

## 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. This makes it relatively easy to detect and handle, regardless of the transformation technique used. The following minimal grammar for a postfix unary incrementing operator is directly left-recursive:

$$\langle inc \rangle ::= \langle inc \rangle '+' \mid \langle number \rangle$$

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))
=    {  $\eta$ -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.

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.

`chain.left1` can be defined in terms of `postfix` [Willis 2024]. This is presented in Haskell syntax, as it illustrates the relationship more clearly. The only major difference in syntax is that Haskell uses `<$>` for the map combinator:

$$\text{chainl1 } p \text{ op} = \text{postfix } p \text{ (flip } <\$> \text{ op } <*> p)$$

Using this definition, the `postfix` version can be provably shown to be equivalent to the `chain.left1` version. This proof will again be shown in Haskell, as it is more suitable for equational reasoning:

$$(\lambda x1 \ x2 \rightarrow \text{Add } x2 \ x1) <\$> ('+' *> \text{term})$$

```

=    { 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

```

### 1.1.2 Indirect Left-Recursion

Indirect left-recursion is more subtle and tricky to spot, since the parser's reference to itself takes more than one step to reach. Consider the following alternate grammar for arithmetic expressions, reduced to only addition for simplicity:

$$\begin{aligned}
 \langle expr \rangle &::= \langle add \rangle \mid '(\langle expr \rangle)' \mid \langle number \rangle \\
 \langle add \rangle &::= \langle expr \rangle '+' \langle expr \rangle
 \end{aligned}$$

The indirect left-recursive cycle arises since `expr` firstly needs to parse `add`, which needs to parse `expr` first. This pattern of left-recursion can be formulated as the following parser:

```

enum Expr {
  case Num(n: Int)
  case Add(x: Expr, y: Expr)
}

lazy val expr: Parsley[Expr] = add | '(' ~> expr <~ ')' | num
lazy val add: Parsley[Expr] = (expr, '+' ~> expr).zipped(Add(_, _))

```

For brevity, the type annotations will be omitted in the transformed code, as they are not changed by parsley-garnish.

```

// Transformed by parsley-garnish
lazy val expr = chain.postfix[Expr]('(' ~> expr <~ ')' | num)
                                   (('+' ~> expr).map(x1 => x2 => Add(x2, x1)))
lazy val add = (expr, '+' ~> expr).zipped(Add(_, _)) // no longer referenced by expr

```

$$\begin{aligned}\langle a \rangle &::= \langle b \rangle \langle a \rangle \dots \\ \langle b \rangle &::= \epsilon\end{aligned}$$

```

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)

```