Imperial College London

# `parsley-garnish`

*A linter for the `parsley` parser combinator library*

Rocco Jiang

May 31, 2024

*A dissertation submitted in fulfilment of the requirements for the degree of Master of Engineering*

*in the*

*Department of Computing*
*Imperial College London*

# Statement of Originality

Write about how this is your own work etc etc (check what's needed)

Rocco Jiang

May 31, 2024

# Contents

# List of Figures

# Chapter 1
# Introduction

## 1.1 Motivation

Parser combinators [Hutton 1992] are an elegant approach for writing parsers in a manner that remains close to the original grammar specification. `parsley` [Willis and Wu 2018] is a parser combinator library implemented as an embedded domain-specific language (DSL) [Hudak 1996] in Scala, with an API inspired by the `parsec` [Leijen and Meijer 2001] family of libraries in Haskell. However, as with many libraries, there exists a learning curve to utilising `parsley` and parser combinator libraries in an idiomatic manner.

While well-documented, the wealth of information to get started with `parsley` can be overwhelming for users, particularly those new to parser combinators. Furthermore, there exists a number of design patterns [Willis and Wu 2022] for writing maintainable parsers, which even experienced users may be unaware of. A potential solution to this problem is tooling to provide automated code hints, which a user can use during the development cycle to evaluate if their code adheres to best practices.

A number of modern integrated development environments (IDEs) provide code hints to warn programmers about problems in their source code, highlighting offending snippets and suggesting actions to improve suboptimal or incorrect code [Kurbatova et al. 2021]. Many of these code analysis tools are designed to detect general issues for the host language, rather than specifically for libraries. However, tools may also utilise domain-specific code analyses in order to detect issues specific to a particular system or problem domain [Renggli et al. 2010; Gregor and Schupp 2006].

## 1.2 Project Goals

This project aims to explore the potential of harnessing static code analysis techniques to develop a new tool, `parsley-garnish`, that offers code hints aimed at assisting programmers in writing idiomatic and correct `parsley` code. Additionally, for certain issues that can be automatically fixed, `parsley-garnish` will provide automated actions to resolve the issue. The goal of `parsley-garnish` is to be used as a companion library to `parsley`, in order to improve its ease of adoption and to help users enforce best practices.

# Chapter 2
# Background

## 2.1 Linting and Static Analysis

**Linting** is the process of analysing source code to identify and report issues related to coding style and potential logical errors. The term originates from the `lint` program [Johnson 1978], which examined C source code for bugs, as well as wasteful code patterns that may be legal but error-prone. The tool was also utilised to enforce portability restrictions which aided users in writing portable code that could be compiled on multiple platforms. Since the release of `lint`, many linting tools, known as **linters**, have been developed for a wide range of programming languages.

Nowadays, many linters can be integrated into IDEs, where code analysis performed by the linter is run incrementally in the background. Any violations found by the linter are displayed directly in the editor as warnings or errors at the relevant locations in the source code. This brings early, real-time feedback to the programmer, allowing them to address issues as they write code, with minimal interruption to their development workflow. Linters can also be integrated as part of the code review process, or into continuous integration (CI) pipelines to ensure that code adheres to a set of standards before being merged into the main codebase.

Although the traditional definition for linting is concerned only with *detecting* issues in code, modern linters have broadened their scope significantly. In addition to detecting issues, many linters provide *auto-fix* capabilities to correct issues by automatically rewriting the offending code snippets. This feature is often integrated into IDEs as well: the popular Language Server Protocol for defining IDE features enables these auto-fix features via *code actions* [Gunasinghe and Marcus 2022]. When a section of code is highlighted by a linter warning, a user can apply a code action to automatically fix the issue with a single click.

Linters and related static analysis tools are increasingly becoming more important in modern software development, as modern code continues to become more complex and difficult to reason about. Industry leaders, such as Google [Sadowski et al. 2018] and Meta/Facebook [Calcagno et al. 2015], have embraced these tools as integral components of their software development workflows. The use of automated tools to detect potential issues is not only faster but in some cases more effective than human review, saving developer time and reducing error rates in the development process.

### 2.1.1 Types of Issues Detected by Linters

Many linters are configurable with a set of rules, which specify the categories of issues that the linter should detect. These rules can be enabled or disabled by users, allowing them to customise the linter to their needs. Rules are usually grouped by purpose: some rules are concerned with simply improving code style, while others are concerned with detecting suspicious code patterns indicative of potential bugs.

### Style checks and code quality

Linters can suggest opportunities to improve code by utilising language features in a more idiomatic manner. Snippets of code that violate these stylistic rules are not necessarily incorrect, but should be fixed as they may be harder to read or maintain in the long term. Furthermore, many idiomatic practices exist to avoid common pitfalls that could lead to unintended behaviour. By highlighting good practices, linters can help users avoid these common mistakes that may cause bugs. For example, *ESLint*[1], one of the most popular JavaScript linters, warns against common JavaScript pitfalls such as using the regular equality operator `==` instead of its type-safe alternative `===`.

A well-designed linter can help programmers learn about useful language constructs by suggesting them in the context of their code, aiding them in adhering to best practices and common style conventions. This category of rules is therefore especially helpful as an educational tool for new users of a language, who may be unaware of these idioms. For example, the *Clippy*[2] linter for Rust [Li et al. 2023] categorises a collection of rules as `clippy::complexity` rules to detect code that does something simple in a complex way and suggests a simpler alternative. Fig. 2.1 provides an example of a similar rule in Haskell, from the *HLint*[3] linter. The rule suggests to rewrite the function into an equivalent but more concise form via $\eta$-reduction, presented to the user as a code action that can be applied automatically.

```
foo xs = map (+1) xs
```

(a) A Haskell function `foo`, which can be made more concise using $\eta$-reduction.

```
Eta reduce
Found:
  foo xs = map (+ 1) xs
Why not:
  foo = map (+ 1)
hlint(refact:Eta reduce)
```

(b) The linter warning shown for `foo`.

Fig. 2.1: An example of a warning from the Haskell linter `hlint`, suggesting a fix that a user can choose to automatically apply.

**Domain-specific idioms** A library or especially an embedded DSL may require a particular style of usage that is different from the host language [Hora et al. 2012]. The majority of linters are designed for general-purpose application domains, so they are unlikely to detect issues specific to a more specialised domain. Therefore, linters may be developed for specific libraries or DSLs, with their own set of domain-specific rules. In this case, the accompanying linter can benefit users and improve developer productivity in a similar manner to general-purpose linters: common misuses can be detected and sometimes automatically fixed, and users can be directed to relevant documentation to learn more about correct usage. For instance, the *xUnit.net* testing framework for C# is accompanied by the `xunit.analyzers`[4] package which provides linting rules to enforce best practices specific to *xUnit*.

