our classes implement some interfaces from the library. It might at worst not even be possible if the graph utility library has typed their input with classes instead of interfaces. This is of course one of the strengths of PT in the first place, being able to add implementing interfaces to a class without altering the original implementation or merging classes if possible, however the fact that we would not have to perform this step in a structurally typed language could arguably be seen as a benefit.

8.2 In Retrospect

8.2.1 Approach

While we were able to implement most of PT with our approach, I fear that further development to reach a complete implementation might be hindered by our chosen approach. This is largely because we might have to implement much of the type system in order to properly identify

references, as we briefly discussed in section 5.9.3 on page 60. Instead of re-inventing the wheel, we might be better off by implementing PT in a fork of the TypeScript compiler. While this *might* make updates harder, than with our implementation, we will most often likely be able to merge the changes to the TypeScript compiler into our fork with the help of Git, conflict free. Greater changes to the language might of course still give us some merge conflicts, however these larger changes could also make our currently used approach break.

8.3 Future Work

8.3.1 Finishing the PTS Compiler

As detailed in section 5.9 on page 59 the implementation of the compiler was not completed, and requires some further work to complete. The majority of the work will presumably lie in performing more advanced semantic analysis in order to correctly identify all references. This could as we have pointed out either be attempted in a fork of the TypeScript compiler, or as a continuation of this compiler. If one is to continue on with this implementation it could be worth looking into if it is possible to use the TypeScript compiler API in order to identify references.

8.3.2 Improve the Compilers Error Messages

As I mentioned shortly in subsection 5.4.1 on page 47 Tree-sitter does have support for giving position of a syntax node, and this could be utilized to produce better error messages. In addition to giving the position of where the error occurred, it would also be helpful to give more informative error messages than we currently do. In our implementation errors are usually first found during the type-check phase, where we might have already instantiated templates, and renamed classes and attributes. We should try to make more effort to spot errors earlier rather than later.

8.3.3 Making Syntax Highlighting for the PTS Language

By utilizing Tree-sitter as our lexer/parser we could presumably pretty easily also utilize it to get syntax highlighting. Most modern editors and IDEs have recently been switching to Tree-sitter for syntax highlighting, opposed to the traditional method of using regex to highlight code files. This should be as simple as writing query files for identifying keywords, operators, etc. Similar to how we extended the TypeScript grammar in order to create our PTS grammar, it should also be possible to extend the TypeScript highlight query files to create syntax highlighting for PTS. In the GitHub repository for the Tree-sitter grammar for PTS there has been done some initial work for setting this up, but has been abandoned as implementing features for the compiler was more urgent.

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] Eyvind W Axelsen and Stein Krogdahl. 'Adaptable generic programming with required type specifications and package templates'. In: *Proceedings of the 11th International Conference on Aspect-oriented Software Development, {AOSD} 2012, Potsdam, Germany, March 25-30, 2012*. Ed. by Robert Hirschfeld et al. ACM, 2012, pp. 83–94. ISBN: 978-1-4503-1092-5. DOI: 10.1145/2162049.2162060. URL: https://doi.org/10.1145/2162049.2162060.

[4] Eyvind W. Axelsen and Stein Krogdahl. 'Groovy package templates: supporting reuse and runtime adaption of class hierarchies'. In: *ACM SIGPLAN Notices* 44.12 (Dec. 2009), pp. 15–26. ISSN: 0362-1340. DOI: 10.1145/1837513.1640139. URL: https://dl.acm.org/doi/10.1145/1837513.1640139.

[5] Eyvind W. Axelsen and Stein Krogdahl. 'Package templates: A definition by semantics-preserving source-to-source transformations to efficient java code'. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE'12*. New York, New York, USA: ACM Press, 2012, pp. 50–59. ISBN: 9781450311298. DOI: 10.1145/2371401.2371409. URL: http://dl.acm.org/citation.cfm?doid=2371401.2371409.

[6] Gilad Bracha. 'The programming language jigsaw: mixins, modularity and multiple inheritance'. In: *Modularity, and Multiple In* March (1992). URL: https://www.researchgate.net/publication/2739177%20http://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:THE+PROGRAMMING+LANGUAGE+JIGSAW%7B%5C#%7D9%7B%5C%%7D5Cnhttp://content.lib.utah.edu/utils/getfile/collection/uspace/id/4356/filename/4228.pdf.

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

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

[9] Paul Hudak. *Domain Specific Languages* *. Tech. rep. Yale University, Department of Computer Science, 1997.

[10] *Inheritance and the prototype chain - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance%7B%5C_%7Dand%7B%5C_%7Dthe%7B%5C_%7Dprototype%7B%5C_%7Dchain (visited on 01/05/2021).

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

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

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.

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

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

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

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

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

[24] Håkon Stordahl. 'BooPT: Implementasjon av Package Templates for Boo'. PhD thesis. University of Oslo, Dec. 2011. URL: https://www.duo.uio.no/handle/10852/9025.

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

[26] *TypeScript: Handbook - Declaration Merging*. URL: https://www.typescriptlang.org/docs/handbook/declaration-merging.html (visited on 28/10/2020).

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