Imperial College London

parsley-garnish

A linter for the `parsley` parser combinator library

Rocco Jiang

June 17, 2024

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

in the

Department of Computing
Imperial College London

Abstract

TODO

Acknowledgements

I would like to thank my supervisor, Jamie Willis, for his support throughout not only this project but also my time at Imperial. Thank you for always being available no matter what time of day, and for introducing me to Scala and parser combinators in the first place.

I would also like to thank my friends for making life interesting, and especially thank those who had to read my drafts and listen to my nonsensical ramblings about this project.

Finally, I'd like to thank my family for their unwavering and unconditional support throughout my life.

Contents

Chapter 1
Introduction

Parser combinators [Hutton 1992] are an elegant approach for writing parsers in a manner that remains close to their 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. Although `parsley` itself has a user-friendly API, parser combinators in general have pitfalls that may be unexpected for new users. Even experienced users can unintentionally write unidiomatic parsers: `parsley` has first-class support for a number of design patterns [Willis and Wu 2022] for writing maintainable parsers, which users migrating from other parser combinator libraries may not be aware of.

This project aims to address these issues by developing a companion *linting* tool for `parsley`, called `parsley-garnish`, that provides automated code hints and fixes to assist users in writing idiomatic and correct parsers. 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 linters 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]. Well-designed linters can offer significant benefits to users:

- Linters can be particularly valuable for uncovering subtle issues that might be hard to diagnose and locate, especially in large codebases. Automated fixes can save further effort by resolving issues without manual intervention.
- Linters are also beneficial for teaching best practices in context, offering relevant hints and improvements precisely where sub-optimal code is detected.

For example, suppose a user wants to write a simple arithmetic expression parser in `parsley`, which evaluates the parsed expression as a floating-point calculation. The parser will be based on the following EBNF grammar, with standard arithmetic operator precedence and left-associativity:

$$
\begin{aligned}
\langle digit \rangle &::= \text{`0'} \ldots \text{`9'} \\
\langle number \rangle &::= \langle digit \rangle + \\
\langle expr \rangle &::= \langle expr \rangle \text{ `+' } \langle term \rangle \mid \langle expr \rangle \text{ `-' } \langle term \rangle \mid \langle term \rangle \\
\langle term \rangle &::= \langle term \rangle \text{ `*' } \langle atom \rangle \mid \langle term \rangle \text{ `/' } \langle atom \rangle \mid \langle atom \rangle \\
\langle atom \rangle &::= \text{`(' } \langle expr \rangle \text{ `)'} \mid \langle number \rangle
\end{aligned}
$$

By closely following the structure of the grammar, a naïve first attempt at writing the parser-evaluator in `parsley` may resemble the following:

```
val number: Parsley[Float] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit).map(_.toFloat)

lazy val expr: Parsley[Float] = (expr, char('+') ~> term).zipped(_ + _)
                              | (expr, char('-') ~> term).zipped(_ - _)
                              | term
lazy val term: Parsley[Float] = (term, char('*') ~> atom).zipped(_ * _)
                              | (term, char('/') ~> atom).zipped(_ / _)
                              | atom
lazy val atom: Parsley[Float] = char('(') ~> expr <~ char(')') | number
```

Although this parser compiles and looks correct at first glance, it suffers from a puzzling runtime behaviour. Attempting to run the parser by executing `expr.parse("1+2*3/4")` results in a cryptic `StackOverflowError`

when compiled in Scala 2. Running the same code on Scala 3 shines some light on the issue: the compiler reports an "infinite loop in function body" for `expr` and `term`. The user has stumbled upon the age-old problem of *left-recursion*, which the grammar uses to encode the left-associative behaviour of the arithmetic operators. Left-recursive grammars are problematic for recursive-descent parsers, which are the class of parsers that `parsley` produces. A recursive-descent parser would see that the first thing to parse when attempting to parse `expr` is `expr` itself, leading to an infinite cycle.

The caveat of left-recursion may not be immediately obvious to a novice user, and even less obvious is how to resolve the issue in an idiomatic manner. This situation is exactly where a domain-specific linter like `parsley-garnish` can be invaluable. A linter with knowledge of the `parsley` library could help users by providing *relevant* suggestions at the *precise* location of the issue:

```
lazy val expr: Parsley[Float] = (expr, char('+') ~> term).zipped(_ + _)
//       ^^^^^^^^^^^^^^^^^^^
// Warning: This parser is left-recursive, which will cause an infinite loop when parsing.
// Suggestion: Refactor using chain combinators from the parsley.expr module,
//             or with a precedence table from the parsley.expr.precedence module.
```

In addition to the correctness issue, the left-recursive parser example also suffers from a stylistic problem: overuse of the `char` combinator leads to visual clutter, making the parser harder to read. In `parsley`, this can be addressed by using implicit conversions to lift character literals directly to parsers – this feature may not be known to users new to the library. Thus, a linter could also aid users in learning about `parsley` idioms and best practices:

```
lazy val atom: Parsley[Float] = char('(') ~> expr <~ char(')') | number
//                              ^^^^^^^^^            ^^^^^^^^^
// Info: Explicit usage of the 'char' combinator may not be necessary.
// Suggestion [auto-fix available]: Use implicit conversions:
// import parsley.syntax.character.charLift
// ──────┬──────────────────────────────────────
//       └── Bring implicit conversion into scope
//
// lazy val atom: Parsley[Float] = '(' ~> expr <~ ')' | number
//                                  ┬┬            ┬┬
//       Remove char combinators ───┴┴────────────┴┴
```

The aim of `parsley-garnish` is to provide relevant linting rules, like the above, to guide users towards writing improved `parsley` code. With useful hints and automatic fixes, our hypothetical user can be steered towards a correct and idiomatic parser:

```
lazy val expr: Parsley[Float] = precedence('(' ~> expr <~ ')', number)(
  Ops(InfixL)('*' as (_ * _), '/' as (_ / _)),
  Ops(InfixL)('+' as (_ + _), '-' as (_ - _))
)
```

With this definition of `expr`, the parser is able to parse left-associative expressions without left-recursion, and the syntactic noise of the `char` combinators has been removed. Executing `expr.parse("1+2*3/4")` now correctly evaluates to `2.5`!

Contributions

The contributions of this project are summarised as follows:

- TODO

Outline

Chapter 2 begins this report by outlining the key background material required to understand the project. This includes an introduction to linters and how they are typically implemented; a discussion on writing linters specifically for Scala; and an overview of parser combinators, their design patterns, and the `parsley` library.

From there, Chapter 3 dives into the first set of linting rules that `parsley-garnish` implements. These relatively simple rules focus on enforcing idiomatic design patterns relating to *implicit conversions* in Scala, and were largely inspired by common issues I've seen in my experiences as a teaching assistant for the second-year undergraduate WACC compilers project at Imperial.

Chapter 4 tackles a more ambitious rule to automatically refactor left-recursive parsers into a form that `parsley` can handle. This chapter introduces the idea of utilising intermediate AST representations to perform higher-level domain-specific transformations. It shows that a basic implementation is not sufficient to handle the complexities of the problem, and motivates the need for a more sophisticated approach. Chapter 5 therefore further develops and refines these ideas, drawing inspiration from metaprogramming techniques and `parsley`'s own optimisation machinery.

These improvements allow the left-recursion transformation to be revisited and completed in Chapter 6, which also explores more advanced rules that are unlocked by the new AST. Finally, Chapter 7 evaluates TODO. The remainder of the report concludes with discussion of related work and future directions for the project.

Chapter 2
Background

This chapter covers the basic knowledge required to understand this project's two main areas of focus:

- **Linters** and the types of issues that they detect (§2.1), and how the `parsley-garnish` linter can be implemented as a Scala metaprogram that manipulates abstract syntax trees (§2.2).
- **Parser combinators**, their design patterns, and the `parsley` library (§2.3).

Basic familiarity with Scala is assumed, although the code examples will be comprehensible for those experienced with a JVM language like Java or Kotlin. Some experience with functional programming concepts is also beneficial, but not essential.

2.1 Linters

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.

2.1.1 Categories of Lint Rules

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.

2.1.1.1 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 `===`.

[1] https://eslint.org/docs/latest/rules/

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 η-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 η-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*.

2.1.1.2 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

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

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.

2.1.1.3 Likely bugs or errors

In some cases, linters may be able to detect with relatively high confidence that a piece of code is likely to contain a bug. Rules that detect more egregious issues like this are helpful for even experienced users to avoid common pitfalls. For example, *Clippy* has `clippy::correctness` rules which identify code very likely to be incorrect or useless, and `clippy::suspicious` rules for code that is *very sus* but could be a false positive. *ESLint* also 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 code 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

2.1.2.1 Static vs Dynamic Analysis

Static analysis analyses 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 run to observe their actual runtime behaviour. Both techniques have their own advantages and disadvantages, and the choice between them depends on the desired properties of a tool. Dynamic analysers are *precise* in that they only observe what a program can actually do, but requires a whole system that can be executed. On the other hand, static analysers can be more lightweight and can be applied to incomplete systems, since they do not require the program to be run. However, the trade-off is that static analysis is generally more *imprecise* since it must approximate the possible behaviours of the program.

Linters should be fast and lightweight, as it is desirable for them to provide feedback to the user in real-time within the IDE. Furthermore, it is desirable for them to be able to provide warnings on incomplete programs and analyse even just fragments of code for suspicious code patterns. For these reasons, the vast majority of linters are static analysis tools.

2.1.2.2 Metaprogramming

Linters that provide code fixes are *metaprograms*: they are programs that manipulate programs. Auto-fixes and refactorings are performed as source-to-source transformations, which 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.

Analyses over ASTs are typically semantic-aware, respecting scoping rules and type information to provide more accurate code fixes. This semantic information is often provided by the compiler – some linters opt to run during compilation as a compiler plugin, while others run as a separate tool after compilation. Lint-on-compile and lint-after-compile tools both have their own advantages and disadvantages, and the choice between them depends on the use case [Sadowski et al. 2018].

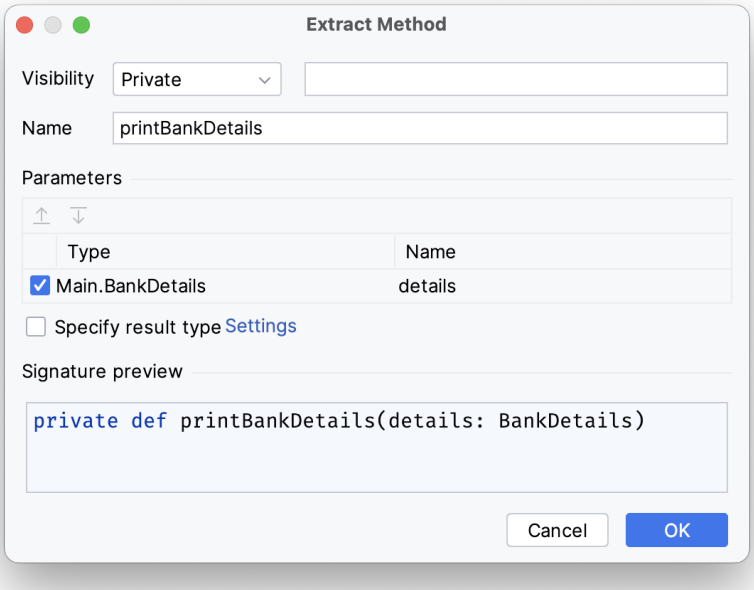
2.2 Writing Linters for Scala

The goal of `parsley-garnish` is to provide linting and refactoring capabilities for the `parsley` parser combinator library. Although `parsley` is its own domain-specific language, it is implemented as an embedded DSL utilising Scala as its host language. This is convenient for writing a linter for `parsley`, as it can be implemented using existing tools available to statically analyse Scala code, rather than having to write all the tooling from scratch.

[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.

2.2.1 Choice of Tooling

There are a few options available for implementing a linter for Scala, ranging from low-level compiler APIs to higher-level metaprogramming frameworks. This section discusses and evaluates 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. Many Scala tools are built with Scalameta, most notably the *Metals*[7] (Scalameta Language Server) LSP server for Scala. 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. Scalameta builds upon the earlier `scala.reflect` metaprogramming framework, unifying runtime and compile-time metaprogramming under a single API. Previously under `scala.reflect`, source code was parsed into lossy trees that discarded syntactic *trivia* such as comments and whitespace [Burmako 2017]. On the other hand, Scalameta trees are fully lossless and preserve all syntactic details: a crucial feature for code transformations and refactorings that must 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 fully preserve formatting details when pretty-printed, despite the underlying trees containing this information.

Scalafix *Scalafix*[8] is a code linting and rewriting tool built on top of Scalameta. It provides a higher-level API over Scalameta's tree transformation capabilities, specifically designed for implementing fine-grained code transformations that preserve the original formatting of the source file. Scalafix provides a framework for implementing linting rules to emit diagnostic 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.

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

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 generic code transformations and analyses. Scalafix rules can be either *syntactic* or *semantic* depending on whether they require semantic information, such as type signatures, to perform their analysis [Scala Center 2024]. Syntactic rules are faster to run, operating purely on the AST without needing to invoke compilation, but are more limited in the analyses they can perform. Semantic rules, on the other hand, are more complicated to run: they require augmenting the Scala compiler with the SemanticDB compiler plugin to provide Scalafix with the required semantic information. They are also slower as a result, since they require the source file to be compiled.

A drawback of Scalafix is that it is primarily a command-line tool, and currently does not provide an interactive interface within any code editor. Ideally, users should be able to see inline diagnostics, as well as preview and select auto-fixes interactively like in fig. 2.2. A Metals integration would be the most favourable outcome, as they already share a Scalameta backend, and it would allow Scalafix to be integrated into any code editor supporting the LSP protocol. However, at the time of writing, Scalafix only has basic integrations with Metals to run rules, but no interactive in-editor integrations [Jaglin 2024]. This feature request is an open issue on GitHub[9], but no timeline has been provided for its implementation.

Nonetheless, Scalafix emerges as the most favourable 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 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 diagnostics and does not support code fixes, making it unsuitable 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 cannot access semantic information, restricting the types of analyses it can perform.

2.2.2 A Crash Course on Scalafix

This section briefly explains how Scalafix rules are written, providing the Scalafix-specific terminology needed to understand the implementation details of `parsley-garnish`.

2.2.2.1 Rules

Scalafix rules are implemented as generic top-to-bottom traversals through the Scala AST, represented as a Scalameta `Tree` data structure. This is achieved with a collection-like API, exposing these operations in the style of standard Scala collection methods such as `traverse` and `collect`. During traversal, side effects are generated using the `Patch` data structure, which represents either a lint diagnostic or a code rewrite. Rules are applied file-by-file, and the patches generated by each rule are collected and applied by Scalafix as a batch rewrite after all rules have finished running. Fig. 2.3 thus showcases the basic structure of a simple syntactic rule:

- `doc.tree.collect` performs a tree traversal, where AST nodes of interest are visited via pattern matching.
- `case v: Defn.Val if !v.mods.exists(_.is[Mod.Lazy])` visits a `val` definition node that is not already marked as `lazy`.

[9] https://github.com/scalameta/metals-feature-requests/issues/230
[10] https://github.com/JetBrains/intellij-scala
[11] https://www.wartremover.org/
[12] http://www.scalastyle.org/

- `Patch.addLeft(v, "lazy ")` describes a patch to add the `lazy` modifier to the left side of the node that was matched on.

```
class MakeValsLazy extends SyntacticRule("MakeValsLazy") {
  override def fix(implicit doc: SyntacticDocument): Patch = {
    doc.tree.collect {
      case v: Defn.Val if !v.mods.exists(_.is[Mod.Lazy]) => Patch.addLeft(v, "lazy ")
    }.asPatch
  }
}
```

Fig. 2.3: A syntactic rule that makes all `val` definitions lazy.