[1] https://eslint.org/docs/latest/rules/
[2] https://doc.rust-lang.org/clippy/
[3] https://hackage.haskell.org/package/hlint
[4] https://github.com/xunit/xunit.analyzers

#### Code smells and opportunities for refactoring

Code refactoring is a well-established practice in software development. In his influential book *Refactoring: Improving the Design of Existing Code* [Fowler 2018], Fowler defines **refactoring** as "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure". Refactoring may be employed to eliminate **code smells**, which are surface indications that could indicate deeper problems in the system. Code smells are not necessarily problematic on their own, however, they may lead to issues such as bugs or poor maintainability if left unchecked. They are conceptually similar to the stylistic issues mentioned earlier, however they may encompass higher-level structural and design-based problems that are not easily fixed by simple stylistic changes. Examples of code smells include duplicated code, which can be hard to update without introducing bugs, and long methods, which can be difficult to understand and maintain. Therefore, it is often productive to refactor code to eliminate code smells, even if the code is still correct and functional.

Certain linting rules can aid in the refactoring process by broadly identifying code smells and candidate areas for refactoring, suggesting appropriate actions that the user can take. As an example, a linter may detect a fragment of code that is repeated in multiple places: this is a code smell, as discussed previously. The linter may then suggest a code action to automatically apply an *Extract Method* [Fowler 2018] refactoring to avoid code duplication: fig. 2.2 demonstrates how this automatic refactoring process can be performed in the IntelliJ IDEA[5] IDE.

#### Possible bugs or errors

Linters may also directly attempt to detect more serious issues in code, such as possible logic errors. This can be helpful for even experienced users to avoid common pitfalls. For example, *Clippy* has `clippy::suspicious` and `clippy::correctness` rule categories to identify code that is very likely to be incorrect or useless. *ESLint* provides several rules to warn against code patterns that are likely to cause runtime errors, such as re-assigning a `const` variable.

Again, linters may attempt to provide auto-fixes for these issues where possible. However, these issues are usually more complex, which may limit the effectiveness or usefulness of auto-fixes: in the case of a suspicious code pattern, the programmer's intent may not be clear, causing the linter to suggest a fix that does not align with the user's expectations.

### 2.1.2 Implementing Linters

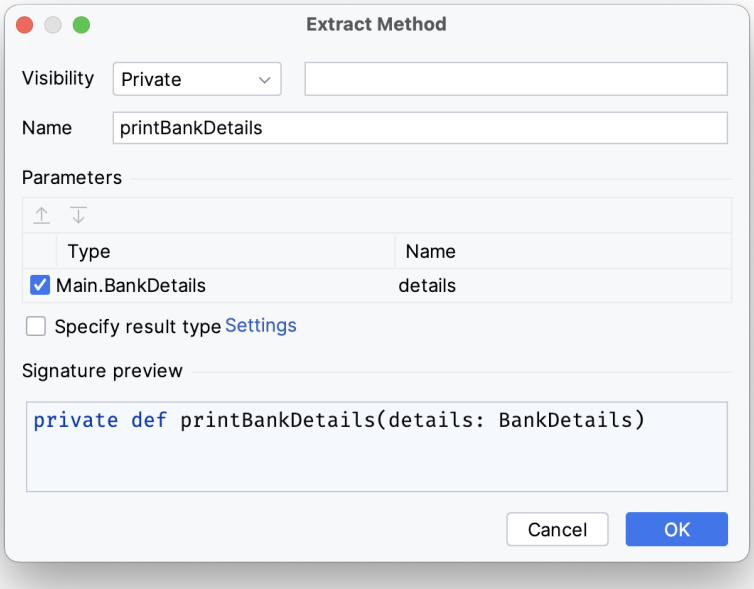
This is the old static analysis section - rework this

The vast majority of linters are implemented using static program analysis, analysing source code to extract information about its behaviour without executing the program itself. This is in contrast to dynamic analysis, which is performed on programs as they are run. Although dynamic analysis can yield more precise results as it observes the actual runtime behaviour of the program, it is also more heavyweight as a result.

[5] https://www.jetbrains.com/idea/

```scala
object Main {
  def main(args: Array[String]): Unit = {
    val bankDetails = getBankDetails()
    println(s"Account name: ${bankDetails.name}")
    println(s"Account balance: ${bankDetails.balance}")
  }
}
```

(a) A snippet of Scala code. A user may wish to extract the highlighted lines into a separate function.




(b) When a user selects the highlighted lines from fig. 2.2a in IntelliJ IDEA, choosing the *Extract Method* refactoring will open this dialogue to preview changes before applying them.

```scala
object Main {
  def main(args: Array[String]): Unit = {
    val bankDetails = getBankDetails()
    printBankDetails(bankDetails)
  }

  private def printBankDetails(details: BankDetails): Unit = {
    println(s"Account name: ${details.name}")
    println(s"Account balance: ${details.balance}")
  }
}
```

(c) The result of applying the *Extract Method* refactoring using the chosen parameters in fig. 2.2b.

Fig. 2.2: An example of the *Extract Method* refactoring in IntelliJ IDEA.

Given their need to be scalable and efficient enough to handle entire codebases, linters are designed to be fast and lightweight. In most practical cases, therefore, dynamic analysis is too slow and resource-intensive to be used for the purpose of linting. However, dynamic languages such as JavaScript present challenges for static analysis due to their dynamic runtime nature. This has led to the exploration of linters that incorporate dynamic analysis techniques to identify issues that static methods cannot detect [Gong et al. 2015]. Despite these developments, state-of-the-art JavaScript linters utilised in industry, such as *ESLint*, still solely rely on static analysis.

* parsley is a scala library * scala is a static language, strongly typed * this section will therefore focus on the traditional static analysis approach, since dynamic analysis is not as relevant here

https://scalacenter.github.io/scalafix/docs/users/related-projects.html Lint on compile vs lint after compile?

**AST stuff**

* how does HLint work? seems like its sometimes prone to false positives

Static analysis tools can reason about how to safely refactor code in an automated manner, performing refactorings as source-to-source transformations. These transformations may be implemented as simple text-based replacements or more robust rewrite rules that operate on the abstract syntax tree (AST) of the source code.

These tools can perform a variety of tasks, ranging from detecting possible bugs [Johnson 1978; Hovemeyer and Pugh 2004] to formal software verification of program properties [Blanchet et al. 2003].

## 2.2 Static Analysis for Scala

DSL linting is hard but luckily Parsley is an eDSL so we can just use Scala metaprogramming utilities

### 2.2.1 Choice of Tooling

The goal of `parsley-garnish` is to provide linting and refactoring capabilities for the `parsley` parser combinator library. Since `parsley` is a Scala library, this project must be implemented using a tool capable of statically analysing Scala code. This section will therefore discuss and evaluate the choices available for implementing `parsley-garnish`.

**Scala compiler plugins** The most powerful approach would be to implement `parsley-garnish` as a compiler plugin [Pickering, Wu, and Németh 2019]. Using the low-level compiler API, it is possible to perform arbitrary code transformations at any step of the compilation process. Compiler plugins therefore offer full freedom to extend the Scala compiler with extra functionality, such as extra passes for code analysis and emitting lint warnings as diagnostics or even compiler errors.

