Chapter 1
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

1.1 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 **??**:

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

1.1.1 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.

1.1.1.1 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.

1.1.1.2 Arithmetic Expression Language

As a larger-scale example, the left-associative arithmetic operators from **??** 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`.

1.1.1.3 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.

1.1.2 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)
```