2.2.2.2 Safer Patches Using Quasiquotes

A careful reader may notice from fig. 2.3 that the `Patch` rewrite method receives a raw string value, which seems unsafe and could potentially lead to malformed code. Indeed, Scalafix provides no guarantees that the output of a patch is a well-formed program, and it is the rule author's responsibility to ensure so.

In the majority of cases, `parsley-garnish` represents intended rewrites as Scalameta `Tree` objects, and only converts them to strings immediately before applying the patch. Fig. 2.4 shows how *quasiquotes* [Shabalin, Burmako, and Odersky 2013] can be used as syntactic sugar to construct trees in a convenient manner:

- `q"..."` is the quasiquote's string interpolation syntax to build a `Term` node, which is a subclass of `Tree`.
- `${...}` is the *splice* or *unquote* operation to insert a term into the constructed tree. Ordinarily, spliced values need to be an instance of a `Tree`, but simple values such as integers (in this case) can be automatically lifted into the appropriate tree node.
- `Tree.structure` represents the resulting `Tree` data structure as a string, useful for debugging.
- `Tree.syntax` pretty-prints the tree back into its source code representation, which can then be fed into a `Patch`.

Quasiquotes expand at compile-time into `Tree` constructors, so they do not preserve formatting details. This turns out to be very convenient for pretty-printing purposes: often, spliced values may contain superfluous parentheses, which are automatically removed (as seen in the final example within fig. 2.4).
By using quasiquotes instead of raw strings, the syntactic correctness of a rewrite is guaranteed. However, this safety is not foolproof: quasiquotes are not guaranteed to be well-typed or well-scoped, so the rewritten program still might not be able to compile. They are also not *hygienic*: generated code will not be able to avoid name clashes with regular code [Burmako 2017]. The lack of hygiene can cause issues with variable capture, allowing a variable to be unintentionally shadowed by a generated variable. Again, it is the rule author's responsibility to ensure that variable capture does not occur: §5.2 in part discusses how this is handled in `parsley-garnish`.

2.2.2.3 Semantic Information

Semantic rules are implemented in the same manner as fig. 2.3, but with the more powerful `SemanticDocument` as an implicit parameter. This allows rule authors to query AST nodes for their associated symbols: textual identifiers that represent definitions in the Scala program. Although symbols are not guaranteed to be unique outside the document they are defined in [Scalameta 2023], within the context of Scalafix rule authoring, they can be treated as unique since rules are applied per-file.

The following methods are the most important for querying the semantic information associated with a tree node:

- `Tree.symbol` retrieves the symbol associated with a node.

```scala
val x = 1
val tree = q"$x + 2"

println(tree.structure)
// Term.ApplyInfix(
//   Lit.Int(1),
//   Term.Name("+"),
//   Type.ArgClause(Nil),
//   Term.ArgClause(List(Lit.Int(2)), None)
// )

println(tree.syntax)
// 1 + 2
println(q"($tree)".syntax)
// 1 + 2
```

Fig. 2.4: Using Scalameta quasiquotes to construct a tree.

```scala
class IdentifyIntAddition extends SemanticRule("IdentifyIntAddition") {
  val intAdd = SymbolMatcher.normalized("scala.Int.`+`")

  override def fix(implicit doc: SemanticDocument): Patch = doc.tree.collect {
    case Term.ApplyInfix(_, intAdd(node), _, _) =>
      Patch.lint(IntAdditionLint(node))
  }.asPatch
}

case class IntAdditionLint(tree: Tree) extends Diagnostic {
  override def position: Position = tree.pos
  override def severity: LintSeverity = LintSeverity.Info
  override def message: String = "Found an addition operation on integers"
}
```

(a) A semantic rule that generates lint diagnostics *only* on integer addition operations.

```scala
object Main {
  val stringAdd = "parsley" + "garnish"
  val intAdd = 1 + 2
  val floatAdd = 1.0 + 2.0
}

// Main.scala:3:18: info: [IdentifyIntAddition]:
// Found an addition operation on integers
//   val intAdd = 1 + 2
//                  ^
```

(b) An example program showcasing the rule in fig. 2.5a.

Fig. 2.5: An example showing how semantic rules are semantic-aware on overloaded method names.

- `Symbol.info` queries a symbol for any `SymbolInformation` attached to it, including metadata such as its type signature.
- `Tree.synthetics` reveals extra compiler-generated tree nodes not present in the original source code, such as implicit conversions or inferred `.apply` methods.

Additionally, the `SymbolMatcher` trait provides a convenient way to create predicates that match specific symbols. Fig. 2.5 demonstrates how symbol matching can be used to detect integer addition operations, emitting a diagnostic warning when found. The `intAdd` symbol matcher only matches the `+` method on integers, so the rule does not emit a diagnostic for addition operations on other types, as seen in fig. 2.5b.

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) 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 overall boilerplate code: the programmer does not need to convert between the AST produced by the parser generator and their own AST. Instead, the results of parsers, known as their *semantic actions*, are incorporated as part of the parsers themselves.

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 a motivating reason for the development of `parsley-garnish`: to lower the barrier of entry for new users of the `parsley` library.

2.3.1 The `parsley` Library

`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 expands upon the example parser from the introduction to provide a further demonstration of `parsley`. As opposed to directly evaluating the results of the parser, this parser will instead construct an AST representing the structure of arithmetic expression language. Recall the EBNF grammar for the arithmetic expression language introduced in the introduction, shown again in fig. 2.6a. The following parser will parse an expression into the AST represented by the Scala datatype in fig. 2.6b. To demonstrate an alternate approach, this parser deals with left-recursion using the lower-level `chain.left1` combinator, rather than the higher-level precedence table API:

```
import parsley.Parsley
import parsley.character.digit
import parsley.expr.chain
```

⟨*digit*⟩ ::= '0' ... '9'
⟨*number*⟩ ::= ⟨*digit*⟩+
⟨*expr*⟩ ::= ⟨*expr*⟩ '+' ⟨*term*⟩ | ⟨*expr*⟩ '-' ⟨*term*⟩ | ⟨*term*⟩
⟨*term*⟩ ::= ⟨*term*⟩ '*' ⟨*atom*⟩ | ⟨*term*⟩ '/' ⟨*atom*⟩ | ⟨*atom*⟩
⟨*atom*⟩ ::= '(' ⟨*expr*⟩ ')' | ⟨*number*⟩

(a) The grammar in EBNF.

```
enum Expr {
  case Num(n: Int)
  case Add(x: Expr, y: Expr)
  case Sub(x: Expr, y: Expr)
  case Mul(x: Expr, y: Expr)
  case Div(x: Expr, y: Expr)
}
```

(b) The Scala AST to parse into.

Fig. 2.6: The grammar and AST for a simple arithmetic expression language.

```
import parsley.syntax.character.charLift

val number: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)

lazy val expr: Parsley[Expr] = chain.left1(term)('+' as Add(_, _) | '-' as Sub(_, _))
lazy val term: Parsley[Expr] = chain.left1(atom)('*' as Mul(_, _) | '/' as Div(_, _))
lazy val atom: Parsley[Expr] = '(' ~> expr <~ ')' | number.map(Num(_))

// Example usage: this parser constructs an AST, rather than evaluating its input
expr.parse("1+2*3/4") // => Add(Num(1), Div(Mul(Num(2), Num(3)), Num(4)))
```

Notice that although the form of the parser retains high-level similarities to the grammar, it is more verbose because it encodes semantic actions (i.e. constructing the resulting AST) directly in the parser.

- The `digit` combinator used in `number` is a parser that recognises a single digit character.
- The `number` parser uses the `foldLeft1` combinator to parse one or more digits, folding them together to form an integer.
- Left-associative parsing is idiomatically handled by the `chain.left1` combinator. This preserves the left-associative structure of the grammar, while avoiding left-recursion. Since the chain combinator makes it explicit that it is parsing a left-associative binary operator, it can be argued that this is a clearer and higher-level way of expressing the intent of the grammar rather than by left-recursion.
- The `'+' as Add(_, _)` parser consumes a `+` character, discarding its result and replacing it with the semantic action of constructing an `Add` node.
- The `map` combinator is used to transform the result of the `number` parser into a `Num` node, lifting its `Int` result to an `Expr`.
- The `~>` and `<~` operators guide the direction of the parser, ensuring that the parentheses are consumed but not included in the final `Expr`.

Design Patterns for Parser Combinators Willis and Wu [2022] describe several design patterns for writing maintainable parsers using parser combinators in Scala. They identify common problems and anti-patterns in parser design, and propose solutions in the form of design patterns. This provides a guideline for writing idiomatic `parsley` code for practical parser design, enabling opportunities for the development of linting and refactoring rules. These design patterns therefore form the basis for many linting rules implemented in `parsley-garnish`.

The example parser already utilises the *Implicit Conversions* design pattern, eliding the need to explicitly use the `char` combinator to parse characters. This is achieved by bringing `parsley.syntax.character.charLift` into scope, allowing Scala to automatically treat character literals as `char` parsers. It also uses the *Chain Combinators*

pattern to parse left-associative binary operators, rather than left-factoring the grammar to remove left-recursion – this is seen as an anti-pattern that obfuscates the intent of the grammar.

Parsing Expression Grammars Formally, the semantics of `parsley` parsers are based on parsing expression grammars (PEGs) [Ford 2004]. They differ from traditional context-free grammars (CFGs) [Mascarenhas, Medeiros, and Ierusalimschy 2014] in that the choice operator is ordered: e_1/e_2 will try e_2 only if e_1 fails. In contrast, the choice operator in a CFG is unbiased, which leads to the possibility of ambiguous grammars. The ordered choice operator means that PEGs cannot be ambiguous.

Chapter 3
Idiomatic Implicits Usage

Implicit conversions are a powerful feature in Scala, allowing users to supply an argument of one type when another is expected, to reduce boilerplate. When the Scala compiler encounters a type mismatch, it will search for any implicit methods that are in scope to convert the argument to the expected type. For example, the Scala standard library defines an implicit method to convert integers to wider `Long` values[1]. This allows any function expecting a `Long` to be passed an `Int` instead, without needing to explicitly call `toLong`.

```
implicit def int2long(x: Int): Long = x.toLong
```

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, allowing parsers to resemble the structure of a BNF grammar more closely:

```
// Without implicit conversions
string("parsley") | string("garnish")
// With implicit conversions
"parsley" | "garnish"
```

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.

3.1 Ambiguous Implicit Conversions

Unfortunately, implicit conversions are a double-edged sword. By their very nature, they can obscure the flow of the program, making it difficult to understand what 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` has first-class support for both design patterns, defining separate implicit methods for each:

- *Implicit Conversions*: the `parsley.syntax.character` package includes implicit methods `stringLift` and `charLift` to lift string and character literals into parsers, respectively.
- *Implicit Lexer*: the `Lexer` class exposes `implicitSymbol` to allow string and character literals to parse as proper tokens.

The two implicit conversions cannot be used in conjunction: the *Implicit Lexer* pattern is a specialisation of the former, so `implicitSymbol` is meant to be a *replacement* for `stringLift`. For novice users, this may not be immediately apparent and it is easy to accidentally bring both sets of these implicits into scope. Anecdotally, this issue has been encountered by a number of WACC students at Imperial. For example, consider the following code snippet:

```
val p = 'g' ~> "arnish"
p.parse("garnish")
// [error] type mismatch;
//   found   : String("arnish")
//   required: Parsley[?]
//  Note that implicit conversions are not applicable because they are ambiguous:
```

[1] https://github.com/scala/scala/blob/v2.13.14/src/library/scala/Int.scala

```
//   both method stringLift in object character of type (str: String): Parsley[String]
//   and method implicitSymbol in class ImplicitSymbol of type (s: String): Parsley[Unit]
//   are possible conversion functions from String("arnish") to Parsley[?]
//    val p = 'g' ~> "arnish"
//                   ^^^^^^^^
```

In this instance, 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'
//            ^^^^^^^^^^^
```

Syntactic or semantic rule? 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[2], 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 implemented as a syntactic rule. Consequently, the solution takes a different approach: examining a file's import clauses to deduce the presence of clashing implicits.

Example Fig. 3.1a 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 similar to that shown in fig. 3.1b.

3.1.1 Implementation as a Syntactic Lint Rule

The rule uses the following heuristics to determine if an import clause brings an implicit into scope:

- An import clause of the form `import parsley.syntax.character.xxx`, where `xxx` is either a wildcard import or specifically contains the importee `stringLift`, indicates the *Implicit Conversions* pattern on strings.
- The import path for the *Implicit Lexer* pattern is subject to how the user names their lexer object, so determining this is much more heuristic-based. The following cases are considered indicative of this pattern:
  - An import containing the keyword `lexer`, as well as either `implicit` or `implicits`.
  - An import clause that brings in an importee called `implicitSymbol`.

Unlike Java, Scala allows import clauses to appear anywhere in source files, not just at the top of the file. The flexibility of Scala's import system has two main repercussions:

1. Imports are visible only within the scope in which they are defined. For example, the members imported by lines 2–3 in fig. 3.1a are only available in the `parser` object, and not visible within `lexer`.

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

```scala
object parser {
  import parsley.syntax.character.stringLift
  import lexer.implicits._
  val p = "garnis" <~ 'h'
}

import parsley.token.Lexer
object lexer {
  import parsley.token.descriptions.LexicalDesc
  private val lexer = new Lexer(LexicalDesc.plain)
  val implicits = lexer.lexeme.symbol.implicits
}
```

(a) 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._
  ^^^^^^^^^^^^^^^^^^^^^^^^
```

(b) The warning message reported at line 3 by the *Ambiguous Implicit Conversions* lint rule.

Fig. 3.1: Example of the *Ambiguous Implicit Conversions* lint rule in action.

2. Imports are resolved lexically in a top-down order within the source file, so members cannot be referenced at any point before they were imported. In fig. 3.1a, the import on line 7 brings parsley's `Lexer` into scope at any scope after that line, but will not be visible before that point.

This behaviour presents extra challenges for this lint rule, as it must be aware of the exact scope of each import to determine which implicits are in scope at any given point.

The lexically ordered management of scope can be respected by visiting all import clauses in the order they appear in the source file. By treating the parent of an AST node as its enclosing scope, it is possible to use ancestor information to determine the scoping relation between two nodes. Scoping rules can then be respected by using this information to determine which previously visited imports are still in scope at any given point in the traversal.

```
extension (tree: Tree) {
  def isWithinScope(scope: Tree): Boolean =
    tree.isEqual(scope) || tree.parent.exists(_.isWithinScope(scope))
}
```

With this, fig. 3.2 shows the core logic of the rule, implemented as a *tail-recursive* visit over the list of imports, which allows the Scala compiler to optimise the recursion into an efficient iterative loop. The `getClashingImports` method uses the heuristic rules described above to determine which implicits are brought into scope by an import. If an import brings in both types of implicits for the first time within its enclosing scope, a patch is generated to report a diagnostic. The traversal then continues recursively, updating its list of visited imports.

The overall entry point of the rule is thus defined as follows, where the import clauses are gathered in order via a top-down traversal of the AST:

```
override def fix(implicit doc: SyntacticDocument): Patch = {
  val orderedImports = doc.tree.collect { case i: Import => i }
  visit(orderedImports, List.empty, Patch.empty)
}

def visit(unvisited: List[Import], visited: List[Import], patches: Patch): Patch = {
  unvisited match {
    case curr :: remaining =>
      val currentScope = curr.parent.get // Parent node represents import's scope
      // Which imports are in scope at this point: lexical ordering is maintained
      // by only considering imports that have been defined in the file so far
      val alreadyInScope = visited.filter(currentScope.isWithinScope(_.parent.get))
      val currentInScope = alreadyInScope :+ curr
      val clashingImports = getClashingImports(currentInScope)

      if (getClashingImports(alreadyInScope).isEmpty && clashingImports.nonEmpty) {
        visit(remaining, currentInScope, patches +
          Patch.lint(AmbiguousImplicitConversionsLint(curr, clashingImports)))
      } else {
        // Continue traversal, discarding imports that have fallen out of scope
        visit(remaining, currentInScope, patches)
      }
    case Nil => patches
  }
}
```

Fig. 3.2: Implementation of the core logic for the *Ambiguous Implicit Conversions* lint rule.