However, this approach has several drawbacks. Firstly, compiler plugins are tightly coupled with the compiler itself, and therefore not portable across major compiler versions. For instance, plugins written for the Scala 3 compiler, known as `dotty`, are completely incompatible with Scala 2 plugins [LAMP/EPFL 2022]. Additionally, developing compiler plugins requires a deep understanding of arcane and poorly documented compiler internals. Exposing the full compiler API permits unsafe operations that may violate undocumented invariants assumed

by the compiler, leading to exceptions during compilation or even malformed bytecode [Sherwany, Zaza, and Nystrom 2015]. The lack of higher-level abstractions also makes it difficult to implement even trivial tasks such as renaming a field.

For these reasons, it would be preferable to explore other tools that may use compiler plugins themselves but provide a higher-level interface for implementing code analysis and transformations.

**Scalameta** *Scalameta*[6] is a metaprogramming framework for Scala that provides a unified interface for performing common metaprogramming tasks. It provides a high-level syntactic API for transforming and pretty-printing Scala source code, as well as a semantic API providing access to semantic information such as type inference and name resolution. make this clearer about the unification of runtime and compile-time metaprogramming? scalameta does more than scala-reflect Scalameta is the successor of the earlier `scala.reflect` metaprogramming framework, which parsed source code into lossy trees that discarded syntactic information such as comments and whitespace [Burmako 2017]. this is called hygiene right? On the other hand, Scalameta trees are lossless and preserve all syntactic details, a key feature that allows code transformations and refactorings to preserve formatting details.

Scalameta's semantic API is powered by *SemanticDB*, a compiler-agnostic data model for semantic information in Scala programs. This allows Scalameta to extract semantic information via compiler plugins that emit data in the SemanticDB format. Thus, Scalameta can work with any compiler that supports SemanticDB, rather than being tied to a specific compiler implementation.

Since Scalameta provides a high-level interface for manipulating syntactic and semantic information, it is a promising choice for this project. Being able to access semantic information is especially helpful for implementing more complex code analyses. However, Scalameta's primary focus is on providing a general metaprogramming framework and therefore lacks API support specifically for implementing linting and refactoring rules. For example, the Scalameta tree transformation utilities do not preserve formatting details when pretty-printed, despite the underlying trees containing this information.

**Scalafix** *Scalafix*[7] is a refactoring and linting tool built on top of Scalameta. It specifically provides an API for implementing fine-grained code transformations that preserve comments and formatting details. Scalafix provides a framework for implementing linting rules to emit warnings, as well as rewrite rules to perform automated code transformations [Geirsson 2017]. Since it is built on Scalameta, a major advantage of Scalafix is that it is also compiler-agnostic and could be integrated into any IDE if a plugin is developed for it.

Originally, Scalafix was designed to help automate the process of migrating code from Scala 2 to 3, which involved many breaking changes to the language [Geirsson 2016]. However, Scalafix has since evolved into a general-purpose tool for implementing code transformations and analyses, utilising the powerful syntactic and semantic APIs provided by Scalameta. Scalafix rules can be either syntactic or semantic, depending on whether they require semantic information to perform their analysis. Syntactic rules are faster to run, since they operate purely on the AST without requiring compilation to extract semantic information, but are more limited in the accuracy of analyses they can perform.

[6] https://scalameta.org/
[7] https://scalacenter.github.io/scalafix/

Scalafix is growing to become the de-facto modern successor to earlier refactoring tools such as Abide[8] and `scala-refactoring`[9]. `scala-refactoring` used `scala.reflect` to implement code transformations, with much extra work utilising the Scala Presentation Compiler AST to preserve formatting details lost by `scala.reflect`. As a result, maintaining the library became difficult and the project was abandoned in favour of a clean implementation using Scalameta, which was designed in part to address the shortcomings of `scala.reflect`.

A drawback of Scalafix is that it is primarily a command-line tool, and therefore by default does not provide a user-friendly interface for interactive usage. However, this can rectified in the future by integrating Scalafix into the Metals LSP server for Scala, which would allow it to be integrated into any IDE that supports the LSP.

Overall, Scalafix emerges as the most favorable choice for implementing `parsley-garnish`. It provides high-level APIs specifically for implementing linting and rewrite rules without necessitating extensive knowledge of compiler internals. Scalafix is also being actively developed and maintained, with good basic documentation and a growing number of examples of usage in the wild.

**Other tools considered** The main alternate contender to Scalafix is the IntelliJ Scala Plugin[10]. However, while the plugin offers superior interactive usage within the IntelliJ IDEA IDE, it is tied to the IntelliJ Scala compiler and therefore not portable across other compilers. To maintain flexibility and not tie `parsley-garnish` to a particular compiler or code editor, Scalafix is a preferable option. Furthermore, documentation is less clear on how to write a Scala plugin for IntelliJ compared to the Scalafix documentation.

WartRemover[11] is a linter implemented as a compiler plugin, with support for writing custom rules. However, it only can emit warnings or errors and does not support auto-fixes, making it less suitable for `parsley-garnish`'s goals.

ScalaStyle[12] is primarily a style checker which also supports custom rules. However, it is only able to perform syntactic analyses and does not have access to semantic information, restricting the types of analyses it can perform.

## 2.3 Parser Combinators

Parsing is the process of extracting structured information from a flat, unstructured representation of the data. Parsers are programs that perform this process, using a specified grammar to determine the structure of the data. They are utilised in a variety of applications such as compilers, interpreters, and processing of data storage formats such as JSON and XML.

Traditionally, parsers have either been written by hand or by using parser generator frameworks such as ANTLR [Parr 2013]. Hand-rolling a parser is a tedious process, requiring the programmer to manually implement the parsing algorithm for the grammar. However, this approach is the most powerful and flexible and can provide excellent performance. Alternatively, parser generators lift the burden of implementing the parsing algorithm, instead requiring the programmer to specify the grammar in the format of a domain-specific language (DSL)

[8] https://contributors.scala-lang.org/t/whats-the-status-of-abide/
[9] https://github.com/scala-ide/scala-refactoring
[10] https://github.com/JetBrains/intellij-scala
[11] https://www.wartremover.org/
[12] http://www.scalastyle.org/

similar to a high-level grammar. The grammar is then compiled by the parser generator tool to produce a parser in a target language. This approach is less flexible but can be more convenient and less error-prone.

Parser combinators [Hutton 1992], which stem from a functional programming background, are a middle ground between the two approaches. They take the form of an embedded DSL written directly in a general-purpose language, rather than the parser generator approach where the DSL is a separate language. With a parser generator, the provided DSL is often limited in its expressiveness. This is not the case with parser combinators, as the full power of the host language is available to the programmer. This approach also reduces boilerplate code: for example, the programmer does not need to convert between the AST produced by the parser generator and their own AST.

A downside of parser combinators, however, is that they are unstandardised compared to parser generators. Across different implementations, parser combinator APIs can vary significantly, making it difficult to transfer knowledge between different libraries. Experienced users of parser combinators may approach a new library with prior knowledge of general concepts but may have misconceptions about the specifics of the API which can lead to confusion and frustration. This is another motivating reason for the development of `parsley-garnish`, to lower the barrier of entry for new users of the `parsley` library.

