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 (EDSL) [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 with little or no 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 evaluates to `2.5`, as expected!

Contributions

This project introduces two major contributions in separate areas:

1. An auto-fix lint rule that detects and refactors left-recursive parsers to an idiomatic form that can be handled correctly by `parsley`. This rule can handle all forms of left-recursion, although to varying degrees of success. Although the most significant lint rule produced in this project is this *left-recursion factoring* rule, `parsley-garnish` also implements a number of simpler rules that enforce idiomatic design patterns in `parsley` code.

2. The motivation behind the *infrastructure* required to support complex lint rules such as the left-recursion rule, and its implementation. At a high-level, these are two separate intermediate representations that abstract away from the generic Scala abstract syntax tree (AST):

    - A parser AST designed to mirror the AST of the `parsley` DSL, which allows *code* of parsers to be manipulated and transformed in a high-level declarative manner. This is based on the insight that any DSL and its accompanying linter are simply different semantic interpretations on the same underlying structure: the former folds over its AST structure to evaluate its results, while the latter performs a fold to emit lint diagnostics and pretty-print a transformed AST.

- An expression AST based on the λ-calculus, granting static expression ASTs the ability to be normalised via β-reduction. This approach draws inspiration from staged metaprogramming frameworks for manipulating ASTs.

Outline

?? 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, **??** 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 have seen in my experiences as a teaching assistant for the second-year undergraduate WACC compilers project at Imperial.

?? 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. **??** 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 **??**, which also explores more advanced rules that are unlocked by the new AST. Finally, **??** evaluates the performance benchmarks and the effectiveness of the left-recursion factoring rule, and discusses the limitations of the current implementation. The remainder of the report concludes with discussion of related work and future directions for the project.