3.2 Remove Explicit Usage of Implicit Conversions

parsley users who are new to Scala may not be aware that implicit conversions are automatically applied by the Scala compiler. This misconception has been observed in the wild (i.e. WACC students in the depths of the Imperial computer labs), where users have treated parsley's implicit methods as regular combinators that are explicitly applied to values.

Example This mistake can be classified as a code smell – although it does not cause compilation errors, it results in unnecessarily verbose parsers as seen in fig. 3.3a. Writing parsers like this is counterproductive, as it negates the purpose of the *Implicit Conversions* patterns. Fig. 3.3b shows how the same parsers should be idiomatically written, removing the explicit application of implicit conversion functions: as long as the implicits are in scope, the Scala compiler will automatically apply them.

```scala
import parsley.syntax.character._
val p: Parsley[String] = stringLift("parsley")
// Using the implicit method's fully qualified name instead of the import
val q: Parsley[Char] = parsley.syntax.character.charLift('p')
```

(a) Example parsers which are unnecessarily verbose by explicitly applying implicit conversions.

```scala
import parsley.syntax.character._
val p: Parsley[String] = "parsley"
// Using the implicit method's fully qualified name instead of the import
import parsley.syntax.character.charLift
val q: Parsley[Char] = 'p'
```

(b) The same parsers, but with the explicit implicit conversions removed.

Fig. 3.3: Example of the *No Explicit Implicit Conversions* rewrite rule in action.

3.2.1 Implementation as a Semantic Rewrite Rule

Unlike the previous rule, this is a situation where a rewrite rule can be applied to automatically fix the issue. Furthermore, since this issue arises on successfully compiled code, it can be implemented as a Scalafix semantic rule. With access to the semantic API, the rule can resolve the origin symbol of a function name and determine if it belongs to parsley's set of implicit methods, even if the method is imported under a different name:

```scala
val implicitConv = SymbolMatcher.normalized(
  "parsley.syntax.character.charLift",
  "parsley.syntax.character.stringLift",
  "parsley.token.symbol.ImplicitSymbol.implicitSymbol"
)

override def fix(implicit doc: SemanticDocument) = doc.tree.collect {
  // Match a function call with a single argument, using the symbol matcher 'implicitConv'
  // The syntactic naming of the function is ignored,
  // as the symbol matcher will pattern match based on its original symbolic name
  case app @ Term.Apply(implicitConv(_), Term.ArgClause(List(liftedArg), _)) =>
    // Replace the function call with just its argument
    Patch.replaceTree(app, liftedArg.syntax)
}.asPatch
```

There is an uncommon edge case, though, where the implicit conversion is not fully brought into scope by an import, but is instead accessed via a qualified name. The `q` parser in fig. 3.3a demonstrates this, where the `charLift` implicit method is referenced directly by its fully qualified name. In this case, the implicit method has to be imported to bring it into scope. This case can be detected by checking if the function call is on a `Term.Select` node, representing a method call on a qualified name:

```
case qualifiedApp @ Term.Apply(
  // Specific case where the 'implicitConv' matches on a qualified Term.Select node
  qual @ implicitConv(_: Term.Select), Term.ArgClause(List(liftedArg), _)
) =>
  // Original patch to remove the explicit function call
  val removeExplicitCall = Patch.replaceTree(qualifiedApp, liftedArg.syntax)
  // Extra patch to add the import clause for the implicit method
  val importQualifiedName = addPatchAbove(qualifiedApp, s"import $qual")
  // Combine the patches to apply both rewrites
  removeExplicitCall + importQualifiedName
```

This places an import clause for the implicit method directly above the qualified method call, as shown in fig. 3.3b. Placing the import next to the implicit conversion is a deliberate design choice: it localises the scope of the implicit conversion to a smaller region, reducing the risk of causing clashing implicits in the global scope. An even more foolproof method would be to wrap the entire section into its own scope, but this would introduce a lot of syntactic noise, when the chances that such a scoping issue would arise is low.

Summary

This chapter introduced two rules to aid users with idiomatic usage of the *Implicit Conversions* family of design patterns. It also serves as a lightweight demonstration of how to implement both syntactic and semantic rules in Scalafix, and why each type of rule is appropriate for the respective issue at hand:

- *Ambiguous Implicit Conversions* (§3.1) is restricted to being a *syntactic* rule, since the nature of the issue means that it must be caught before compilation. The rule issues *lint diagnostics* to warn users of the issue, but cannot automatically fix it as it does not have enough information to resolve the ambiguity.
- *No Explicit Implicit Conversions* (§3.2) is a *semantic* rule as it can be run in a post-compiled state. By having access to semantic information, the linter can be more confident in resolving symbols, rather than relying on syntactic heuristics. The issue is also simpler in nature and is automatically solvable in the general case, so the rule provides *code rewrite* capabilities.

Chapter 4
Removing Left-Recursion

Left-recursive grammar rules are a common pattern to represent left-associativity. Take for example the following definition of a left-associative addition operator:

$$\langle expr \rangle ::= \langle expr \rangle \text{ `+' } \langle term \rangle \mid \langle term \rangle$$

Since the first production of $\langle expr \rangle$ is itself, this rule is said to be left-recursive. This poses a problem for recursive-descent parsers, such as those that `parsley` produces: it will try to parse $\langle expr \rangle$ by first trying to parse $\langle expr \rangle$, and so on, resulting in an unproductive infinite loop.

Although it is possible to address the issue by transforming the grammar with algorithms such as Paull's algorithm [Moore 2000], in the context of parser combinators this is considered an anti-pattern by Willis and Wu [2021]. They argue that this transformation obscures the original intent of the grammar, and exposes lower-level implementation details when this can be abstracted behind a combinator. Instead, they propose that the idiomatic method to handle left-recursion in parser combinators is to use the `chain` family of combinators [Fokker 1995]. These combinators embody the behaviour of right-associating left-recursive rules and correcting the result back to a left-associative form.

Left-recursion often comes as a nasty surprise for novice users naïvely translating BNF grammars into parser implementations – this issue is not unique to parser combinators, but also extends to many popular parser generators that use recursive-descent. Thus, it would be beneficial to provide a linting rule for `parsley` that can warn users when parsers are left-recursive. In fact, the next major release of `parsley` 5.0 will introduce a `detectDivergence` combinator, which performs *dynamic* analysis to detect unproductive looping at runtime. Therefore, `parsley-garnish` could complement this functionality with an auto-fix rule to refactor left-recursive parsers to use `parsley`'s idiomatic `chain` combinators.

Running example The following left-recursive parser and its transformation into a non-left-recursive form will be used as an example for this chapter:

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

The `example` parser intends to express the following simple grammar expressed using left-recursion. The goal is to refactor `example` so that it retains the intended semantics, but is transformed into a parser that `parsley` can handle correctly.

$$\langle example \rangle ::= \langle example \rangle \text{ “a” } \mid \text{ “b”}$$

4.1 The Left-Recursion Factoring Transformation

`parsley-garnish` bases its left-recursion factoring transformation on the work of Baars and Swierstra [2004], adapted to fit the PEG semantics of `parsley`. At a high-level, the transformation involves "unfolding" each non-terminal production into three parts:

- `results`: The semantic actions of the parser, if it can derive the empty string. Conceptually, this has type `Option[A]` where `A` is the type of the result.
- `nonLeftRec`: The non-left-recursive part of the parser that does not derive the empty string. This will have some type `Parsley[A]`.
- `leftRec`: The left-recursive call, which in the general left-recursive case, corresponds to a repeated postfix operator of type `Parsley[A => A]`. This is a function which requires the semantics of the left-recursive non-terminal argument.

This transformation is applied in-order to each parser in the source file, replacing the original parser with its factored form if it was left-recursive. An unfolded parser is recombined using `chain.postfix`: this combinator

encapsulates the general form of left-associative parsing, and most other iterative combinators can be derived from it [Willis 2024].

```
def transform(
  results: Option[A],
  nonLeftRec: Parsley[A],
  leftRec: Parsley[A => A]
): Parsley[A] = {
  val result: Parsley[A] = results match {
    case None    => empty
    case Some(x) => pure(x)
  }
  // The type signature of postfix in Parsley:
  // def postfix[A](p: Parsley[A])(op: => Parsley[A => A]): Parsley[A]
  chain.postfix(nonLeftRec | results)(leftRec)
}
```

4.2 Necessary Infrastructure

The comparatively simple linting rules discussed in the previous chapter were implemented by directly inspecting the generic Scala AST provided by Scalafix. However, even though `parsley` programs are written in Scala, it is important to remember that `parsley` is a DSL borrowing Scala as a host language. Domain-specific transformations like left-recursion factoring are therefore naturally defined as transformations on the `parsley` AST, at a higher level of abstraction than the generic Scala AST. Thus, this section discusses the extra infrastructure used to support the left-recursion factoring transformation:

- Firstly, §4.2.1 motivates the idea of using an intermediate AST representation for parsers, distinct from the general-purpose Scala AST.
- Following this, §4.2.2 shows how the AST of a Scala source file is converted into this intermediate representation.
- Finally, §4.2.3 discusses how the intermediate AST is converted back into Scala code so that it can be applied as a Scalafix patch.

4.2.1 An Intermediate AST

The transformations described by Baars and Swierstra [2004] require an explicit representation of the grammar and production rules so that they can be inspected and manipulated before generating code. They achieve this by representing parsers as a deep-embedded datatype in the form of an intermediate AST, in a similar manner to `parsley`.

Since `parsley-garnish` is a linter, by nature, it has access to an explicit grammar representation in the form of the full `scala.meta.Tree` AST of the source program. However, this datatype represents general-purpose abstract Scala syntax, rather than the abstract syntax of a specialised parser combinator DSL. This makes it not well-suited for performing domain-specific operations over the AST.

Take for example the task of combining two AST nodes `Term.Name("p")` and `Term.Name("q")`, representing named parsers `p` and `q`, with the combinator `<*>` (pronounced "ap"). This operation can be concisely expressed with Scalameta quasiquotes, rather than manually writing out the full explicit AST:

```
q"p <*> q" ==
  Term.ApplyInfix(
    Term.Name("p"),
    Term.Name("<*>"),
    Type.ArgClause(Nil),
```

```
    Term.ArgClause(List(Term.Name("q")), None)
  )
```

However, the operation of inspecting the individual parsers p and q is not as straightforward. Although quasiquotes can be used as extractor patterns in pattern matching, this usage is discouraged due to limitations in their design that makes it easy to accidentally introduce match errors[1]. Thus, extracting the parsers necessitates a long-winded pattern match like so:

```
val ap = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

def deconstructAp(parser: Term) = parser match {
  case Term.ApplyInfix(p, ap(_), _, Term.ArgClause(List(q), _)) => (p, q)
}
```

This involves dealing with abstract general-purpose syntax constructs like `Term.ApplyInfix`, which are low-level details not relevant to the task of manipulating parsers. Although this is not an issue for simple one-off transformations, for more specialised transformations like left-recursion factoring, it would be desirable to abstract away from these low-level syntactic details. This motivates the need for an higher-level, intermediate AST representation that is more specialised to the domain of parser combinators.

4.2.1.1 The Parser ADT

parsley-garnish therefore takes a similar approach as Baars and Swierstra [2004] and parsley itself, building an intermediate AST as a deep-embedded parser combinator tree. Fig. 4.1 shows how this is implemented as a `Parser` algebraic data type (ADT). All `Parser` types represent parsley combinators, with the sole exception of `NonTerminal` to represent references to named parsers.

```
trait Parser
case class NonTerminal(ref: Symbol) extends Parser
case class Pure(x: Term) extends Parser
case object Empty extends Parser
case class <*>(p: Parser, q: Parser) extends Parser
case class <|>(p: Parser, q: Parser) extends Parser
case class Str(s: String) extends Parser
case class Chr(c: Char) extends Parser
```

Fig. 4.1: A subset of the `Parser` ADT, representing the core combinators in parsley-garnish.

Deconstructing parsers Scala allows users to define symbolic class names (as evidenced by the definitions of `<*>` and `<|>` in fig. 4.1), and provides syntactic sugar to pattern match on these constructors using infix notation. This results in a very natural and readable pattern matching syntax:

```
def deconstructAp(parser: Parser) = parser match {
  case p <*> q => (p, q)
}
```

Constructing parsers Defining infix operators as extension methods on the `Parser` trait provides a similar syntactic sugar for constructing parsers:

```
extension (p: Parser) {
  def <*>(q: Parser) = <*>(p, q)
  def |(q: Parser) = <|>(p, q)
  def map(f: Term) = FMap(p, f)
```

[1] https://scalameta.org/docs/trees/guide.html#with-quasiquotes-1

```
}
extension (ps: List[Parser]) {
  def zipped(f: Term) = Zipped(f, ps)
}
```

This makes the syntax for writing `Parser` terms feel natural and similar to writing `parsley` code. For example, notice how constructing the *code* representation of the `example` parser resembles how the original parser itself would be written:

```
val EXAMPLE = NonTerminal(Sym(Term.Name("example").symbol))

// val example: Parsley[String] =     (example,  string("a")).zipped(  _ + _ ) | string("b")
   val example: Parser          = List(EXAMPLE,     Str("a")).zipped(q"_ + _") |    Str("b")
```

4.2.2 Lifting to the Intermediate Parser AST

Converting the raw Scala AST to this intermediate parser combinator AST requires the following basic operations:

1. Identifying all named parsers defined in the source program – these correspond to non-terminal symbols in the grammar.
2. Lifting the definition each parser into the intermediate AST, i.e. a `Parser` object.
3. Collecting these into a map to represent the high-level grammar – the unique symbol of each named parser is mapped to its corresponding `Parser` object, along with extra metadata required for the transformation.

Most importantly, this metadata includes a reference to a parser's original node in the Scala AST, so lint diagnostics or code rewrites can be applied to the correct location in the source file:

```
case class ParserDefn(name: Term.Name, parser: Parser, tpe: Type.Name, originalTree: Term)
```

4.2.2.1 Identifying Named Parsers

Finding AST nodes corresponding to the definition sites of named parsers involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be some `Parsley[_]`. This type information is accessed by querying the Scalafix semantic API for the node's symbol information. Consider the labelled AST structure of the `example` parser:

```
// lazy val example: Parsley[String] = (example, string("a")).zipped(_ + _) | string("b")
// ^^^^     ^^^^^^^  ^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// mods      pats        decltpe                              rhs

val exampleTree = Defn.Val(
  mods = List(Mod.Lazy()),
  pats = List(Pat.Var(Term.Name("example"))),
  decltpe = Some(
    Type.Apply(Type.Name("Parsley"), Type.ArgClause(List(Type.Name("String"))))
  ),
  rhs = Term.ApplyInfix(...)
)
```

Note that the `decltpe` field refers to the *syntax* of the explicit type annotation, not the *semantic* information the variable's inferred type. Therefore, this field will not always be present, so in the general case, the type must be queried via a symbol information lookup:

```
exampleTree match {
  case Defn.Val(_, List(Pat.Var(varName)), _, body) =>
```

```
    println(s"qualified symbol = ${varName.symbol}")
    // Query the symbol information of the variable name, and get its type signature
    varName.symbol.info.get.signature match {
      // Scalameta treats this as a zero-arg method, so the relevant part is its return type
      case MethodSignature(_, _, returnType) =>
        println(s"type = $returnType")
        println(s"structure of type object = ${returnType.structure}")
    }
}
// qualified symbol = path/to/package/ObjectName.example.
// type = Parsley[String]
// structure of type object = TypeRef(
//   NoType,
//   Symbol("parsley/Parsley#"),
//   List(TypeRef(NoType, Symbol("scala/Predef.String#"), List()))
// )
```

Seeing that the type of this AST node is `Parsley[String]`, `parsley-garnish` can then proceed to convert the `rhs` term into a `Parser` ADT object. The map entry uses the fully qualified symbol for `example` as the key, and the lifted `Parser` object as the value.