### 2.3.1 Parsley

TODO: proper, worked example showcasing relevant design patterns and stuff which will be picked up by the linter

`Parsley` [Willis and Wu 2018] is a parser combinator library for Scala that provides an API inspired by the `parsec` [Leijen and Meijer 2001] style of parser combinators. This section provides an illustrative example of a simple expression parser to demonstrate what a parser written in `parsley` looks like.

Consider the EBNF grammar for a simple expression language shown in fig. 2.3a. The parser in fig. 2.4 will parse an expression into the AST represented by the Scala datatype in fig. 2.3b.

Notice how the parser closely resembles the high-level EBNF grammar. The main differences of note include the use of:

- `map` to transform the result of a parser to help construct tree nodes consisting of a single value.
- `zipped` to combine the results of two parsers to help construct tree nodes consisting of multiple values.
- `<~` and `~>` operators to guide the direction of parsers.

Except for the possibly cryptic-looking implementation of `num` to parse a series of digits into an integer, the parser is relatively straightforward to understand.

### 2.3.2 Design Patterns for Parsley

*(This background section is a work-in-progress, and will likely expand to include more information about the specific problems and design patterns I choose to explore in the project.)*

Willis and Wu [Willis and Wu 2022] describe several design patterns for writing maintainable parsers using parser combinators in Scala. They identified common problems and anti-patterns in parser design, and proposed

```
ident ::= "x" | "y" | "z"
num ::= digit+
expr ::= factor "+" expr
factor ::= atom "*" factor
atom ::= ident | num | "(" expr ")"
```

(a) The grammar in EBNF.

```scala
sealed trait Expr
case class Ident(name: String) extends Expr
case class Num(x: Int) extends Expr
case class Add(x: Expr, y: Expr) extends Expr
case class Mul(x: Expr, y: Expr) extends Expr
```

(b) The Scala AST to parse into.

Fig. 2.3: The grammar and AST for our simple expression language.

```scala
val ident = "x" | "y" | "z"
val num: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)

lazy val expr: Parsley[Expr] = (factor, "+" ~> expr).zipped(Add)
lazy val factor: Parsley[Expr] = (atom, "*" ~> factor).zipped(Mul)
lazy val atom: Parsley[Expr]
  = ident.map(Ident) | num.map(Num) | "(" ~> expr <~ ")"
```

Fig. 2.4: A parser for our simple expression language.

solutions in the form of design patterns. This provides a guideline for writing idiomatic `parsley` code for practical parser design, which enables opportunities for the development of linting and refactoring rules.

This thesis hopes to explore how these common problems can be formalised into code smells and suspicious code patterns that can be automatically detected using linting rules. Some of the design patterns are also theoretically amenable to automated refactoring, which we hope to explore and implement in `parsley-garnish`.

Misc. things to write (might move to other sections): * Top-down parsers – intro to left-recursion and the types of left recursion in PEG grammars

Not sure how to order things: lint rules first, or implementation?

# Chapter 3

# Lint Rules in `parsley-garnish`

Catalogue of lint rules implemented.

Categorise these – but also somehow split into the ”simple” rules and the ”complex” rules. Simple rules can consist of a single heading, containing: * Explanation of the rule * Simple example to show a diagnostic, and a before and after if it’s fixable * How it’s implemented in the code * Proof (if applicable) * Limitations

Simple rule ideas: * Overly complex parser definitions * Manually calling implicitSymbol instead of using the implicit

Not sure how to lay out the complex rules yet – so far this is just the left-recursion removal rule. The other complex rule(s) will likely share implementation details with the Parser/Func representation, so work from there.

## 3.1 Avoid Redefining Existing Parsers

* Catch cases when user manually writes out a parser that is already defined in the library

## 3.2 Simplify Complex Parsers

* Apply parser laws, re-using Parser and Func representations to do cool things <- should this be a separate rule?

## 3.3 Ambiguous Implicit Conversions

> Heroin is just one letter away from heroine, and implicit conversions are the heroine we don’t deserve.
>
> Jamie Willis, 2024

### Problem

Implicit conversions are a powerful feature in Scala, allowing users to supply an argument of one type when another is expected, to reduce boilerplate. As noted by Willis and Wu [2022], implicit conversions are particularly useful for designing DSLs. In the context of parser combinators, they introduce the usage of implicit conversions to automatically lift string and character literals into parsers in the *Implicit Conversions* design pattern. This eliminates the need to explicitly wrap these elements in combinators: `string("parsley") | string("garnish")` can now be expressed as just `"parsley" | "garnish"`, more closely resembling the style of a BNF grammar.

The *Implicit Lexer* pattern is a further specialisation of this approach, hiding the boilerplate of whitespace handling entirely within a `lexer` object. This design pattern allows whitespace handling to be encapsulated as private combinators within the `lexer` object, which are then made available only through implicit conversions automatically applied by the Scala compiler.

However, due to their utility, implicit conversions are also an easily abused feature of Scala. They can obscure the flow of the program, making it difficult to understand what the code is doing and potentially hiding side effects or costly operations. A downside particularly relevant to Parsley is that implicit conversions often lead to confusing error diagnostics when the compiler is unable to resolve them.

One common issue arises from ambiguous implicits when there are multiple implicit conversions in scope. Parsley provides `stringLift` and `charLift` combinators in the `parsley.syntax.character` package for the *Implicit Conversions* pattern, and exposes an `implicitSymbol` combinator for lexers to use in the *Implicit Lexer* pattern. For novice users, it is easy to accidentally import both sets of these implicits, when it is likely that they only intended to use the `implicitSymbol` implicit. For example, consider the following code snippet:

```
val p = 'g' ~> "arnish"
p.parse("garnish")
// [error] type mismatch;
//   found   : String("arnish")
//   required: parsley.Parsley[?]
//  Note that implicit conversions are not applicable because they are ambiguous:
//   both method stringLift in object character of type (str: String): parsley.Parsley[String]
//   and method implicitSymbol in class ImplicitSymbol of type (s: String): parsley.Parsley[Unit]
//   are possible conversion functions from String("arnish") to parsley.Parsley[?]
//    val p = 'g' ~> "arnish"
//                   ^^^^^^^^
```

Here, the compiler provides a detailed error message indicating the ambiguity between two possible implicit conversions. However, the compiler is not always able to report such issues clearly. For instance, switching the position of the intended implicit conversion results in a less helpful message:

```
val p = "garnis" <~ 'h'
p.parse("garnish")
// [error] value <~ is not a member of String
//    val p = "garnis" <~ 'h'
//            ^^^^^^^^^^^
```

## Solution

Ideally, this issue would be addressed by implementing a lint-on-compile rule, which could annotate the compiler error message at the exact location of the issue. If this were implemented as a compiler plugin, partial information available from the compiler stages before the error could potentially provide enough detail to identify the exact clashing implicits. This approach would allow leveraging domain knowledge to update the error message with more useful Parsley-specific diagnostics.

Incidentally, WartRemover has a related lint rule for implicit conversions[1], although it only targets the locations where implicit conversions are *defined*, not where they are *applied*. Despite this limitation, it serves as a proof of concept demonstrating the feasibility of such an approach.

