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 Evaluation criteria: * Was it able to detect this instance of left-recursion? * If it performed an auto-fix, correctness * Clarity of output * Does it compile?

* Dealing with types of syntactic sugar in parsley

Types of left-recursion: Assuming we have the number parser from before:

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

Direct left-recursion

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

```
enum Expr {
  case Num(n: Int)
  case Inc(x: Expr)
}

val incs: Parsley[Expr] = Inc.lift(incs) <~ "+" | Num.lift(number)

// 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)
```

Note `chain.postfix[Expr]` type ascription is necessary in this case to help type inference – even in the hand-written version. This is not always necessary, but since parsley-garnish cannot typecheck its outputs, it is always included.

Indirect left-recursion

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

```
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
```

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)
```