4.2.2.2 Converting Scalameta Terms to the Parser ADT

Having identified the AST nodes which represent parsers, they need to be transformed into the appropriate `Parser` representation. This involves pattern matching on the `scala.meta.Term` to determine which parser combinator it represents, and then constructing the appropriate `Parser` instance.

Each `Parser` defines a partial function `fromTerm` to instantiate a parser from the appropriate `scala.meta.Term`. These `fromTerm` methods perform the menial work of pattern matching on the low-level syntactic constructs of the Scala AST. All `fromTerm` methods are combined to define the `toParser` extension method on `scala.meta.Term` – this is where AST nodes are lifted to their corresponding `Parser` representation.

The pattern matching example from §4.2.1 makes a reappearance in the definition of `<*>.fromTerm`, where the arguments to the `<*>` combinator are instead recursively lifted to `Parser` objects:

```
// Type signatures in Parsley:
// p: Parsley[A => B], q: =>Parsley[A], p <*> q: Parsley[B]
case class <*>(p: Parser, q: Parser) extends Parser
object <*> {
  // Match the specific symbol for parsley's <*> combinator
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

  def fromTerm: PartialFunction[Term, <*>] = {
    // Pattern match succeeds only if the term has the structure 'p <*> q'
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
      p.toParser <*> q.toParser
  }
}
```

Where a combinator takes a non-parser argument, this is treated as a black box and kept as a raw AST node of type `scala.meta.Term`:

```
// x: A, pure(x): Parsley[A]
case class Pure(x: Term) extends Parser
object Pure {
  val matcher = SymbolMatcher.normalized("parsley.ParsleyImpl.pure")
```

```
  def fromTerm: PartialFunction[Term, Pure] = {
    // expr is an opaque AST node that can't be further inspected
    case Term.Apply(matcher(_), Term.ArgClause(List(expr), _)) => Pure(expr)
  }
}
```

4.2.2.3 Building the Grammar Map

The overall process of converting the source file AST to a high-level map of the grammar can therefore be expressed as a single traversal over the AST:

```
// Encapsulate all valid pattern matches into a single extractor object
object VariableDecl {
  def unapply(tree: Tree): ParserDefn = tree match {
    // isParsleyType uses symbol info to check if variable type is Parsley[_]
    case Defn.Val(_, List(Pat.Var(varName)), _, body) if isParsleyType(varName) =>
      // If the pattern match is successful, convert the definition body to a Parser
      // Collect metadata and bundle into a parser definition object
      ParserDefn(
        name = varName,
        parser = body.toParser,
        tpe = getParsleyType(varName),
        originalTree = body
      )
    // ... similar cases for Defn.Var and Defn.Def
  }
}

type Grammar = Map[Symbol, ParserDefn]
val grammar: Grammar = doc.tree.collect {
  // Every AST node that satisfies the pattern match is added to the map
  case VariableDecl(parserDef) => parserDefn.name.symbol -> parserDef
}.toMap
```

4.2.3 Lowering Back to the Scalameta AST

After all necessary transformations have been applied to parser terms, the final step is to convert them back to a textual representation to be applied as a Scalafix patch. Parsers can be lowered back to `scala.meta.Term` nodes by the inverse of the original `fromTerm` transformation. The `Parser` trait defines this transformation as the method `term`, using quasiquotes to simplify the construction of the `scala.meta.Term` nodes. For example:

```
case class Zipped(func: Function, parsers: List[Parser]) extends Parser {
  val term: Term = q"(..${parsers.map(_.term)}).zipped(${func.term})"
}
```

The `Parser` trait can then define a `toString` method based on Scalameta's AST pretty-printer on the `term` field. This "stringified" term is therefore guaranteed to be syntactically well-formed and is the appropriate type expected by Scalafix's `Patch` methods.

4.3 Implementing the Left-Recursion Transformation

With the `Parser` AST defined, it is now possible to implement the left-recursion factoring transformation in a more high-level manner. Using the `grammar` map collected previously, the overall transformation can be summarised as follows (in Scala-like pseudocode for brevity):

```
val transformedGrammar = grammar.to(mutable.Map)
for (currentNT <- grammar.keys) {
  // Unfold each non-terminal parser into its three parts
  val (results, nonLeftRec, leftRec) = unfold(transformedGrammar, currentNT)
  // Update the grammar with the recombined postfix parser if it was left-recursive
  transformedGrammar(currentNT).parser = transform(results, nonLeftRec, leftRec)
}
// Generate rewrite patches for each parser definition
transformedGrammar.mapValues { defn =>
  Patch.replaceTree(defn.originalTree, defn.parser.toString)
}
```

The unfolding transformation is defined as follows, which is dynamically dispatched to the appropriate combinator's `unfold` method:

```
def unfold(env: Grammar, currentNT: Symbol): UnfoldedParser =
  env(currentNT).parser.unfold(currentNT, env, visited = Set.empty)
```

4.3.1 Unfolding the Core Combinators

The most important core combinators in the left-recursion transformation are `NonTerminal`, `Pure`, `Empty`, `<|>`, and `<*>`. Character combinators such as `Str` (`string`), `Chr` (`char`), and `Item` (`item`), although technically handled by `parsley-garnish` as primitives, are grouped together as they behave in a simple and uniform manner. The rest of the combinators in `parsley` are defined in terms of the primitive combinators, and are desugared into the core combinators during unfolding.

Non-terminals Non-terminals are the primary base case of the recursive unfolding transformation. There are three cases to consider when a non-terminal symbol is encountered:

- This non-terminal is the same as the current non-terminal being analysed, which means it may be in a left-recursive position. Recall that the `leftRec` result corresponds to a function, where the call to the left-recursive non-terminal is stripped off. Therefore in this case, the left-recursive call is the `identity` function which just returns its argument, i.e. the non-terminal itself. If the non-terminal is not in left-recursive position, this doesn't matter as the `leftRec` result will be subsumed by the `nonLeftRec` result in the final recombination.
- The non-terminal refers to a different non-terminal, but has already been visited in the current unfolding process. This means it shouldn't be recursively visited again, as it has already been factored out. It is not left-recursive and the `nonLeftRec` part is just a reference to this non-terminal's name.
- The non-terminal refers to a different non-terminal that has not been visited yet. In this case, the non-terminal is recursively unfolded, and the `visited` set is updated to include this non-terminal.

The three cases can be implemented as so:

```
case class NonTerminal(ref: Symbol) extends Parser {
  def unfold(currentNT: Symbol, env: Grammar) =
    if (ref == currentNT) (None, Empty, Pure(q"identity"))
    else if (visited.contains(ref)) (None, NonTerminal(ref), Empty)
    else env(ref).parser.unfold(currentNT, env, visited + ref)
}
```

Base cases The base cases are mostly straightforward. The unfolded `result` portion can be reasoned with by the combinators' correspondence to PEG expressions [Eichenroth, Rein, and Hirschfeld 2022], as shown in table 4.1. If the combinator can succeed by parsing the empty string, `result` is a `Some` value of its semantic

action, otherwise it is `None`. The other two portions are also easy: none are left-recursive, and if the combinator consumes input on success then `nonLeftRec` is just the combinator itself. Thus, the base cases can be summed up as follows:

```
/* Pure */                 (Some(x), Empty, Empty)
/* Empty */                (None, Empty, Empty)
/* p: Str | Chr | Item */  (None, p, Empty)
```

Combinator	Can succeed without consuming input?
`pure(x)`	Yes, semantic action is `x`.
`empty`	No, combinator fails immediately regardless of input.
`string(s)`	Functionally no. Can theoretically be given the empty string `""`, but this is illegal in `parsley` and triggers a runtime exception.
`char(c)`	No, always consumes the given character.
`item`	No, always consumes any single character.

Table 4.1: Semantic action (`result`) portion behaviour for each base case.

Choice (<|>) This recursive case is also relatively simple: each branch is unfolded, then the `nonLeftRec` and `leftRec` portions are recombined with the `|` combinator. The `result` portion is that of the left branch if it can derive the empty string, otherwise it is the right branch. This behaviour differs slightly from the original presentation of the transformation by Baars and Swierstra [2004], where `|` is based on CFG semantics. The consequence of this is that choice is unbiased in the original transformation, triggering an ambiguity error if both branches can succeed without consuming input. However, since `parsley` uses PEG semantics, this ambiguity is not possible – the choice operation represented by the `|` combinator is *left-biased*, so if the left parser succeeds the right parser is not attempted.

```
case class <|>(p: Parser, q: Parser) extends Parser {
  def unfold() = {
    val (pe, pn, pl) = p.unfold
    val (qe, qn, ql) = q.unfold
    (pe.orElse(qe), pn | qn, pl | ql)
  }
}
```

Ap (<*>) This is an important case, as `parsley-garnish` chooses to use `<*>` as the primitive combinator for composing parsers. It is also the most complex:

- `p <*> q` can succeed without consuming input only if *both* `p` and `q` can do so. Parser combinators abide by the law that `pure(f) <*> pure(x) = pure(f(x))`, so the resulting semantic action is that of `p` applied as a function to the semantic action of `q`.
- In the `nonLeftRec` and `leftRec` portions, if `p` can succeed without consuming input, then `q` must be subjected to the semantic value corresponding to `p` [Swierstra and Duponcheel 1996]. This is achieved using the result shown in eq. (4.1), which is a special case of grammar right-factoring in reverse.
  - The `leftRec` portion requires re-associating semantic actions back to the left, which requires using the `flip` higher-order function. Thus, the left branch of the RHS of eq. (4.1) is reformulated as `p.map(flip) <*> q`.
  - Since `leftRec` is a parser of a function, the right branch also needs adjusting: the function composition operation must be used in order to apply the function to `q` with the `<*>` combinator.

```
(p <|> pure f) <*> q   ⇒   (p <*> q) <|> (f <$> q)
```
(4.1)

Given the above observations, the implementation of the `<*>` combinator's unfolding is as follows:

```
case class <*>(p: Parser, q: Parser) extends Parser {
  def unfold() = {
    val (pe, pn, pl) = p.unfold
    val (qe, qn, ql) = q.unfold

    val result = if (pe.isDefined && qe.isDefined) Some(q"${pe.get}(${qe.get})") else None
    val nonLefts = {
      val lnl = pn <*> q
      val rnl = pe.map(q"f => qn.map(f)").getOrElse(Empty) // f <$> qn
      lnl | rnl
    }
    val lefts = {
      val llr = pl.map(q"flip") <*> q
      val rlr = pe.map(q"$ql.map(compose)").getOrElse(Empty) // f.map(compose) <$> ql
      llr | rlr
    }

    (result, nonLefts, lefts)
  }
}
```

4.3.2 Composite Combinators

The remaining combinators are defined in terms of the core combinators, and are unfolded by recursively unfolding their constituent parts. For example, the `map` combinator is defined as `p.map(f) = pure(f) <*> p`, so its unfolding is simply implemented as:

```
case class FMap(p: Parser, f: Term) extends Parser {
  def unfold() = (Pure(f) <*> p).unfold
}
```

Further high-level combinators are defined in a similar manner. The improved sequencing combinators `lift` and `zipped`, as well as bridge constructors using the *Parser Bridges* design pattern [Willis and Wu 2022], represent a further generalisation of the `map` combinator:

```
liftN(f, p1, ..., pN) // explicit lift syntax, where f is a function of arity N
f.lift(p1, ..., pN) // implicit lift syntax
(p1, ..., pN).zipped(f) // zipped syntax, improves Scala's ability to infer types
F(p1, ..., pN) // parser bridge pattern, abstracts the construction of f behind bridge F

// All are equivalent to the following desugared form,
// which is idiomatic in Haskell, but not in Scala for performance reasons
= pure(f.curried) <*> p1 <*> ... <*> pN
```

These are handled in `parsley-garnish` as the `LiftLike` family of combinators, which converts them into the desugared form in terms of core combinators `pure` and `<*>`, allowing the unfolding transformation to be applied:

```
trait LiftLike extends Parser {
  // foldLeft builds a parser of the form 'pure(f.curried) <*> p1 <*> ... <*> pN'
  def unfold() = parsers.foldLeft(Pure(q"$func.curried"))(_ <*> _).unfold
}
```

4.3.3 Defining Utility Functions

In various places within the unfolding transformations, three higher-order functions were utilised which may not be provided by the Scala standard library:

- The flip function reverses the order of arguments applied to a function. This isn't defined in the standard library, so it must be defined manually.
- Function composition is defined in the standard library, but a more versatile curried version is required by the transformation, so it is also defined manually.
- The identity function `identity[A]: A => A` is defined in the standard library.

Therefore, `parsley-garnish` will insert the following definitions into the source file as a patch:

```
def flip[A, B, C](f: A => B => C)(x: B)(y: A): C = f(y)(x)
def compose[A, B, C](f: B => C)(g: A => B)(x: A): C = f(g(x))
```

Although this isn't very aesthetically pleasing, it is necessary to bring these higher-order functions into scope for the transformed code to make use of them.

Success...?

With the unfolding transformations defined for all core combinators, the left-recursion factoring transformation is now complete. Running the transformation on the `example` parser yields the output in fig. 4.2.

```
def flip[A, B, C](f: A => B => C)(x: B)(y: A): C = f(y)(x)
def compose[A, B, C](f: B => C)(g: A => B)(x: A): C = f(g(x))

lazy val example: Parsley[String] = chain.postfix(
  empty | (empty.map((_ + _).curried) | empty <*> example) <*> string("a")
    | string("b") | empty
)(
  (empty.map(flip) <*> example | pure(identity).map(compose((_ + _).curried)))
    .map(flip) <*> string("a")
    | empty | empty
)
```

Fig. 4.2: The initial attempt at factoring out left-recursion from the `example` parser.

This is... disappointing, to say the least. There are *many* things wrong with the transformed output:

- The parser is horrendously complex and unreadable, its intent entirely obfuscated in a sea of combinators. It's especially frustrating that there are so many `empty` combinators, when `p | empty` and `empty | p` are both actually just equivalent to `p`.
- Having to define the `flip` and `compose` functions is not ideal, but inlining them as lambdas would make the code even worse.
- Even worse, the parser does not even typecheck – unlike classical Hindley-Milner-based type systems, Scala only has *local* type inference [Cremet et al. 2006]. As a result, the compiler is unable to correctly infer correct types for `flip` and also asks for explicit type annotations in the lambda `(_ + _).curried`.

The result is discouraging especially because it is not impossible to factor out the left-recursion in a nice manner: a hand-written equivalent using `postfix` would resemble the concisely defined parser in fig. 4.3. There is still hope, though – if the `empty` combinators can be removed and something is done about the higher-order functions, perhaps fig. 4.2 could be salvaged into something that looks more like the human-written version.

```
lazy val example: Parsley[String] = chain.postfix(string("b"))(string("a").as(_ + "a"))
```

Fig. 4.3: An idiomatic way to express the `example` parser using `chain.postfix`.

Chapter 5
Simplifying Parsers and Expressions

At this stage, the left-recursion factoring transformation leaves a lot to be desired in terms of output quality. This chapter takes a step back from linting rules and focuses on ensuring how transformed terms can get pretty-printed in a human-readable form. The following ideas are explored:

- First, §5.1 discusses how parser terms can be simplified via domain-specific optimisations based on parser laws.
- Afterwards, §5.2 discusses how expressions can be partially evaluated to some extent. This is achieved using another intermediate AST, this time based on the λ-calculus, which unlocks the idea of β-reduction and normalisation as tools to reduce the complexity of these terms.