Unfortunately, Scalafix restricts usage to only syntactic rules on the bare AST or semantic rules that operate fully post-compilation. Since the ambiguous implicit conversions will cause compilation failures, this lint must be

[1] http://www.wartremover.org/doc/warts.html#implicitconversion

implemented as a syntactic rule. Consequently, the solution takes a different approach: estimating the presence of clashing implicits by examining their import statements within each scope.

## Example

Fig. 3.1 extends the previous example to a full Scala source file following the *Implicit Lexer* pattern, but where the user has erroneously additionally imported the `stringLift` implicit from the *Implicit Conversions* pattern. This results in the Scala compiler throwing an error on line 6 due to ambiguous implicits. When run on this file, `parsley-garnish` will report a warning at line 3 similar to that shown in fig. 3.2.

```
1  object parser {
2    import parsley.syntax.character.stringLift
3    import lexer.implicits._
4
5    val p = "garnis" <~ 'h'
6  }
7
8  object lexer {
9    import parsley.token.Lexer, parsley.token.descriptions.LexicalDesc
10
11   private val lexer = new Lexer(LexicalDesc.plain)
12   val implicits = lexer.lexeme.symbol.implicits
13 }
```

Fig. 3.1: A minimal Parsley program which fails to compile due to ambiguous implicits in the `parser` object.

```
warning: [AmbiguousImplicitConversions] This import may cause clashing implicit conversions:
* import parsley.syntax.character.stringLift at line 2
* import lexer.implicits._ at line 3
If this is the case, you may encounter confusing errors like 'method is not a member of String'.
To fix this, ensure that you only import a single implicit conversion.

  import lexer.implicits._
  ^^^^^^^^^^^^^^^^^^^^^^^^
```

Fig. 3.2: The warning message produced by the `AmbiguousImplicitConversions` lint rule.

## Implementation

Unlike Java, Scala offers more flexibility with import statements, allowing them to appear anywhere in source files rather than just at the top. Scala's import statements are lexically scoped, allowing their visibility to be limited to a single class, object, or function. Additionally, Scala processes import statements in a top-down order within the file, further restricting their visibility, as scopes above an import cannot see the imports defined below them.

* Scalafix (via scalameta) provides a generic traversal of the AST: filter to find all import statements in top-down order * This allows the scope to be lexically managed – traversal in the same order that the compiler reads imports * The ancestor AST node of an import statement is its enclosing scope * Use ancestor information to determine which of the visited imports are in scope at that point

* to find stringLift: Pattern match to find if import is of form ‘import parsley.syntax.character._‘ * to find implicit lexer: pattern match to find if there is an importee called ‘implicitSymbol‘ or if an import contains keywords ‘lexer‘ and ‘implicit(s)‘

* if at any point in the traversal, both types of imports are in scope, report a warning

## 3.4 Remove Explicit Usage of Implicit Conversions

## 3.5 Refactor to use Parser Bridges

* This would be cool, idk if I have time though, but this should also piggyback off of Func * the pos bridges don’t actually exist, so we can ignore that case and just say its too much code synthesis * shouldn’t be too bad? idk * indicate limitations that this will only work if the ADT is defined in the same file, in order to extend it

## 3.6 Left Recursion Factoring

# Chapter 4

# Implementation

Non-terminal detection. This may get reworked/renamed since it's pretty specialised for leftrec rn, and in reality it's just trying to grab all the parsers.

Other util things? ACTUALLY NEED TO DO: import combinators if they aren't already imported

## 4.1 Parser Representation

Several of the more complex lint rules, most notably §3.6, require manipulating parser combinators in a high-level manner. It would be possible, but extremely cumbersome, to work directly with the generic Scala AST nodes provided by Scalameta. Instead, it is very useful to represent parsers as an algebraic data type ADT to simplify the manipulation of combinators. This section explores the motivation behind this and the design choices made in the implementation. Use the §3.6 rule as a basis/context to demonstrate the utility of this representation.

Why do we want a parser representation? Because the left-recursion factoring is based on lawful transformations and rearrangements of parsers, which are much easier to achieve and reason about when parsers are represented as a high-level ADT rather than as raw Scala AST nodes. This representation also then gives us for free the implementation for lint rules such as *Simplify Complex Parsers* rule, which applies parser laws to simplify parser definitions.

Running example to motivate all requirements for the parser representation – removing left recursion from the following simple parser:

```
lazy val p: Parsley[String] = (p, string("a")).zipped(_ + _) | string("b")
```

### 4.1.1 Detecting Parsers

Finding AST nodes of interest which actually represent a Parser. Find val/var/def definitions which have a type that is inferred to have type `Parsley[_]`.

Take our example:

```
lazy val p: Parsley[String] = (p, string("a")).zipped(_ + _) | string("b")
^^^^^^^^ ^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
val      sym p: Parsley[_]    rhs of definition: convert this to Parser
```

Convert rhs of definition to Parser. In each document, build a map of all parsers defined in that document, indexed by the symbol of the definition. This gets us all the non-terminals in the grammar defined within that file.

### 4.1.2 Converting Scalameta Trees to the Parser ADT

Pattern match on AST nodes of interest to build the appropriate Parser. Each Parser defines a partial function `fromTerm` which creates an instance of that parser from the appropriate Scalameta `Term`. Use Scalafix's `SymbolMatcher` to match tree nodes that resolve to a specific set of symbols. This makes use of semantic information from SemanticDB, so we are sure that a `<*>` is actually within the `parsley.Parsley` package, rather than

some other function with the same name. This is a drawback of HLint, which is prone to false positives due to its reliance on syntactic information only.

For example:

```
object Ap {
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

  def fromTerm(implicit doc: SemanticDocument): PartialFunction[Term, Ap] = {
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
      Ap(p.toParser, q.toParser)
  }
}
```

So this looks for an infix application `p <*> q`, where we can then build an `Ap` parser while recursively converting `p` and `q` to parsers.

Our running example would then be converted as so (cleaned up to be less verbose). For now we are only concerned with the parser representation; we will touch upon how `Function`s are represented in the next section.

```
// (p, string("a")).zipped(_ + _) | string("b")
Choice(
  Zipped(Function(_ + _), List(NonTerminal(p), Str(a))),
  Str(b)
)
```

### 4.1.3 Building New Parsers From Existing Parsers

Now that we have parsers represented as an ADT, we can easily build new parsers from existing parsers. This is crucial for the left-recursion factoring rule, which "unfolds" parsers into separate parsers representing the left-recursive and non-left-recursive parts. These are then recombined to form parsers which are free from left recursion.

Make this even easier by utilising Scala's ability to define infix operators, define them as extension methods on the `Parser` trait. For example:

```
implicit class ParserOps(private val p: Parser) extends AnyVal {
  def <*>(q: Parser): Parser = Ap(p, q)
  def <|>(q: Parser): Parser = Choice(p, q)
  def map(f: Function): Parser = FMap(p, f)
}
```

This makes it more ergonomic to manipulate parsers, it's like we're writing Parsley code itself. A small example snippet from the `unfold` method on the `Ap` parser:

```
val lefts = {
  val llr = pl.map(flip) <*> q
  val rlr = pe.map(f => ql.map(composeH(f))).getOrElse(Empty)
  llr <|> rlr
}
```

Notice how the code closely resembles the high-level description of the transformation, using `<*>`, `<|>`, `map`, operators.

