Chapter 1
Removing Left-Recursion

Left-recursive grammar rules are a common pattern to represent left-associativity. Take for example the following definition of a left-associative addition operator:

$$\langle expr \rangle ::= \langle expr \rangle \text{ `+' } \langle term \rangle \mid \langle term \rangle$$

Since the first production of $\langle expr \rangle$ is itself, this rule is said to be left-recursive. This poses a problem for recursive-descent parsers, such as those that `parsley` produces: it will try to parse $\langle expr \rangle$ by first trying to parse $\langle expr \rangle$, and so on, resulting in an unproductive infinite loop.

Although it is possible to address the issue by transforming the grammar with algorithms such as Paull's algorithm [Moore 2000], in the context of parser combinators this is considered an anti-pattern by Willis and Wu [2021]. They argue that this transformation obscures the original intent of the grammar, and exposes lower-level implementation details when this can be abstracted behind a combinator. Instead, they propose that the idiomatic method to handle left-recursion in parser combinators is to use the `chain` family of combinators [Fokker 1995]. These combinators embody the behaviour of right-associating left-recursive rules and correcting the result back to a left-associative form.

Left-recursion often comes as a nasty surprise for novice users naïvely translating BNF grammars into parser implementations – this issue is not unique to parser combinators, but also extends to many popular parser generators that use recursive-descent. Thus, it would be beneficial to provide a linting rule for `parsley` that can warn users when parsers are left-recursive. In fact, the next major release of `parsley` 5.0 will introduce a `detectDivergence` combinator, which performs *dynamic* analysis to detect unproductive looping at runtime. Therefore, `parsley-garnish` could complement this functionality with an auto-fix rule to refactor left-recursive parsers to use `parsley`'s idiomatic `chain` combinators.

Running example The following left-recursive parser and its transformation into a non-left-recursive form will be used as an example for this chapter:

```
lazy val example: Parsley[String] = (example, string("a")).zipped(_ + _) | string("b")
```

The `example` parser intends to express the following simple grammar expressed using left-recursion. The goal is to refactor `example` so that it retains the intended semantics, but is transformed into a parser that `parsley` can handle correctly.

$$\langle example \rangle ::= \langle example \rangle \text{ `a' } \mid \text{ `b'}$$

1.1 The Left-Recursion Factoring Transformation

`parsley-garnish` bases its left-recursion factoring transformation on the work of Baars and Swierstra [2004], adapted to fit the PEG semantics of `parsley`. At a high-level, the transformation involves "unfolding" each non-terminal production into three parts:

- `results`: The semantic actions of the parser, if it can derive the empty string. Conceptually, this has type `Option[A]` where `A` is the type of the result.
- `nonLeftRec`: The non-left-recursive part of the parser that does not derive the empty string. This will have some type `Parsley[A]`.
- `leftRec`: The left-recursive call, which in the general left-recursive case, corresponds to a repeated postfix operator of type `Parsley[A => A]`. This is a function which requires the semantics of the left-recursive non-terminal argument.

This transformation is applied in-order to each parser in the source file, replacing the original parser with its factored form if it was left-recursive. An unfolded parser is recombined using `chain.postfix`: this combinator encapsulates the general form of left-associative parsing, and most other iterative combinators can be derived from it [Willis 2024].

```
def transform(
  results: Option[A],
  nonLeftRec: Parsley[A],
```

```
  leftRec: Parsley[A => A]
): Parsley[A] = {
  val result: Parsley[A] = results match {
    case None    => empty
    case Some(x) => pure(x)
  }
  // The type signature of postfix in Parsley:
  // def postfix[A](p: Parsley[A])(op: => Parsley[A => A]): Parsley[A]
  chain.postfix(nonLeftRec | results)(leftRec)
}
```

1.2 Necessary Infrastructure

The comparatively simple linting rules discussed in the previous ?? were implemented by directly inspecting the generic Scala AST provided by Scalafix. However, even though `parsley` programs are written in Scala, it is important to remember that `parsley` is a DSL borrowing Scala as a host language. Domain-specific transformations like left-recursion factoring are therefore naturally defined as transformations on the `parsley` AST, at a higher level of abstraction than the generic Scala AST. Thus, this section discusses the extra infrastructure used to support the left-recursion factoring transformation:

- Firstly, §1.2.1 motivates the idea of using an intermediate AST representation for parsers, distinct from the general-purpose Scala AST.
- Following this, §1.2.2 shows how the source Scala AST is converted into this intermediate representation.
- Finally, §1.2.3 discusses how the intermediate AST is converted back into Scala code so that it can be applied as a Scalafix patch.