5.1 Simplifying Parsers

Reusing a similar abstract syntax representation as `parsley` itself unlocks some interesting insights for `parsley-garnish`. Gibbons and Wu [2014] note that a deep-embedded DSL consists of two components:

1. A representation of the language's abstract *syntax*, in the form of the aforementioned datatype.
2. Some traversals over the datatype, which gives *semantics* to that syntax.

A deep-embedded DSL and a linter for that DSL can thus be viewed as two different semantic interpretations over the same abstract syntax:

- The DSL semantics are *evaluation*. The syntactic structure may be transformed for optimisation purposes before generating code to be evaluated.
- The linter's semantics are two-fold for lint diagnostics and code rewrites:
  - *Emitting side-effects* in the form of diagnostics, based on patterns of interest within the syntactic structure.
  - *Pretty-printing* a transformation over the syntactic structure, as a rewrite action – crucially, unlike evaluation, the transformed output is not converted into code but rather a textual representation to be rewritten over the original source file. The output of this transformation may benefit from *the same optimisation transformations as with the DSL semantics* to simplify the pretty-printed textual output.

This section shows that this is indeed the case for `parsley-garnish`: the same optimisation transformations apply for both `parsley` (the DSL) and `parsley-garnish` (the linter). The only difference lies in the purpose of performing these transformations:

- `parsley-garnish` needs to perform simplifications on the `Parser` AST to produce output of hand-written quality, or else the resulting parser would be unreadable, as was the case in fig. 4.2.
- `parsley` performs simplifications on its combinator tree to produce output of hand-written quality, in order to deliver excellent parser performance.

5.1.1 Parser Laws

Willis [2024] notes that parser combinators are subject to *parser laws*, which often form a natural simplification in one direction. Both `parsley` Scala [Willis and Wu 2018] and `parsley` Haskell [Willis, Wu, and Pickering 2020; Willis 2024] use these laws as the basis for high-level optimisations to simplify the structure of deeply-embedded parsers. These same principles are used by `parsley-garnish` to simplify parser terms to resemble the natural style that a human would write by hand.

Fig. 5.1 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of these laws have already been shown to hold for Parsley by Willis and Wu [2018]; an additional proof for eq. (5.8) can be found in appendix A.

```
p.map(f).map(g) = p.map(g compose f)                    (5.1)
  pure(f) <*> pure(x) = pure(f(x))                      (5.2)
      pure(f) <*> x = x.map(f)                          (5.3)
            empty | u = u                               (5.4)
            u | empty = u                               (5.5)
        pure(x) | u = pure(x)                           (5.6)
         empty <*> u = empty                            (5.7)
        empty.map(f) = empty                            (5.8)
```

Fig. 5.1: Functor (5.1), Applicative (5.2, 5.3), and Alternative (5.4–5.8) laws.

5.1.1.1 Simplifying the Example Parser

It is useful to illustrate how these laws are used to simplify a parser term, by starting with the parser in fig. 4.2. In the following example, function terms will be greyed out as they are still currently uninspectable – this is work that is addressed in the next section.

First of all, most of the noise in `example` comes from the large number of `empty` combinators. These can be eliminated using eqs. (5.4), (5.5), (5.7), and (5.8):

```
lazy val example: Parsley[String] = chain.postfix(string("b"))(
  (pure(identity).map(compose((_ + _).curried))).map(flip) <*> string("a")
)
```

This already looks a lot better, but the second parameter to `postfix` can be further simplified as follows:

```
    (pure(identity).map(compose((_ + _).curried))).map(flip) <*> string("a")
=       { eqs. (5.2) and (5.3) }
    pure(compose((_ + _).curried)(identity)).map(flip) <*> string("a")
=       { eqs. (5.2) and (5.3) }
    pure(flip(compose((_ + _).curried)(identity))) <*> string("a")
=       { eq. (5.3) }
    string("a").map(flip(compose((_ + _).curried)(identity)))
```

The final simplified form of the parser is then:

```
val f = flip(compose((_ + _).curried)(identity))
lazy val expr: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

The parser is now expressed in a much simplified form, in a similar style to the hand-written example in fig. 4.3! The remaining challenge is to simplify the contents of the expression `f`, which is tackled in §5.2.

5.1.2 Implementing Rewrites on the Parser AST

Lawful simplifications are applied by a bottom-up transformation over the recursively defined `Parser` AST. Since there are many parser cases, this inevitably leads to repetitive and error-prone boilerplate code which simply exists to recursively propagate the transformation through each case. To avoid this, the recursive traversal itself can be decoupled from the definition of the transformation function. Although the traversal is still hand-written, this implementation is inspired by the generic traversal patterns offered by Haskell's `uniplate` library [Mitchell and Runciman 2007].

Partial functions Scala supports *partial functions*, which are functions only defined for a subset of its possible input values. If invoked on an undefined input, this results in a runtime error. A useful idiom utilising partial functions is the `collect` method on collections, which combines functionality of `filter` and `map` in a clean manner. This method will attempt to apply a partial function to each element in the collection, and only include the result if the function is defined. In fact, `collect` has already been used in earlier sections to inspect and gather AST nodes of interest in the Scalameta AST:

```
doc.tree.collect { case Term.Name(name) => name }
//                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A partial function only defined for Term.Name nodes, not on any other type of AST node
```

Rewrite rules on parsers are most naturally expressed as a partial function, since they are only defined for certain parser cases. Therefore, the traversal method `transform` takes a partial function, applying it to nodes where it is defined. The transformation is applied via a bottom-up traversal:

```
def transform(pf: PartialFunction[Parser, Parser]): Parser = {
  val p = this match {
    case p <*> q       => p.transform(pf) <*> q.transform(pf)
    case Zipped(f, ps) => ps.map(_.transform(pf)).zipped(f)
    case Pure(f)       => Pure(f)
    ...
  }
  // Apply the partial function if defined, otherwise return the original parser
  pf.applyOrElse(parser, identity[Parser])
}
```

A `rewrite` method can then be defined in terms of `transform`, applying the partial function everywhere and re-applying it until it no longer makes a change. This has the effect of applying a transformation exhaustively until a normal form is reached.

```
def rewrite(pf: PartialFunction[Parser, Parser]): Parser = {
  def pf0(p: Parser) = if (pf.isDefinedAt(p)) pf(p).rewrite(pf) else p
  this.transform(pf0)
}
```

With this, any transformation on parsers can be defined without having to worry about recursion boilerplate: the act of traversal itself is fully abstracted away and encapsulated within the `transform` method. Using `rewrite`, parser simplification can then be expressed in a clean and maintainable manner:

```
def simplify: Parser = this.rewrite {
  case FMap(FMap(p, f), g) => FMap(p, compose(g, f))
  case Pure(f) <*> Pure(x) => Pure(App(f, x))
  case u <|> Empty         => u
  case Pure(f) <|> _       => Pure(f)
  ...
}
```

Extensibility and Safety Further design considerations are made to ensure the extensibility of this approach: the `Parser` trait is sealed, which enables compiler warnings if a new `Parser` case is added and the `transform` method is not updated. Although this formulation of the traversal is inspired by generic traversals, it still manually defines the traversal for each case: a safer approach would be to generically derive this. In Scala, this would require the use of an external dependency such as `shapeless`[1], which is overkill given the relative simplicity of the `Parser` ADT.

[1] https://github.com/milessabin/shapeless

5.1.3 Discussion

The design of the parser simplification process for `parsley-garnish` was not intended to closely follow the methods used in `parsley`, so it is remarkable that the two approaches have ended up being so similar. However, in retrospect, this resemblance is not surprising given that the act of parser simplification and optimisation are fundamentally the same transformation. Since both `parsley` and `parsley-garnish` represent the parser AST as a deep-embedded structure, it is natural that this transformation is implemented similarly in both cases as a bottom-up traversal over the abstract syntax.

This insight can be extended to any deep embedded DSL based on an algebra, where constructs within the DSL are subject to algebraic laws and operations. It would be interesting to see an EDSL and linter pair that shares a unified data structure for its abstract syntax, in order to take full advantage of this duality.

5.2 Representing and Normalising Expressions

The previous section demonstrated the process of simplifying the `Parser` AST, but this is not the only syntactic structure that requires simplification. So far, parsers such as `pure` and `map` still treat expressions as black boxes in the form of raw `scala.meta.Term` AST nodes. This is evident from where the example in §5.1.1.1 left off, where the parser itself is in a simplified form, but the function passed to `map` isn't:

```
val f = flip(compose((_ + _).curried)(identity))
```

Therefore, this section explores the following:

- How expressions can be represented as another intermediate AST, so that they are statically inspectable enough to be simplified.
- The notion of *normalisation*, reducing expressions into a semantically equivalent but syntactically simpler form.

5.2.1 The *n*-ary Lambda Calculus

Once again, the complexity of manipulating the generic Scalameta AST can be avoided by building a new intermediate AST representation for expression terms.

Scala, as a functional programming language, uses an extension of the λ-calculus [Church 1936] as its theoretical foundations [Cremet et al. 2006; Amin et al. 2016]. The expression terms present within parsers are equivalent to λ-terms, just with extra syntactic sugar. In the standard λ-calculus, each function only takes one argument, and multi-argument functions are represented as a chain of single-argument functions: this is known as *currying*. Scala supports curried functions using multiple parameter lists, but uncurried functions are preferred for performance reasons. Since these functions will be transformed from Scala code and back, it is desirable to maintain a high-level equivalence between these two representations. Thus, the expression AST will be based on fig. 5.2, which extends the λ-calculus to support proper multi-argument functions using n-ary abstraction and application.

$$
\begin{array}{rll}
M, N ::= & x & \text{variable} \\
\mid & (\lambda \overline{\mathbf{x}}.\ M) & n\text{-ary abstraction, where } \overline{\mathbf{x}} = (x_1, \ldots, x_n) \\
\mid & (M\ \overline{\mathbf{N}}) & n\text{-ary application, wher } \overline{\mathbf{N}} = (N_1, \ldots, N_n)
\end{array}
$$

Fig. 5.2: Syntax for the untyped λ-calculus extended with n-ary abstraction and application.

5.2.1.1 β-Reduction and α-Conversion

In the λ-calculus, terms are evaluated via β-reduction: fig. 5.3 shows how this can be defined for the n-ary λ-calculus. Unlike the standard λ-calculus, reduction will only take place if the expected number of arguments in $\overline{\mathbf{x}}$ are equal to the number of arguments in $\overline{\mathbf{N}}$; otherwise, evaluation is stuck.

$$(\lambda\overline{\mathbf{x}}.\ M)\ \overline{\mathbf{N}} \to_\beta M[\overline{\mathbf{N}}/\overline{\mathbf{x}}] \qquad (\text{if } |\overline{\mathbf{x}}| = |\overline{\mathbf{N}}|)$$

Fig. 5.3: The β-reduction rule for the n-ary lambda calculus.

The syntax $M[N/x]$ denotes term substitution, where all free occurrences of x in M are replaced with N. Substitution must avoid *variable capture*, when N contains free variables that are bound in the scope where x is found [Bakel 2022]. Avoiding capture is achieved by performing α-conversion, which is the process of renaming bound variables. In the λ-calculus, two terms are considered α-equivalent if they can be transformed into each other by only renaming bound variables: the term $\lambda x.x$ is equivalent to $\lambda y.y$.

Illustrating variable capture For example, substitution without α-conversion incorrectly β-reduces the following term:

$$\begin{aligned}(\lambda x.\lambda y.xy)y &\to_\beta (\lambda y.xy)\ [y/x] \\ &= \lambda y.yy\end{aligned}$$

The y that was substituted was originally a free variable, distinct from the y bound in the lambda $\lambda y.xy$. However, after substitution, it became captured under the lambda, where the two y terms are now indistinguishable in the incorrect expression $\lambda y.yy$. The correct β-reduction with capture-avoiding substitution would instead proceed as follows:

$$\begin{aligned}(\lambda x.\lambda y.xy)y &\to_\beta (\lambda y.xy)\ [y/x] \\ &=_\alpha (\lambda z.xz)\ [y/x] \\ &= \lambda z.yz\end{aligned}$$

5.2.1.2 Simplifying the Example Expression

The example from the beginning of the section can thus be evaluated by hand via β-reduction, representing the higher-order functions as λ-abstractions:

$$\begin{aligned}\texttt{flip(compose((\_ + \_).curried)(identity))} &= \text{flip (compose } (\lambda a.\lambda b.a + b) \text{ identity)} \\ &= \text{flip } ((\lambda f.\lambda g.\lambda x.f\,(g\ x))(\lambda a.\lambda b.\ a + b)(\lambda x.x)) \\ &\to_{\beta*} \text{flip } (\lambda g.\lambda x.\ (\lambda b.\ g\ x + b)(\lambda x.x)) \\ &\to_{\beta*} \text{flip } (\lambda x.\lambda b.\ x + b) \\ &= (\lambda f.\lambda x.\lambda y.\ f\ y\ x)(\lambda x.\lambda b.\ x + b) \\ &\to_{\beta*} \lambda x.\lambda y.\ y + x \\ &= (\mathsf{x}, \mathsf{y}) \Rightarrow \mathsf{y} + \mathsf{x}\end{aligned}$$

This normalised expression has the same meaning as the original, but is now expressed in a clearer form! This is much more suitable to be placed back into the code rewrite, as it is easier to understand by the user. The rest of the section now explores how this process can be implemented in `parsley-garnish`.

5.2.2 Representing Names

There are a plethora of approaches to implementing the λ-calculus, mostly differing in how they represent variable names. This affects how variable capture is handled, and also how α-equivalence of two terms can be determined. For `parsley-garnish`, cheap α-equivalence is desirable to help check equivalence of parser terms, which is useful for some transformations.

Naïve capture-avoiding substitution Representing variable names as strings is the most straightforward approach in terms of understandability. The example below shows how the simply typed λ-calculus can be represented as a generalised algebraic data type (GADT) [Cheney and Hinze 2003] in Scala:

```
type VarName = String

trait Lambda
case class Abs[A, B](x: Var[A], f: Lambda[B]) extends Lambda[A => B]
case class App[A, B](f: Lambda[A => B], x: Lambda[A]) extends Lambda[B]
case class Var[A](name: VarName) extends Lambda[A]

// λf. λx. f x
val f = Var("f")
val x = Var("x")
val expr = Abs(f, Abs(x, App(f, x)))
```

Although naïvely substituting these terms seems logically simple, it can be very tricky to get right. This approach requires calculating the free variables in a scope before performing substitution, renaming bound variables if it would lead to variable capture. Due to the inefficiency of having to traverse the whole term tree multiple times, this approach is not used in any real implementation of the λ-calculus. Furthermore, checking α-equivalence is also tedious, requiring another full traversal of the term tree to compare variable names.

Barendregt's convention Renaming all bound variables to be unique satisfies *Barendregt's convention* [Barendregt 1984], which removes the need to check for variable capture during substitution. However, to maintain this invariant, variables must also be renamed during substitution – this administrative renaming incurs a relatively high performance overhead and chews through a scarily large number of fresh variable names. The approach has been successfully optimised to very impressive performance, though: the Haskell GHC compiler uses Barendregt's convention with a technique dubbed "the Rapier" [Peyton Jones and Marlow 2002], maintaining further invariants to avoid renaming on substitution when unnecessary. Unfortunately, maintaining the invariants to keep this transformation correct becomes very difficult [Maclaurin, Radul, and Paszke 2023].

Nameless and hybrid representations Nameless representations like *De Bruijn indices* [de Bruijn 1972] eschew names entirely, instead representing variables as the number of binders between the variable and its binding site. For example, $\lambda x.x(\lambda y.xy)$ would be represented as $\lambda.0(\lambda.1\ 0)$: the variable y is replaced with 0 to signify that y is bound by the closest λ binder. The first occurrence of x is replaced with 0 in the same way, but its second occurrence is replaced with 1 since it is bound by the λ two levels up.

An advantage of De Bruijn indexing is that it makes α-equivalence trivial to check, as it is just a matter of comparing the indices. However, although an elegant representation, De Bruijn terms are notoriously difficult to work with as they are not easily human-readable. Furthermore, performing substitutions with De Bruijn terms has an overhead as variable positions have to be shifted – this is undesirable given that the purpose of the AST is to normalise λ-terms. To avoid this, hybrid representations combining named and nameless representations exist [McBride and McKinna 2004; Charguéraud 2012], but they become rather complex solutions for what should be a relatively simple λ-calculus implementation for `parsley-garnish`'s needs.

Higher-order abstract syntax Using *higher-order abstract syntax* (HOAS) [Pfenning and Elliott 1988] sidesteps variable binders entirely by borrowing substitution from the meta-language, making it the meta-language's responsibility to handle variable capture instead. In contrast, the previous techniques were examples of first-order abstract syntax, which represents variables and unknowns with identifiers (whether with names or indices). A HOAS approach does not name bound variables, instead representing them as bindings in the meta-language:

