Chapter 1
Using the New Parser ast

1.1 Removing Left-Recursion: Revisited

Armed with the new ast representation of parsers, the left-recursion factorisation transformation can be revisited. What has changed?

- Parser terms should be lawfully simplified to a more readable form before being written as a Scalafix rewrite patch.
- 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 **??**.

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

```
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)
  }
}
```

The Final Result

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

1.2 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

1.3 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

?? 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