1.2.1 An Intermediate AST

The transformations described by Baars and Swierstra [2004] require an explicit representation of the grammar and production rules so that they can be inspected and manipulated before generating code. They achieve this by representing parsers as a deep-embedded datatype in the form of an intermediate AST, similarly to `parsley`.

Since `parsley-garnish` is a linter, by nature, it has access to an explicit grammar representation in the form of the full `scala.meta.Tree` AST of the source program. However, this datatype represents general-purpose abstract Scala syntax, rather than the abstract syntax of a specialised parser combinator DSL. This makes it not well-suited for performing domain-specific operations over the AST.

Take for example the task of combining two AST nodes `Term.Name("p")` and `Term.Name("q")`, representing named parsers `p` and `q`, with the combinator `<*>` (pronounced "ap"). This operation can be concisely expressed with Scalameta quasiquotes, rather than manually writing out the full explicit AST:

```
q"p <*> q" ==
  Term.ApplyInfix(
    Term.Name("p"),
    Term.Name("<*>"),
    Type.ArgClause(Nil),
    Term.ArgClause(List(Term.Name("q")), None)
  )
```

However, the operation of inspecting the individual parsers `p` and `q` is not as straightforward. Although quasiquotes can be used as extractor patterns in pattern matching, this usage is discouraged due to limitations in their design that makes it easy to accidentally introduce match errors[1]. Thus, extracting the parsers necessitates a long-winded pattern match like so:

```
val ap = SymbolMatcher.normalized("parsley.Parsley.`<*>`")
def deconstructAp(parser: Term) = parser match {
```

[1] https://scalameta.org/docs/trees/guide.html#with-quasiquotes-1

```
  case Term.ApplyInfix(p, ap(_), _, Term.ArgClause(List(q), _)) => (p, q)
}
```

This involves dealing with abstract general-purpose syntax constructs like `Term.ApplyInfix`, which are low-level details irrelevant to the task of manipulating parsers. Although this is not an issue for simple one-off transformations, for more specialised transformations like left-recursion factoring, it would be desirable to abstract away from these low-level syntactic details. This motivates the need for an higher-level, intermediate AST representation that is more specialised to the domain of parser combinators.

1.2.1.1 The Parser ADT

`parsley-garnish` therefore takes a similar approach as Baars and Swierstra [2004] and `parsley` itself, building an intermediate AST as a deep-embedded parser combinator tree. Fig. 1.1 shows how this is implemented as a `Parser` algebraic data type (ADT). All `Parser` types represent `parsley` combinators, with the sole exception of `NonTerminal` to represent references to named parsers.

```
trait Parser
case class NonTerminal(ref: Symbol) extends Parser
case class Pure(x: Term) extends Parser
case object Empty extends Parser
case class <*>(p: Parser, q: Parser) extends Parser
case class <|>(p: Parser, q: Parser) extends Parser
case class Str(s: String) extends Parser
case class Chr(c: Char) extends Parser
```

Fig. 1.1: A subset of the `Parser` ADT, representing the core combinators in `parsley-garnish`.

Deconstructing parsers Scala allows users to define symbolic class names (as evidenced by the definitions of `<*>` and `<|>` in fig. 1.1), and provides syntactic sugar to pattern match on these constructors using infix notation. This results in a very natural and readable pattern matching syntax:

```
def deconstructAp(parser: Parser) = parser match { case p <*> q => (p, q) }
```

Constructing parsers Defining infix operators as extension methods on the `Parser` trait provides a similar syntactic sugar for constructing parsers:

```
extension (p: Parser) {
  def <*>(q: Parser) = <*>(p, q)
  def |(q: Parser) = <|>(p, q)
  def map(f: Term) = FMap(p, f)
}
extension (ps: List[Parser]) {
  def zipped(f: Term) = Zipped(f, ps)
}
```