```
trait HOAS
case class Abs[A, B](f: HOAS[A] => HOAS[B]) extends HOAS[A => B]
case class App[A, B](f: HOAS[A => B], x: HOAS[A]) extends HOAS[B]

// λf. λx. f x
val expr = Abs(f => Abs(x => App(f, x)))
```

Therefore, this representation performs substitution through Scala's function application, which makes it extremely fast compared to the other approaches. However, since lambda abstractions are represented as closures within Scala itself, the function body becomes wrapped under Scala's variable bindings, making them difficult to inspect and work with.

5.2.3 Normalisation Strategies

One remaining hurdle stands before deciding on an ADT representation: how normalisation will be implemented. The ideas of partial evaluation and normalisation are related concepts – it is useful to view normalisation as statically evaluating as many terms as possible, but since not all terms have known values, the expression cannot be fully evaluated to a result value. Normalisation can thus be viewed simply as a process of evaluation, but in the presence of unknown terms. This section briefly explains the traditional notion of reduction-based normalisation, before introducing normalisation by evaluation as a more elegant and efficient strategy.

5.2.3.1 Reduction-Based Normalisation

The β-reduction rule is a *directed* notion of reduction, which can be implemented as a syntax-directed term-rewriting system, in a similar way to how `Parser` terms are simplified. The goal is to achieve beta normal form (β-NF) by allowing β-reduction to occur deep inside λ-terms, in all redexes of a term, until no more reductions can be made.

5.2.3.2 Normalisation by Evaluation

An interesting alternative strategy stems from a notion of *reduction-free* normalisation, based on an undirected notion of term equivalence, rather than directed reduction. *Normalisation by Evaluation* (NBE) [Filinski and Korsholm Rohde 2004] achieves this by *evaluating* syntactical terms into a semantic model, then *reifying* them back into the syntactic domain. The denotational model (denoted by $[\![-]\!]$) generally involves implementing a separate datatype from the syntactic AST representation of functions. The semantics is specifically constructed to be *residualising*, meaning that terms can be extracted out into the original syntactic representation. Normalisation is then just defined as the composition of these two operations, as illustrated in fig. 5.4.

$$\text{Syntactic domain} \underset{\textit{reify}}{\overset{[\![-]\!]}{\rightleftarrows}} \text{Semantic domain}$$

$$\textit{normalise} = \textit{reify} \circ [\![-]\!]$$

Fig. 5.4: Normalisation by evaluation in a semantic model.

5.2.4 The Expression ADT

The final implementation of the `Expr` AST normalises terms with NBE, which results in a two-tiered representation of expression terms:

1. Scalameta AST nodes corresponding to expressions are lifted to the `Expr` ADT, which represents the syntax of lambda expressions using a simple named approach.
2. `Sem` uses HOAS to leverage Scala semantics as the denotational model for lambda expressions. During normalisation, `Expr` terms are evaluated into `Sem`, then reified back into `Expr`.

This achieves the following desired properties for `parsley-garnish`'s use cases:

- The syntactic `Expr` ADT is represented in a simple manner, which is easy to construct and manipulate as opposed to a HOAS representation. This allows function terms to be pattern matched on, as part of parser simplifications.
- Lifting the syntactic constructs to Scala semantics with HOAS unlocks extremely efficient normalisation, and easier guarantees of correctness with respect to variable capture.
- Reifying `Sem` terms back into syntactic `Expr` terms automatically α-converts names, granting α-equivalence for free.

Fig. 5.5a shows the implementation of the untyped `Expr` ADT representing the abstract syntax of n-ary λ-terms, extended with the following:

- Optional explicit type annotations for variables – these are not used for type-checking, but are there to preserve Scala type annotations originally written by the user.
- `Translucent` terms to encapsulate open terms holding a `scala.meta.Term` which cannot be normalised further. These carry an environment of variable bindings to substitute back in during pretty-printing – in a metaprogramming context, this is analogous to splicing into a quoted expression.

This structure is largely mirrored by the HOAS-based `Sem` ADT shown in fig. 5.5b, which allows it to be reified back into `Expr` terms.

Constructing higher-order functions `Expr` also implements some helper objects to make it more convenient to construct and deconstruct single-parameter abstractions and applications:

```
object Abs {
  // Convenience factory method to create a single-parameter abstraction
  def apply(x: Var, f: Expr) = AbsN(List(x), f)
  // Extractor allows user to pattern match on Abs(x, f)
  // instead of the more tedious AbsN(List(x), f)
  def unapply(func: AbsN): Option[(Var, Expr)] = func match {
    case AbsN(List(x), f) => Some((x, f))
    case _ => None
  }
}

object App {
  // Convenience factory method to apply a single argument to a function
  def apply(f: Expr, x: Expr) = AppN(f, List(x))
  // Apply multiple arguments in sequence, as f(x)(y)(z) instead of f(x, y, z)
  def apply(f: Expr, xs: Expr*) = xs.foldLeft(f)(App(_, _))
}
```

Using these objects, fig. 5.6 shows how the higher-order functions necessary for left-recursion factoring can be implemented as constructors for `Expr` terms.

```
trait Expr
case class AbsN(xs: List[Var], f: Expr) extends Expr
case class AppN(f: Expr, xs: List[Expr]) extends Expr
case class Var(name: VarName, displayType: Option[scala.meta.Type]) extends Expr
case class Translucent(t: Term, env: Map[VarName, Expr]) extends Expr
```

(a) The `Expr` ADT for representing the abstract syntax of lambda expressions.

```
trait Sem
case class Abs(paramTypes: List[Option[scala.meta.Type]], f: List[Sem] => Sem) extends Sem
case class App(f: Sem, xs: List[Sem]) extends Sem
case class Var(name: VarName, displayType: Option[scala.meta.Type]) extends Sem
case class Translucent(t: Term, env: Map[VarName, Sem]) extends Sem
```

(b) The `Sem` ADT for representing the residualising semantics of lambda expressions.

Fig. 5.5: The intermediate AST for expressions.

```
/* id : A => A */
def id: Expr = {
  val x = Var.fresh()
  Abs(x, x)
}

/* flip : (A => B => C) => B => A => C */
def flip: Expr = {
  val (f, x, y) = (Var.fresh(), Var.fresh(), Var.fresh())
  Abs(f, Abs(x, Abs(y, App(f, y, x)))) // λf. λx. λy. f y x
}

/* compose : (B => C) => (A => B) => A => C */
def compose: Expr = {
  val (f, g, x) = (Var.fresh(), Var.fresh(), Var.fresh())
  Abs(f, Abs(g, Abs(x, App(f, App(g, x))))) // λf. λg. λx. f (g x)
}
def compose(f: Expr) = App(compose, f)
def compose(f: Expr, g: Expr) = App(compose, f, g)
```

Fig. 5.6: Constructors for higher-order functions represented as λ-expressions in `Expr`.

Improved type safety The originally intended design was to represent `Expr` as a type-parameterised GADT for improved type safety, where it would be based on a *typed* variant of the λ-calculus. This would have also allowed `Parser` to be represented as a GADT parameterised by the result type of the parser. However, attempting to implement this ran into two main hurdles:

- `Var` and `Translucent` terms would need to be created with concrete type parameters of their inferred types. Scalafix's semantic API is not powerful enough to guarantee that all terms can be queried for their inferred types – in fact, the built-in Scalafix rule *Explicit Result Types* calls the Scala 2 presentation compiler to extract information like this[2]. This solution is complex and brittle due to its reliance on unstable compiler internals, which undermines Scalafix's goal of being a cross-compatible, higher-level abstraction over compiler details.
- Scala 2's type inference for GADTs is less than ideal, requiring extra type annotations and unsafe casts which ultimately defeat the original purpose of type safety. This situation is improved, although not completely solved, in Dotty [Parreaux, Boruch-Gruszecki, and Giarrusso 2019] – but Scalafix does not yet support Scala 3.

5.2.4.1 Evaluating Performance of Normalisation Strategies

`parsley-garnish` originally used a named approach with Barendregt's convention, generating fresh variable names using an atomic counter. However, this required an extra α-conversion pass to clean up variable names before pretty-printing the term, since the fresh variable names were very ugly.

5.2.5 Lifting to the Intermediate Expression AST

The `Parser` AST is amended to take `Expr` arguments where they used to take `scala.meta.Term` values. Take the `Pure` parser as an example:

```
case class Pure(x: Expr) extends Parser
object Pure {
  def fromTerm: PartialFunction[Term, Pure] = {
    case Term.Apply(matcher(_), Term.ArgClause(List(func), _)) => Pure(func.toExpr)
  }
}
```

The `toExpr` extension method on `scala.meta.Term` is used to lift `Term` AST nodes to `Expr` terms. Expression lifting is invoked whenever a parser expects an expression (whether a function or simple value) as an argument. This section gives a high-level overview of the three cases that `toExpr` handles.

5.2.5.1 Lambda Expressions

Writing parsers often involves defining simple lambda expressions used to glue together parsers, or to transform the result of a parser, as so:

```
val asciiCode: Parsley[Int] = item.map(char => char.toInt)
```

These lambda expressions are represented in the Scalameta AST as `Term.Function` nodes, which are recursively traversed to collect all parameter lists. This is folded into a chain of n-ary abstractions, with the final term being the body of the lambda, which is wrapped into a `Translucent` term.

To ensure that the parameter names in the `Translucent` body term are unique, the parameters are α-converted to fresh names. The body is also transformed to make sure references to these bound variables use their new names: this conversion is well-scoped as it compares terms using their unique Scalameta symbols. The following example illustrates when this is necessary:

```
a => (a, b) => a + b
```

[2] https://github.com/scalacenter/scalafix/issues/1583

Although no sane Scala programmer would write this, this lambda demonstrates how it is possible to shadow variables – the a in the function body refers only to the a in the second parameter list, as it shadows the a in the first parameter list. The lifted `Expr` term would then resemble the following λ-calculus expression, where **bold** values correspond to `scala.meta.Term` nodes as opposed to `Expr` values:

```
λ(x1). λ(x2, x3). Translucent(x2 + x3, env = {x1 → x1, x2 → x2, x3 → x3})
```

This shows how the lambda body's environment maps `Term.Name` nodes to their corresponding variable terms. When the term is pretty-printed, each `Term.Name` node is replaced with their corresponding `Expr` term – this is analogous to the splicing operation on quasiquotes:

```
q"x1 => (x2, x3) => $x2 + $x3"
```

5.2.5.2 Placeholder Syntax

Scala supports a placeholder syntax using underscores to make lambda expressions more concise, so the earlier parser can be rewritten as:

```
val asciiCode: Parsley[Int] = item.map(_.toInt)
```

Scalameta differentiates between regular lambda expressions and those using placeholder syntax, representing the latter as `Term.AnonymousFunction` nodes. This makes it easy to identify which approach to be taken during conversion. To convert this case, each successive underscore in the expression body is replaced with a fresh variable name. Placeholder syntax creates a fully uncurried function with a single parameter list[3]. Therefore, the converted `Expr` term is always a single n-ary abstraction, where the arguments are the freshly generated variable names in order of their occurrence in the expression body.

5.2.5.3 Eta-Expansion

If the term is not a lambda expression, `parsley-garnish` attempts to η-expand the term if possible. For example, an idiomatic parser written using the *Parser Bridges* pattern [Willis and Wu 2022] could resemble the following:

```
case class AsciiCode(code: Int)
object AsciiCode extends ParserBridge1[Char, AsciiCode] {
  def apply(char: Char): AsciiCode = AsciiCode(char.toInt)
}
val asciiCode = AsciiCode(item)
```

When `parsley-garnish` converts `asciiCode` to a `Parser`, it desugars the bridge constructor into something resembling `item.map(AsciiCode.apply)`. The η-expanded form of `AsciiCode.apply` would be as follows:

```
(char: Char) => AsciiCode.apply(char)
```

To η-expand `scala.meta.Term` nodes, `parsley-garnish` attempts to look up the method signature of its symbol using Scalafix's semantic API. This is not always possible – in that case, the term can't be statically inspected any further and is just wrapped in a `Translucent` term.

5.2.6 Normalising Expression Terms

Using NBE, normalisation therefore follows a two-step process: `Expr` values `evaluate` into `Sem` values, which are then `reify`ed back into `Expr`:

```
trait Expr {
  def normalise: Expr = this.evaluate.reify
}
```

[3] https://www.scala-lang.org/files/archive/spec/2.13/06-expressions.html#anonymous-functions

Evaluation Evaluation proceeds by carrying an environment mapping bound variables to their semantic representations. Evaluating a variable looks up its name in the environment, while evaluating a lambda abstraction produces a closure using the current environment – using HOAS allows these closures to be represented as native Scala closures. The interesting case is evaluating function application: this allows β-reduction within the *semantic domain* at any point within the term, not just on the head term. The function and its arguments are first evaluated separately – then, if the function evaluates to an abstraction, the arguments are passed to the Scala closure `g: List[Sem] => Sem`, collapsing the term structure by one step.

```
trait Expr {
  def evaluate: Sem = {
    def eval(func: Expr, boundVars: Map[Var, Sem]): Sem = func match {
      case v @ Var(name, displayType) =>
        boundVars.getOrElse(v, Sem.Var(name, displayType))
      case AbsN(xs, f) =>
        Sem.Abs(xs.map(_.displayType), vs => eval(f, boundVars ++ xs.zip(vs)))
      case AppN(f, xs) => eval(f, boundVars) match {
        case Sem.Abs(_, g) => g(xs.map(eval(_, boundVars)))
        case g => Sem.App(g, xs.map(eval(_, boundVars)))
      }
      case Translucent(term, env) =>
        Sem.Translucent(term, env.mapValues(eval(_, boundVars)))
    }

    eval(this, Map.empty)
  }
}
```

Reification The NBE semantics utilised by `parsley-garnish` are *intensional* [Lindley 2005], meaning that once syntactic terms are fully evaluated into their semantics, the expression is normalised to β-NF. Reification is then a simple process of converting each level of the term back into its syntactic counterpart. When a lambda abstraction is reified, bound variables are assigned names from a fresh name supply. This step is what grants α-equivalence for free, as the fresh name generator can be made deterministic: given two terms that evaluate to the same semantic structure, reifying both will yield syntactic representations with the same names.