### 4.1.4 Simplifying Parsers Using Parser Laws

Once all the unfolded parsers have been recombined, the raw output is very noisy and difficult to read. Again, ignore the functions, these will be covered in §4.2.

```
lazy val p: Parsley[String] = chain.postfix(
  empty | (empty.map(a => b => a + b) | empty <*> p) <*> string("a") | string("b") | empty
)(
  (empty.map(FLIP) <*> p | pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
    | empty | empty
)
```

For human readability of the transformed output, it is therefore important to simplify the parser as much as possible. This is another key motivation for the parser representation, bringing static inspectability. Similar to the high-level optimisations done in [Willis, Wu, and Pickering 2020] using parser laws based on applicative, alternative, selective. Willis [2023] provides the most comprehensive list of parser laws which Parsley adheres to.

```
p.map(f).map(g) = p.map(g compose f)                    (4.1)
  pure(f) <*> pure(x) = pure(f(x))                      (4.2)
      pure(f) <*> x = x.map(f)                          (4.3)
           empty | u = u                                (4.4)
           u | empty = u                                (4.5)
        pure(x) | u = pure(x)                           (4.6)
         empty <*> u = empty                            (4.7)
         empty.map(f) = empty                           (4.8)
```

Fig. 4.1: Functor (4.1), Applicative (4.2, 4.3), and Alternative (4.4–4.8) laws.

Most of the laws in fig. 4.1 have already been shown to hold for Parsley in [Willis and Wu 2018]. An additional proof for eq. (4.8) can be found in appendix A.

Example by hand. In the example we can see that the most noise results from the `empty` combinators. Tidying those up first:

```
lazy val p: Parsley[String] = chain.postfix(
  empty | empty | string("b") | empty
)(
  (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
    | empty | empty
)
```

`empty | empty | string("b") | empty` simplifies to just `string("b")` via eqs. (4.4) and (4.5).

```
    (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
 =      { eqs. (4.2) and (4.3) }
    pure(COMPOSE(a => b => a + b)(ID)).map(FLIP) <*> string("a")
 =      { eqs. (4.2) and (4.3) }
    pure(FLIP(COMPOSE(a => b => a + b)(ID))) <*> string("a")
 =      { eq. (4.3) }
    string("a").map(FLIP(COMPOSE(a => b => a + b)(ID)))
```