```
trait Sem {
  def reify: Expr = {
    def reify0(func: Sem)(implicit freshSupply: Fresh): Expr = func match {
      case Abs(tpes, f) =>
        val params = tpes.map(Expr.Var(freshSupply.next(), _))
        Expr.AbsN(params, reify0(
          f(params.map { case Expr.Var(name, tpe) => Sem.Var(name, tpe) } )
        ))
      case App(f, xs) => Expr.AppN(reify0(f), xs.map(reify0))
      case Translucent(t, env) => Expr.Translucent(t, env.mapValues(reify0))
      case Var(name, displayType) => Expr.Var(name, displayType)
    }

    reify0(this)(new Fresh)
  }
}
```

5.2.7 Lowering Back to the Scalameta AST

Once any transformations on the `Expr` terms are complete, they need to be converted back to strings to be placed in a Scalafix patch. This is achieved in the same way as parsers in §4.2.3, by lowering them back as `scala.meta.Term` ast nodes and re-using Scalameta's pretty-printer to generate a syntactically well-formed string representation. These can also be achieved using quasiquotes:

- Lambda abstractions are transformed into a lambda expression of form `q"(...params) => body"`.
- Function application is transformed into method calls.
- Variables are simply `Term.Name` nodes with their syntactic names.
- Translucent terms splice their environment bindings back into their term body.

Discussion

`parsley` Haskell, as a *staged* parser combinator library, also has the ability to inspect and optimise the code of user-defined functions. The approach taken by `parsley-garnish` and parsley share many similarities, both using the λ-calculus as a core language to normalise expressions. In both cases, the need to reduce expression terms is motivated by how parser simplifications involve fusion, which results in function applications that can be partially evaluated.

However, the two have different motivations and requirements for normalising expressions, so their approaches differ in some ways – fig. 5.7 illustrates these differences.

Syntactic representation Unlike `parsley-garnish`, `parsley` has a two-level syntactic representation for expressions. `Defunc` is akin to a deep embedding of higher-order functions, representing them as a GADT: this process is known as *defunctionalisation* [Reynolds 1972; Danvy and Nielsen 2001]. This helps facilitate certain parser law optimisations which require pattern matching on functions as well as parsers, for example:

```
pure(identity) <*> u = u
```

After this step, `Defunc` values are then brought into the lower-level λ-calculus representation `Lambda`, to be normalised by β-reduction.

At the moment, `parsley-garnish` does not have a need to implement any parser simplifications based on these laws, although this may change in the future. Adding an extra defunctionalised layer to the expression AST would be fairly straightforward.

Normalisation strategy `parsley` normalises terms to full $\eta\beta$-NF, whereas `parsley-garnish` only normalises to β-NF. This is because η-reduction in Scala 2 is not as straightforward as in Haskell, and is not always possible – in most cases the appropriate reduction is instead to convert lambdas to placeholder syntax. This is left as future work.

In `parsley`, normalisation is implemented as a reduction-based approach over the HOAS `Lambda` datatype. Normalisation by β-reduction with Haskell function application brings this to β-WHNF. Then, code generation brings this further to β-NF as desired, as well as an extra step for η-reduction to put the term into full $\eta\beta$-NF.

The main reason why `parsley-garnish` takes a different normalisation approach is because unlike `parsley`, there is still a need for α-equivalence checking after normalisation. In `parsley`, the normalised forms are immediately utilised for code generation, so they can be kept as HOAS the entire time, without representing variables with any names. Conversely, in `parsley-garnish`, these normalised terms undergo further analysis before being transformed into code patches for pretty-printing.

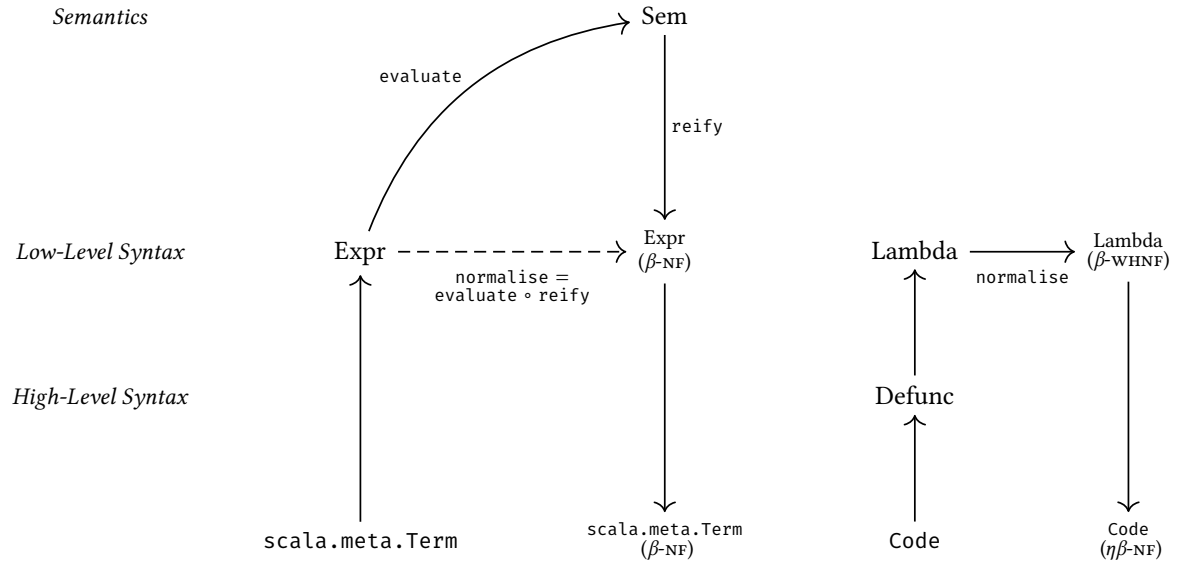


Fig. 5.7: Comparison of expression normalisation in `parsley-garnish` (left) and `parsley` Haskell (right).

Summary

This chapter introduced the idea of simplifying parsers and normalising expressions, by representing both as intermediate ASTs to improve their static inspectability. It also demonstrated how these processes are related to the optimisation techniques used in both `parsley` Scala and `parsley` Haskell.

With promising results applying these simplifications on the `example` parser from last chapter, the improved `Parser` AST now unlocks the potential for more powerful and interesting transformations utilising specialised domain knowledge of parser combinators.

Chapter 6
Using the New Parser ast

The ability to simplify parsers, as well as normalising their expression arguments, makes it feasible to use the intermediate `Parser` ast for high-level parser transformations.

6.1 Removing Left-Recursion: Revisited

Chapter 4 left off with a disappointing result for the left-recursion factorisation transformation, producing a mostly unreadable output in fig. 4.2. This transformation can now be revisited given the improvements made to the `Parser` ast. The following changes can be made to improve the rule's output:

- Parser terms should be simplified via parser laws to get a more readable final output.
- Parsers should take advantage of the new `Expr` representation to allow for partial evaluation of higher-order functions, rather than manipulating terms as opaque quasiquotes.
- The higher-order functions `flip` and `compose` can now be represented as `Expr` lambda expressions, eliminating the need to patch in their definitions as previously required in §4.3.3.

Define normalise and prettify, attempt to resugar decomposed parsers back into a nicer form.

```
trait Parser {
  def normalise: Parser = this.simplify.normaliseExprs
  def isEquivalent(other: Parser) = this.normalise == other.normalise

  def prettify: Parser = this.normalise.resugar
  def resugar: Parser = this.rewrite {
    // p.map(\x -> \y -> y) <*> q == p ~> q
    case FMap(p, Abs(_, Abs(Var(y, _), Var(z, _)))) <*> q if (y == z) => p ~> q
    // p.map(\x -> \y -> x) <*> q == p <~ q
    case FMap(p, Abs(Var(x, _), Abs(_, Var(z, _)))) <*> q if (x == z) => p <~ q

    // f.curried.map(p) <*> q == (p, q).zipped(f)
    case FMap(p1, Abs(x1, Abs(x2, body))) <*> p2 =>
      Zipped(AbsN(List(x1, x2), body), List(p1, p2))

  }.transform {
    // Scala 2 cannot resolve implicit stringLifts in some positions
    case FMap(Str(s, _), f) => FMap(Str(s, implicitSyntax = false), f)
    case Zipped(f, Str(s, _) :: ps) => Zipped(f, Str(s, implicitSyntax = false) :: ps)
  }
}
```

No .curried

```
trait Expr {
  def curried: Expr = this match {
    case Abs(x, f)        => Abs(x, f.curried)
    case AbsN(x :: xs, f) => Abs(x, AbsN(xs, f).curried)
    case _                => this
  }
}
```

```
trait LiftLike extends Parser {
  def unfold() = {
    val curriedFunc = Pure(func match {
      // If the Expr is still opaque, it is treated as before
      case Translucent(term, env) => Translucent(q"($term).curried", env)
      // If statically inspectable, curry the lambda expr properly
      case _ => func.curried
    })

    parsers.foldLeft(curriedFunc)(_ <*> _).unfold
  }
}
```

* no .curried * Resugaring * Optimisation: normalise leftrec part to see if it comes to empty – if so, don't inline in the NT case

```
def transform(results: Option[Expr], nonLeftRec: Parser, leftRec: Parser) = {
  val result = results match {
    case None    => Empty
    case Some(t) => Pure(t)
  }

  leftRec.normalise match {
    case Empty   => Patch.empty // not left-recursive, don't rewrite
    case Pure(_) => Patch.lint(LeftRecDerivesEmptyLint) // left-recursive but unfixable
    case _ =>
      val transformed = Postfix(nonLeftRec | results, leftRec).prettify
      Patch.replaceTree(originalTree, transformed.toString)
  }
}
```

The Final Result

```
lazy val example: Parsley[String] =
  chain.postfix[String](string("b"))(string("a").map(x1 => x2 => x2 + x1))
```

6.2 Simplify Parser

The improved `Parser` AST also gives a new auto-fix rule for free: automatic simplification of parsers.

Idea * For each parser, simplify it via parser laws, and compare to its original. * If the result is different, apply the simplified version as a patch.

The implementation of the entire rule is only 20 lines long:

```
class SimplifyParser extends SemanticRule("SimplifyParser") {
  override def fix(implicit doc: SemanticDocument): Patch = {
    getAllParserDefns.map { case ParserDefinition(_, parser, _, originalTree) =>
      val simplifiedParser = parser.prettify
      if (parser.normaliseExprs != simplifiedParser) {
        val simplifiedParserTerm = simplifiedParser.term.syntax
        Patch.replaceTree(originalTree, simplifiedParserTerm)
      } else {
        Patch.empty
      }
    }.asPatch
```

```
  }
}
```

* Apply parser laws, re-using Parser and Func representations to do cool things prettify = resugar . normaliseFunctions . simplify

6.3 Avoid Parser Redefinition

Similar in spirit to the previous rule * Catch cases when user manually writes out a parser that is already defined in the library

Section 5.1.2 makes it easy to write syntax-directed rewrite rules on parsers. Less annoying than working with scalameta ast directly better design patterns idk, symbolmatcher etc all in one place defined as a trait for parser

so: Looking for dumb definitions of things e.g. endBy(p, sep) implemented as many(p <* sep); or count implemented as a foldLeft, fold fusion

Chapter 7
Evaluation

* Project ended up requiring a lot more work on internal machinery i.e. intermediate ASTs in order to get high-level parser transformations to work * This is good because it makes it allows parsley-garnish to be easily extended with more domain-specific rules in the future * But it also meant that the project was more complex than initially anticipated, and that the variety of implemented linting rules were not as comprehensive as I would have liked * Left-recursion transformation is the big one which motivated much of the work on the intermediate ASTs in the first place, and therefore uses it the most * Evaluating the outputs of the left-recursion transformation therefore also evaluates the success of the intermediate machinery

7.1 Removing Left-Recursion

Broadly speaking, there are three classes of left-recursion: direct, indirect, and hidden left-recursion. This section evaluates `parsley-garnish`'s ability to handle each of these cases, based on a mostly qualitative set of evaluation criteria:

- Was the instance of left-recursion detected?
- If an auto-fix was performed, was it correct?
- How clear was the output? How does it compare to an idiomatic, manually fixed version?
- Does the output compile?

The following examples assume the existence of a `number` parser, defined the same way as earlier in §2.3:

```
val number: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)
```

7.1.1 Direct Left-Recursion

Direct left-recursion is the simplest and most obvious form of left-recursion, where a parser directly refers to itself in its definition. Thus, it is the easiest form of left-recursion to detect and handle, regardless of the transformation technique used. This section evaluates `parsley-garnish`'s handling of direct left-recursion in a few different scenarios.

7.1.1.1 Unary Postfix Operator

The following minimal grammar for a postfix unary incrementing operator is directly left-recursive:

$$\langle inc \rangle ::= \langle inc \rangle \text{ `+'} \mid \langle number \rangle$$

A parser for this grammar, written in left-recursive form, would be as follows:

```
enum Expr {
  case Num(n: Int)
  case Inc(x: Expr)
}
val incs: Parsley[Expr] = Inc.lift(incs) <~ "+" | Num.lift(number)
```

In this instance, `parsley-garnish` detects the left-recursion and transforms the parser into the form shown below; this is compared with a hand-written version.

```
// Transformed by parsley-garnish
val incs = chain.postfix[Expr](number.map(x1 => Num(x1)))
                              (string("+").map(x1 => x2 => Inc(x2)))
// "Optimal" version written by hand
val incsByHand = chain.postfix[Expr](number.map(Num(_)))("+" as Inc)
```

The output from parsley-garnish manages to compile and is clear and idiomatic, although the hand-written version is slightly more concise. These two versions can be shown to be equivalent via equational reasoning – the first argument to postfix is obviously the same, but with placeholder syntax instead of using explicit lambda arguments. The second argument can be derived as follows:

```
string("+").map(x1 => x2 => Inc(x2))
 =     { Definition of as: p.as(x) = p.map(_ => x) }
string("+") as (x2 => Inc(x2))
 =     { η-reduction on Inc }
string("+") as Inc
 =     { Re-introduce Implicit Conversions pattern }
"+" as Inc
```

This example highlights some subtle points that parsley-garnish considers in order to improve the likelihood of producing compilable output:

- The type ascription chain.postfix[Expr] is not always necessary, although in this case it is actually required to help Scala correctly unify the types expected by the combinator. Since parsley-garnish cannot typecheck its outputs, it always includes this type ascription to boost Scala's type inference.
- The original parser used the *Implicit Conversions* pattern to elide the string combinator, but parsley-garnish re-introduces the explicit string combinator in its output. This is intentional – Scala 2 has trouble with implicit conversions in certain positions, such as in this case where "+".map(...) would not compile. parsley-garnish attempts to add the explicit combinator back in cases like this, although in general it will respect the original style the parser was written in.

7.1.1.2 Arithmetic Expression Language

As a larger-scale example, the left-associative arithmetic operators from §2.3 are also defined in a directly left-recursive manner. Recall how the hand-written version using chain.left1 was presented:

```
lazy val expr: Parsley[Expr] = chain.left1(term)('+' as Add(_, _) | '-' as Sub(_, _))
```

Compared with the full transformed output from parsley-garnish:

```
lazy val expr = chain.postfix[Expr](term)(
  ('+' ~> term).map(x1 => x2 => Add(x2, x1)) | ('-' ~> term).map(x1 => x2 => Sub(x2, x1)))
lazy val term = chain.postfix[Expr](atom)(
  ('*' ~> atom).map(x1 => x2 => Mul(x2, x1)) | ('/' ~> atom).map(x1 => x2 => Div(x2, x1)))
lazy val atom = '(' ~> expr <~ ')' | number.map(Num(_))
```

This also manages to successfully compile, and is relatively clear to read. However, it does highlight a current shortcoming of the left-recursion rule: left-recursive parsers can only be transformed into the most generalised postfix form, which may not always be the most optimal choice of combinator. In this case, the chain.left1 combinator would've been a more appropriate choice; even better would be the precedence combinator, since the grammar forms multiple layers of expression operators.

It is theoretically possible to automatically transform a postfix parser into one of its more specialised brethren, but this would require a more sophisticated ability to factor out common patterns in parsers and expressions. Willis [2024] demonstrates how chain.left1 can be defined in terms of postfix, presented in Haskell as it illustrates the relationship more clearly. The only major syntactical difference of note is that Haskell uses <$> for the map combinator:

```
chainl1 p op  =  postfix p (flip <$> op <*> p)
```

Using this definition, the `postfix` version can be provably shown to be equivalent to the `chain.left1` version. This proof will again use Haskell syntax as it is more suitable for equational reasoning. The goal is to show that `('+' ~> term).map(x1 => x2 => Add(x2, x1))` can be rewritten in the form `op.map(flip) <*> p`, where `p` is `term` and `op` is shown to be `'+' as Add(_, _)`.