So final result:

```
val f: Function = FLIP(COMPOSE(a => b => a + b)(ID))
lazy val p: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

#### Implementation

Bottom-up traversal to apply peephole optimisations based on parser laws. For maintainability, code readability, and extensibility: decouple the application of transformation function from recursive traversal through the datatype. Hand-written traversal, but inspired by the generic traversal techniques in [Mitchell and Runciman 2007]. Make ADT sealed so the compiler will warn if we miss on adding a new case: this is still more error prone than generically deriving, but we don't want to introduce a new dependency on Shapeless library (it's not complex enough to warrant bringing in a dependency for generic derivation).

Representation of Parsley combinators in parsley-garnish. Compare with approach in Scala Parsley, take cues from the 2018 paper. * Approach to composites? Need to think about this. * For LeftRec: Parse ASTs into a small group of core combinators, but we also need to represent composite combinators as their own case classes – recombine/"simplify" after analysis is concluded, it doesn't really matter if we completely change what combinators are used as long as semantic meaning is preserved. * For others: probably need to parse directly into composite combinators, since we don't want to destructively modify what combinators have been used. * Optimisations: for us, the goal is human readability, so this is interesting to compare to the paper. Lots of similar stuff actually, like top-down peephole optimisations utilising parser laws (I think I do it this way? Need to double check). * For cleanliness to isolate boilerplate: https://blog.sumtypeofway.com/posts/introduction-to-recursion-schemes.html – we don't have a generic traversal, but we can decouple the recursive application of a given partial function from the actual pf itself (I've called it .transform for the Parser class)

## 4.2 Function Representation

From the running example it is hopefully becoming clear that functions also need to be simplified, for the same reasons that parsers need to be simplified.

Abstraction built over scalafix/meta ASTs to represent functions. Allows us to statically evaluate function composition/flipping etc, so it doesn't turn into one big mess – again, human readability of the transformed output is the goal. Abstraction is again an ADT as a lambda calculus, but with parameter lists so not everything is curried. îdk, this is still a work-in-progress. Seems that there might not be enough time to uncurry the leftrec analysis so this design decision might not be super important. Representation as a lambda calc has allocation overhead, but greatly simplifies function evaluation via beta reduction, instead of having to deal with high-level representations of compose/id (not too bad tbh) and flip (annoying). Also attempted to make it typed but that didn't go so well with Scala's limitations on type inference.

* Extracting method arguments (alongside their types) is very painful * Need to unify information from signature (within symbolinformation) and synthetics * synthetics exist in certain cases: .apply methods, showing the concrete type of a generic argument, implicit conversions * from https://scalacenter.github.io/scalafix/docs/developers/semantic-tree.html: SemanticTree is a sealed data structure that encodes tree nodes that are generated by the compiler

from inferred type parameters, implicit arguments, implicit conversions, inferred .apply and for-comprehensions.

* map, lift (implicit and explicit), zipped, (.as perhaps?) – these should surely boil down into two cases: (x, y).xxx(f) and xxx(f, x, y) * named function literals (val) * named method literals (def) * anonymous functions i.e. lambdas * functions with placeholder syntax * apply methods of case classes - symbol will tell its a class signature so we use this as a clue to look at synthetics??? * generic bridges – I reckon the information will probably show up in synthetics again

* Don't have full access to type information - can do more work here theoretically, but its difficult and error-prone * So we don't model a typed lambda calculus, just have it untyped

Approaches - AVOIDING capture via substitution * Substitution approaches * De Bruijn indices - inefficient to open/close terms so much - De Bruijn levels as an alternative * HOAS * Normalisation by evaluation

# Chapter 5
# Project Plan

## 5.1 Proposed deliverables and milestones

This project is rather exploratory in nature, and as such, these milestones are highly subject to change as the project progresses. However, I have attempted to outline some key deliverables which I believe to be achievable, as well as some optional deliverables to explore if time permits.

### 5.1.1 Key deliverables

- Simple lint rules for non-idiomatic combinator usage which can be simplified with an automated refactoring. For example, `endBy(p, sep)` is the idiomatic equivalent to `many(p <* sep)`.
- Detection of left-recursive parsers, which are a bug pattern since they will cause infinite recursion at runtime. It should be possible to perform a static analysis to produce a code transformation to utilise the `postfix` family of combinators to eliminate left-recursion and resolve the bug.
- Lint rules to detect ambiguity in parsers, which can be a bug pattern since the parser may not behave as expected. It may be possible to provide an automated fix in some cases, but more research has to be done to determine the feasibility of this.

  Conversely, it may also be possible to detect overuse of the `atomic` combinator. Users may overuse this combinator to avoid ambiguity, but overuse is a code smell as it can negatively impact error messages reported by the parser.
- Proofs that all code transformations preserve the semantics of the original parser.

### 5.1.2 Optional deliverables

- Lint rules that warn against side-effecting parsers. `Parsley` employs aggressive parser optimisations which may assume that parsers are pure. The lint rule would remind users to use the `impure` combinator to avoid `parsley` from optimising away any intended behaviour.
- Lint rules to detect when a parser using `chain/infix/postfix` combinators can be rewritten to use a precedence table, and the automated refactoring to accompany this rule. Further lints/refactorings related to precedence tables may also be possible, such as looking at type information to determine the most appropriate shape (`Ops/SOps/GOps`).
- An editor plugin or extension to provide integration of `parsley-garnish` into the code editor. As explored in the background section, it is possible to provide this integration for Scalafix, but the difficulty of this task is unknown.
- Exploration of possible pedagogical applications of `parsley-garnish`. For example, lint rules may be useful for marking submissions for the Second Year WACC compilers project.

## 5.2 Current Progress

Most of the work so far has been on research and experimentation with Scalafix. Although there was good documentation to get started, Scalafix is an active project and the documentation is not always fully comprehensive or up-to-date. I still have some concerns about whether the Scalafix API is expressive enough to implement all of the analyses I have in mind, but I will continue to explore this as the project progresses.

From my experimentation, I have a relatively good idea of how to implement most of the "simple" lint rules.

At the time of writing, I have also implemented a proof-of-concept rule for transforming left-recursive parsers, on a small subset of `parsley`'s combinators. This has been encouraging but much work remains to be done to be able to apply the rule on more complex parsers. The current results of the transformations are also sometimes too complex, so I will need to investigate methods such as defunctionalisation and/or alternate methods of representing terms to simplify the output statically. Furthermore, the transformation is likely to be correct but not proven yet.

## 5.3 Project Timeline

My immediate plan is to continue work on the left-recursion transformation, while also working on the simpler lint rules. I hope to complete the left-recursion rule by early March, although unforseen difficulties may arise (such as if output simplification proves to be more difficult than expected). If it is not complete or close to complete by then, I will re-evaluate the feasibility of the rest of the milestones and adjust accordingly. I have already communicated to my supervisor that this left-recursion rule will likely be the most involved deliverable of the project, although there is also a chance that many of the techniques employed could be reused for other transformations.

The next two months will be spent exploring other milestones. I plan on approaching this by alternating research and implementation, as I have found that this has been a good way to make progress so far, while also guaranteeing that I will make progress on the final deliverable.

By the beginning of summer term, I would also like to have a very rough draft of the final report completed, based on the work completed so far. This will allow me to communicate with my supervisor on the state of the project, as well as leaving me ample time to edit and refine the report. The remaining milestones to cover will be decided during this meeting; at this point, having implemented a number of rules, I hope to have a much better grasp on how long each deliverable could take. This remaining time will then be divided between writing the report and implementing the agreed-upon milestones.

## Chapter 6

# Evaluation Plan

A large corpus of student WACC parsers is available to evaluate the real-world applicability of `parsley-garnish`. This can be utilised to assess the tool's ability to achieve quantitative and qualitative metrics.

The large number of parsers available to test can help evaluate the correctness and accuracy of each lint rule. We might find that certain rules are more prone to false positives than others, or that some automated refactorings may introduce new issues such as compilation errors. These quantitative metrics can be counted and used to evaluate rules against each other, as well as the overall effectiveness of the tool.

We can also estimate the proportion of issues that `parsley-garnish` is able to detect automatically. This can be done by comparing the number of issues detected automatically against the number of issues found manually by a code reviewer.

Qualitative metrics such as the quality of automated fixes can also be assessed by manually reviewing the output of the tool. For example, an issue I pointed out in the project plan is that the output of the left-recursion transformation may be in a form that is "ugly" when it could be simplified to a much more human-readable form. The quality of fixes would affect the practicality of the `parsley-garnish`, since users may be reluctant to apply automated fixes if the output quality can be poor.

Furthermore, the performance of `parsley-garnish` can be evaluated in terms of execution time and resource consumption. There are no direct competitors to benchmark against, but we can still evaluate the tool's performance against a baseline of acceptable performance, or perhaps a collection of other Scala static analysis tools deemed to be of similar complexity. I am thinking of benchmarking the tool during development as well, since it may affect design decisions such as implementing some rules as purely syntactic rather than semantic rules. I am not sure how much resource overhead a semantic rule would incur, so this could be insignificant or a major tradeoff (between speed, correctness, and accuracy) to consider.

Finally, it is possible to gather feedback from students interested in using `parsley-garnish` to improve their WACC parsers. User surveys and interviews would provide insights into the tool's usability, helpfulness, and areas for improvement. Although the timing of the WACC Lab does not align with the project timeline, I am hoping that some students may still be interested in trying out the tool on their final WACC submissions or possibly personal projects using `parsley`.

# Bibliography

Blanchet, Bruno et al. (June 2003). "A static analyzer for large safety-critical software". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. San Diego, California, USA: Association for Computing Machinery, pp. 196–207. ISBN: 1581136625. DOI: 10.1145/781131.781153. URL: https://doi.org/10.1145/781131.781153.

Burmako, Eugene (Mar. 2017). "Unification of Compile-Time and Runtime Metaprogramming in Scala". Ph.D. Thesis. Lausanne: EPFL. DOI: 10.5075/epfl-thesis-7159. URL: https://doi.org/10.5075/epfl-thesis-7159.

Calcagno, Cristiano et al. (2015). "Moving Fast with Software Verification". In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. Cham: Springer International Publishing, pp. 3–11. ISBN: 978-3-319-17524-9. DOI: 10.1007/978-3-319-17524-9_1. URL: https://doi.org/10.1007/978-3-319-17524-9_1.

Fowler, Martin (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. ISBN: 9780134757681.

Geirsson, Ólafur (Oct. 2016). *Introducing Scalafix: a code migration tool for Scala*. URL: https://scala-lang.org/blog/2016/10/24/scalafix.html (visited on 01/25/2024).

— (Nov. 2017). *Catch bugs with Scalafix v0.5*. URL: https://www.scala-lang.org/blog/2017/09/11/scalafix-v0.5.html (visited on 01/25/2024).

Gong, Liang et al. (July 2015). "DLint: dynamically checking bad coding practices in JavaScript". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, pp. 94–105. ISBN: 9781450336208. DOI: 10.1145/2771783.2771809. URL: https://doi.org/10.1145/2771783.2771809.

Gregor, Douglas and Sibylle Schupp (2006). "STLlint: lifting static checking from languages to libraries". In: *Software: Practice and Experience* 36.3, pp. 225–254. DOI: 10.1002/spe.683. URL: https://doi.org/10.1002/spe.683.

Gunasinghe, Nadeeshaan and Nipuna Marcus (2022). *Language Server Protocol and Implementation: Supporting Language-Smart Editing and Programming Tools*. en. Berkeley, CA: Apress. ISBN: 978-1-4842-7791-1. DOI: 10.1007/978-1-4842-7792-8. URL: https://doi.org/10.1007/978-1-4842-7792-8 (visited on 01/25/2024).

Hora, André et al. (2012). "Domain specific warnings: Are they any better?" In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 441–450. DOI: 10.1109/ICSM.2012.6405305. URL: https://doi.org/10.1109/ICSM.2012.6405305.

Hovemeyer, David and William Pugh (Dec. 2004). "Finding bugs is easy". In: *SIGPLAN Not.* 39.12, pp. 92–106. ISSN: 0362-1340. DOI: 10.1145/1052883.1052895. URL: https://doi.org/10.1145/1052883.1052895.

Hudak, Paul (Dec. 1996). "Building domain-specific embedded languages". In: *ACM Comput. Surv.* 28.4es, 196–es. ISSN: 0360-0300. DOI: 10.1145/242224.242477. URL: https://doi.org/10.1145/242224.242477.

Hutton, Graham (July 1992). "Higher-order functions for parsing". In: *Journal of Functional Programming* 2.3, pp. 323–343. ISSN: 1469-7653. DOI: 10.1017/s0956796800000411. URL: https://doi.org/10.1017/S0956796800000411.

Johnson, Stephen C (1978). *Lint, a C program checker*. Technical Report 65. AT&T Bell Laboratories.

Kurbatova, Zarina et al. (2021). "The IntelliJ Platform: A Framework for Building Plugins and Mining Software Data". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pp. 14–17. DOI: 10.1109/asew52652.2021.00016. URL: https://doi.org/10.1109/asew52652.2021.00016.

LAMP/EPFL (Nov. 2022). *Changes in Compiler Plugins*. URL: https://docs.scala-lang.org/scala3/reference/changed-features/compiler-plugins.html (visited on 01/25/2024).

Leijen, Daan and Erik Meijer (July 2001). *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-27. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007. URL: https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/.

Li, Chunmiao et al. (Oct. 2023). *Unleashing the Power of Clippy in Real-World Rust Projects*. DOI: 10.48550/arXiv.2310.11738. URL: https://doi.org/10.48550/arXiv.2310.11738 (visited on 01/25/2024).

Mitchell, Neil and Colin Runciman (2007). "Uniform boilerplate and list processing". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Freiburg, Germany: Association for Computing Machinery, pp. 49–60. ISBN: 9781595936745. DOI: 10.1145/1291201.1291208. URL: https://doi.org/10.1145/1291201.1291208.

Parr, Terence (Jan. 2013). *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf. ISBN: 1934356999.

Pickering, Matthew, Nicolas Wu, and Boldizsár Németh (2019). "Working with source plugins". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, pp. 85–97. ISBN: 9781450368131. DOI: 10.1145/3331545.3342599. URL: https://doi.org/10.1145/3331545.3342599.

Renggli, Lukas et al. (2010). "Domain-Specific Program Checking". en. In: *TOOLS'10: Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*. Ed. by Jan Vitek. Vol. 6141. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 213–232. ISBN: 978-3-642-13953-6. DOI: 10.1007/978-3-642-13953-6_12. URL: https://doi.org/10.1007/978-3-642-13953-6_12.

Sadowski, Caitlin et al. (Mar. 2018). "Lessons from building static analysis tools at Google". In: *Commun. ACM* 61.4, pp. 58–66. ISSN: 0001-0782. DOI: 10.1145/3188720. URL: https://doi.org/10.1145/3188720.

Sherwany, Amanj, Nosheen Zaza, and Nathaniel Nystrom (2015). "A Refactoring Library for Scala Compiler Extensions". In: *Compiler Construction*. Ed. by Björn Franke. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 31–48. ISBN: 978-3-662-46663-6. DOI: 10.1007/978-3-662-46663-6_2. URL: https://doi.org/10.1007/978-3-662-46663-6_2.

Willis, Jamie (Aug. 2023). "Parsley: Optimising and Improving Parser Combinators". PhD Thesis. London: Imperial College London. DOI: 10.25560/110313. URL: https://doi.org/10.25560/110313.

Willis, Jamie and Nicolas Wu (Sept. 2018). "Garnishing parsec with parsley". In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. Scala 2018. New York, NY, USA: Association for Computing Machinery, pp. 24–34. ISBN: 978-1-4503-5836-1. DOI: 10.1145/3241653.3241656. URL: https://doi.org/10.1145/3241653.3241656.

— (2022). "Design patterns for parser combinators in Scala". In: *Proceedings of the Scala Symposium*. Scala '22. Berlin, Germany: Association for Computing Machinery, pp. 9–21. ISBN: 9781450394635. DOI: 10.1145/3550198.3550427. URL: https://doi.org/10.1145/3550198.3550427.

Willis, Jamie, Nicolas Wu, and Matthew Pickering (Aug. 2020). "Staged selective parser combinators". In: *Proc. ACM Program. Lang.* 4.ICFP. DOI: 10.1145/3409002. URL: https://doi.org/10.1145/3409002.

## Appendix A

# Proofs of Parser Laws

The approach taken to prove the following parser laws for `parsley` is via equational reasoning on `gigaparsec` semantics, under the assumption that their semantics are equivalent. While there is no formal proof of this equivalence at the present, `gigaparsec` was designed to have semantics equivalent to `parsley`'s.

## A.1 Left absorption for `fmap`

```
  f <$> empty
=   { applicative functor law }
  pure f <*> empty
=   { definition of <*> }
  liftA2 ($) (pure f) empty
=   { semantics of liftA2 }
  Parsec $ λst ok err →
    let ok' x st' = (unParsec empty) st' (ok . (x $)) err
    in (unParsec $ pure f) st ok' err
=   { semantics of empty }
  Parsec $ λst ok err →
    let ok' x st' = (unParsec $ raise (`emptyErr` 0)) st' (ok . (x $)) err
    in (unParsec $ pure f) st ok' err
=   { semantics of raise }
  Parsec $ λst ok err →
    let ok' x st' = (unParsec $ Parsec $ λst'' _ bad →
      useHints bad (emptyErr st'' 0) st') st' (ok . (x $)) err
    in (unParsec $ pure f) st ok' err
=   { β-reduction }
  Parsec $ λst ok err →
    let ok' x st' = useHints err (emptyErr st' 0) st'
    in (unParsec $ pure f) st ok' err
=   { semantics of pure }
  Parsec $ λst ok err →
    let ok' x st' = useHints err (emptyErr st' 0) st'
    in (unParsec $ Parsec $ λst'' ok'' _ → ok'' f st'') st ok' err
=   { β-reduction }
  Parsec $ λst ok err →
    let ok' x st' = useHints err (emptyErr st' 0) st'
```

```
    in ok' f st
=   { inline ok' }
  Parsec $ λst ok err → useHints err (emptyErr st 0) st
=   { rearrange and α-conversion }
  Parsec $ λst _ bad → useHints bad ((`emptyErr` 0) st) st
=   { fold definition of raise }
  raise (`emptyErr` 0)
=   { fold definition of empty }
  empty
```