```
    (λx1 x2 → Add x2 x1) <$> ('+' *> term)
=     { Definition of flip }
    flip Add <$> ('+' *> term)
=     { Definition of *> (equivalent to ~> in Scala): p *> q = const id <$> p <*> q }
    flip Add <$> (const id <$> '+' <*> term)
=     { Re-association: u <*> (v <*> w) = pure (∘) <*> u <*> v <*> w }
    pure (∘) <*> flip Add <$> (const id <$> '+') <*> term
=     { Applicative fusion: pure f <*> pure x = pure (f x) }
    (flip Add .) <$> (const id <$> '+') <*> term
=     { Functor composition: fmap f ∘ fmap g = fmap (f ∘ g) }
    ((flip Add .) . const id) <$> '+' <*> term
=     { Point-free manipulation }
    (flip . const Add) <$> '+' <*> term
=     { Functor composition, in reverse }
    flip <$> (const Add <$> '+') <*> term
=     { Definition of $> (equivalent to .as in Scala) }
    flip <$> ('+' $> Add) <*> term
```

The same proof can be applied to the other operators, and the `map` combinator distributes over choice:

```
(u | v).map(f) = u.map(f) | v.map(f)
```

Therefore, the proof shows that the final term obtained from `parsley-garnish` can be rewritten in the form `op.map(flip) <*> p`, so the obtained `postfix` parser is equivalent to the hand-written version using `chain.left1`.

7.1.1.3 Evaluating the Arithmetic Expression Language

The evaluating parser variant of the same grammar, as presented in the introduction, has the same resulting form:

```
lazy val expr: Parsley[Float] = chain.postfix[Float](term)(
  ('+' ~> term).map(x1 => x2 => x2 + x1) | ('-' ~> term).map(x1 => x2 => x2 - x1))
```

However, in this case, the output unfortunately fails to compile. Scala's local type inference prevents the compiler from inferring the types of the `x2` parameters, as the arithmetic operators used in the lambdas are overloaded. This problem does not occur for the previous example, since the `Expr` constructors are explicitly and unambiguously typed. Users will have to manually fix `parsley-garnish`'s output by adding an explicit type annotation `(x2: Float)`.

Resolving this issue is future work: the most likely solution is to refactor the parser so that they do not take curried functions, which would make it easier for Scala to infer their types. For example, if the parser was refactored into the `chain.left1` form utilising the `as` combinator, the addition function `_ + _` would be in a fully uncurried form.

Summary Direct left-recursion is the most straightforward form of left-recursion to detect, so it is unsurprising that `parsley-garnish` handles it well. The transformation on these test examples are provably correct, and the

resulting parsers are relatively clear and idiomatic. `parsley-garnish` also takes care to improve the likelihood of producing compilable output, although there is still some future work to be done in this area. The most significant weakness is the inability to specialise the `postfix` parser into a more specific form, however this is not a critical issue as the `postfix` form is still correct and idiomatic.

7.1.2 Indirect Left-Recursion

Instances of *indirect* left-recursion are harder to detect, since the parser's reference to itself takes more than one step to reach. Consider the following alternative grammar for arithmetic expressions, reduced to only addition for simplicity:

$$\langle expr \rangle ::= \langle add \rangle \mid \text{`('} \langle expr \rangle \text{`)'} \mid \langle number \rangle$$
$$\langle add \rangle ::= \langle expr \rangle \text{ `+'} \langle expr \rangle$$

The indirect left-recursive cycle arises since `expr` firstly needs to parse `add`, which firstly needs to parse `expr`, and so on. This grammar can be naïvely translated in its left-recursive form in the following code segment. For variety, this example utilises the *Parser Bridges* pattern instead of using `lift` combinators:

```
enum Expr {
  case Num(n: Int)
  object Num extends ParserBridge1[Int, Num]
  case Add(x: Expr, y: Expr)
  object Add extends ParserBridge2[Expr, Expr, Add]
}
lazy val expr: Parsley[Expr] = add | '(' ~> expr <~ ')' | Num(number)
lazy val add: Parsley[Expr] = Add(expr, '+' ~> expr)
```

`parsley-garnish` successfully detects the indirect left-recursion and offers an automated fix. For brevity, the parser type annotations will be omitted in subsequent examples, as they are not changed by `parsley-garnish`.

```
lazy val expr = chain.postfix[Expr]('(' ~> expr <~ ')' | number.map(x1 => Num(x1)))
                                   (('+' ~> expr).map(x1 => x2 => Add(x2, x1)))
lazy val add = Add(expr, '+' ~> expr) // unchanged and no longer referenced by expr
```

Hidden left-recursion:

$$\langle a \rangle ::= \langle b \rangle \langle a \rangle \cdots$$
$$\langle b \rangle ::= \epsilon$$

```
lazy val a: Parsley[Int] = b ~> a
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// error: [FactorLeftRecursion] Left-recursion detected, but could not be removed from a.
// The resulting chain would be given a parser which consumes no input,
// causing it to loop indefinitely:
// chain.postfix[Int](some(digit) ~> a)(pure(x1 => x1))

lazy val b: Parsley[Int] = many(digit).map(_.mkString.toInt)
```

Chapter 8
Related Work

Library-specific linters are rare because they require a non-trivial amount of work to implement. Even more niche is linters for embedded DSLs. This thesis can also be seen as a rough framework for implementing a linter for an eDSL (esp. Parser datatype to represent DSL model). afaik nothing this complex has been done in scalafix

Related to scalafix, so it could be possible to do very similar things as I've done: Rust (I can see this as possible, ask Boogle people for thoughts?) – Clippy lint passes C# – Roslyn

Library-specific linters: xunit.analyzers – seems to be relatively simple one-off rules though, not a DSL

Things that could help: Squid quasiquotes: type-safe and hygienic quasiquotes – but only for scala 2.11/12 * squid quasiquotes have rudimentary function inlining features https://infoscience.epfl.ch/record/231700 Scala 3 macros: I wonder if this could help?

Related concepts to the Function stuff: Metaprogramming and multi-staged programming? a dual: instead of for optimisation/code-generation purposes, we do it just for stringifying Actually I feel like I've borrowed a lot of stuff from metaprogramming optimisation techniques – Haskell Parsley itself

The leftrec transformation is not new but its implementation as a linter rule is novel

Chapter 9
Conclusion

9.1 Review

The left-recursion factoring transformation could be implemented in a staged version of `parsley` to eliminate left-recursion at compile-time with no run-time overhead. However, from a design perspective this is potentially controversial as it obfuscates the PEG semantics of the parser combinator library, allowing CFG-like parsers to be written directly. One could also argue (pedagogical perspective? not really) that the higher-level abstraction of the `chain` combinators is a more desirable form to write parsers for left-associative operations, rather than a lower-level left-recursive grammar rule.

9.2 Future Work

Separate into practical (improvements to parsley-garnish) and theoretical (cool research things)?

9.2.1 Expression AST

Eta reduction – this is more complicated than in Haskell since Scala has special syntax Proper partial evaluation, not just normalisation (if we reduce to fully closed terms 1+1 can we get it to evaluate to 2? – except currently this would be a Translucent term)

9.2.2 Resugaring

https://dl.acm.org/doi/10.1145/2858949.2784755

Bibliography

Amin, Nada et al. (2016). "The Essence of Dependent Object Types". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley et al. Cham: Springer International Publishing, pp. 249–272. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_14. URL: https://doi.org/10.1007/978-3-319-30936-1_14.

Baars, Arthur I. and S. Doaitse Swierstra (2004). "Type-safe, self inspecting code". In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: Association for Computing Machinery, pp. 69–79. ISBN: 1581138504. DOI: 10.1145/1017472.1017485. URL: https://doi.org/10.1145/1017472.1017485.

Bakel, Steffen van (2022). *Lecture Notes on Type Systems for Programming Languages*. URL: https://www.doc.ic.ac.uk/~svb/TSfPL/.

Barendregt, Henk P. (1984). "Chapter 2 - Conversion". In: *The Lambda Calculus*. Vol. 103. Studies in Logic and the Foundations of Mathematics. ISSN: 0049-237X. Elsevier, pp. 22–49. DOI: https://doi.org/10.1016/B978-0-444-87508-2.50010-1. URL: https://www.sciencedirect.com/science/article/pii/B9780444875082500101.

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.

Charguéraud, Arthur (Oct. 2012). "The Locally Nameless Representation". In: *Journal of Automated Reasoning* 49.3, pp. 363–408. ISSN: 1573-0670. DOI: 10.1007/s10817-011-9225-2. URL: https://doi.org/10.1007/s10817-011-9225-2.

Cheney, James and Ralf Hinze (July 2003). *First-class phantom types*. Tech. rep. Cornell University. URL: https://hdl.handle.net/1813/5614.

Church, Alonzo (1936). "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2, pp. 345–363. ISSN: 00029327, 10806377. DOI: 10.2307/2371045. URL: https://doi.org/10.2307/2371045 (visited on 06/06/2024).

Cremet, Vincent et al. (2006). "A Core Calculus for Scala Type Checking". In: *Mathematical Foundations of Computer Science 2006*. Ed. by Rastislav Královič and Paweł Urzyczyn. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–23. ISBN: 978-3-540-37793-1. DOI: 10.1007/11821069_1. URL: https://doi.org/10.1007/11821069_1.

Danvy, Olivier and Lasse R. Nielsen (2001). "Defunctionalization at work". In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '01. Florence, Italy: Association for Computing Machinery, pp. 162–174. ISBN: 158113388X. DOI: 10.1145/773184.773202. URL: https://doi.org/10.1145/773184.773202.

de Bruijn, N.G (1972). "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5, pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

Eichenroth, Friedrich, Patrick Rein, and Robert Hirschfeld (2022). *Fast packrat parsing in a live programming environment*. improving left-recursion in parsing expression grammars 135. ISBN: 978-3-86956-503-3. DOI: 10.25932/publishup-49124. URL: https://doi.org/10.25932/publishup-49124.

Filinski, Andrzej and Henning Korsholm Rohde (2004). "A Denotational Account of Untyped Normalization by Evaluation". In: *Foundations of Software Science and Computation Structures*. Ed. by Igor Walukiewicz. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 167–181. ISBN: 978-3-540-24727-2. DOI: 10.1007/978-3-540-24727-2_13. URL: https://doi.org/10.1007/978-3-540-24727-2_13.

Fokker, Jeroen (1995). "Functional parsers". In: *Advanced Functional Programming*. Ed. by Johan Jeuring and Erik Meijer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–23. ISBN: 978-3-540-49270-2. DOI: 10.1007/3-540-59451-5_1. URL: https://doi.org/10.1007/3-540-59451-5_1.

Ford, Bryan (Jan. 2004). "Parsing expression grammars: a recognition-based syntactic foundation". In: *SIGPLAN Not.* 39.1, pp. 111–122. ISSN: 0362-1340. DOI: 10.1145/982962.964011. URL: https://doi.org/10.1145/982962.964011.

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).

Gibbons, Jeremy and Nicolas Wu (2014). "Folding domain-specific languages: deep and shallow embeddings (functional pearl)". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, pp. 339–347. ISBN: 9781450328739. DOI: 10.1145/2628136.2628138. URL: https://doi.org/10.1145/2628136.2628138.

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.

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.

Jaglin, Brice (Feb. 2024). *Unleashing Scalafix potential with custom rules*. en. Nantes, France. URL: https://scala.io/talks/unleashing-scalafix-potential (visited on 03/25/2024).

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).

Lindley, Sam (June 2005). "Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages". Ph.D. Thesis. Edinburgh: University of Edinburgh. URL: http://hdl.handle.net/1842/778.

Maclaurin, Dougal, Alexey Radul, and Adam Paszke (2023). "The Foil: Capture-Avoiding Substitution With No Sharp Edges". In: *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*. IFL '22. <conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>: Association for Computing Machinery. ISBN: 9781450398312. DOI: 10.1145/3587216.3587224. URL: https://doi.org/10.1145/3587216.3587224.

Mascarenhas, Fabio, Sérgio Medeiros, and Roberto Ierusalimschy (2014). "On the relation between context-free grammars and parsing expression grammars". In: *Science of Computer Programming* 89, pp. 235–250. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.01.012. URL: https://doi.org/10.1016/j.scico.2014.01.012.

McBride, Conor and James McKinna (2004). "Functional pearl: i am not a number–i am a free variable". In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: Association for Computing Machinery, pp. 1–9. ISBN: 1581138504. DOI: 10.1145/1017472.1017477. URL: https://doi.org/10.1145/1017472.1017477.

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.

Moore, Bob (Apr. 2000). "Removing Left Recursion from Context-Free Grammars". In: Association for Computational Linguistics. URL: https://www.microsoft.com/en-us/research/publication/removing-left-recursion-from-context-free-grammars/.

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

Parreaux, Lionel, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso (2019). "Towards improved GADT reasoning in Scala". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. Scala '19. London, United Kingdom: Association for Computing Machinery, pp. 12–16. ISBN: 9781450368247. DOI: 10.1145/3337932.3338813. URL: https://doi.org/10.1145/3337932.3338813.

Peyton Jones, Simon and Simon Marlow (July 2002). "Secrets of the Glasgow Haskell Compiler inliner". In: *J. Funct. Program.* 12.5, pp. 393–434. ISSN: 0956-7968. DOI: 10.1017/S0956796802004331. URL: https://doi.org/10.1017/S0956796802004331.

Pfenning, F. and C. Elliott (June 1988). "Higher-order abstract syntax". In: *SIGPLAN Not.* 23.7, pp. 199–208. ISSN: 0362-1340. DOI: 10.1145/960116.54010. URL: https://doi.org/10.1145/960116.54010.

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.

Reynolds, John C. (1972). "Definitional interpreters for higher-order programming languages". In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852. URL: https://doi.org/10.1145/800194.805852.

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.

Scala Center (Feb. 2024). *Scalafix: Developer Guide*. en. URL: https://scalacenter.github.io/scalafix/docs/developers/tutorial.html (visited on 03/25/2024).

Scalameta (Nov. 2023). *Scalameta: SemanticDB Specification*. en. URL: https://scalameta.org/docs/semanticdb/specification.html (visited on 03/25/2024).

Shabalin, Denys, Eugene Burmako, and Martin Odersky (Mar. 2013). *Quasiquotes for Scala*. Tech. rep. EPFL, p. 15. URL: http://infoscience.epfl.ch/record/185242.

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.

Swierstra, S. Doaitse and Luc Duponcheel (1996). "Deterministic, error-correcting combinator parsers". In: *Advanced Functional Programming*. Ed. by John Launchbury, Erik Meijer, and Tim Sheard. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 184–207. ISBN: 978-3-540-70639-7. DOI: 10.1007/3-540-61628-4_7. URL: https://doi.org/10.1007/3-540-61628-4_7.

Willis, Jamie (Mar. 2024). "Parsley: Optimising and Improving Parser Combinators". Ph.D. 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.

Willis, Jamie and Nicolas Wu (2021). "Design patterns for parser combinators (functional pearl)". In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*. Haskell 2021. Virtual, Republic of Korea: Association for Computing Machinery, pp. 71–84. ISBN: 9781450386159. DOI: 10.1145/3471874.3472984. URL: https://doi.org/10.1145/3471874.3472984.

— (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 Haskell `gigaparsec` library uses a continuation-passing style (CPS) to define its parsers, which makes it easier to reason about their semantics. The approach taken to prove the following parser law 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
```