This makes the syntax for writing `Parser` terms feel natural and similar to writing `parsley` code. For example, notice how constructing the *code* representation of the `example` parser resembles how the original parser itself would be written:

```
val EXAMPLE = NonTerminal(Sym(Term.Name("example").symbol))

// val example: Parsley[String] =     (example,  string("a")).zipped(  _ + _ ) | string("b")
   val example: Parser          = List(EXAMPLE,     Str("a")).zipped(q"_ + _") |    Str("b")
```

1.2.2 Lifting to the Intermediate Parser AST

Converting the raw Scala AST to this intermediate parser combinator AST requires the following basic operations:

1. Identifying all named parsers defined in the source program – these correspond to non-terminal symbols in the grammar.
2. Lifting the definition each parser into the intermediate AST, i.e. a `Parser` object.
3. Collecting these into a map to represent the high-level grammar – the unique symbol of each named parser is mapped to its corresponding `Parser` object, along with extra metadata required for the transformation.

Most importantly, this metadata includes a reference to a parser's original node in the Scala AST, so lint diagnostics or code rewrites can be applied to the correct location in the source file:

```
case class ParserDefn(name: Term.Name, parser: Parser, tpe: Type.Name, originalTree: Term)
```

1.2.2.1 Identifying Named Parsers

Finding AST nodes corresponding to the definition sites of named parsers involves pattern matching on `val`, `var`, and `def` nodes with a type inferred as some `Parsley[_]`. This type information is queried using the Scalafix semantic API on the node's symbol information. Consider the labelled AST structure of the `example` parser:

```
// lazy val example: Parsley[String] = (example, string("a")).zipped(_ + _) | string("b")
// ^^^^     ^^^^^^^  ^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// mods      pats        decltpe                              rhs

val exampleTree = Defn.Val(
  mods = List(Mod.Lazy()),
  pats = List(Pat.Var(Term.Name("example"))),
  decltpe = Some(
    Type.Apply(Type.Name("Parsley"), Type.ArgClause(List(Type.Name("String"))))
  ),
  rhs = Term.ApplyInfix(...)
)
```

Note that the `decltpe` field refers to the *syntax* of the explicit type annotation, not the *semantic* information the variable's inferred type. Therefore, this field will not always be present, so in the general case, the type must be queried via a symbol information lookup:

```
exampleTree match {
  case Defn.Val(_, List(Pat.Var(varName)), _, body) =>
    println(s"qualified symbol = ${varName.symbol}")
    // Query the symbol information of the variable name, and get its type signature
    varName.symbol.info.get.signature match {
      // Scalameta treats this as a zero-arg method, so the relevant part is its return type
      case MethodSignature(_, _, returnType) =>
        println(s"type = $returnType")
        println(s"structure of type object = ${returnType.structure}")
    }
}
// qualified symbol = path/to/package/ObjectName.example.
// type = Parsley[String]
// structure of type object = TypeRef(
//   NoType,
//   Symbol("parsley/Parsley#"),
//   List(TypeRef(NoType, Symbol("scala/Predef.String#"), List()))
// )
```

Seeing that the type of this AST node is `Parsley[String]`, `parsley-garnish` can then proceed to convert the `rhs` term into a `Parser` ADT object. The map entry uses the fully qualified symbol for `example` as the key, and the lifted `Parser` object as the value.

1.2.2.2 Converting Scalameta Terms to the Parser ADT

Having identified the AST nodes which represent parsers, they need to be transformed into the appropriate `Parser` representation. This involves pattern matching on the `scala.meta.Term` to determine which parser combinator it represents, and then constructing the appropriate `Parser` instance.

Each `Parser` defines a partial function `fromTerm` to instantiate a parser from the appropriate `scala.meta.Term`. These `fromTerm` methods perform the menial work of pattern matching on the low-level syntactic constructs of the Scala AST. All `fromTerm` methods are combined to define the `toParser` extension method on `scala.meta.Term` – this is where AST nodes are lifted to their corresponding `Parser` representation.

The pattern matching example from §1.2.1 makes a reappearance in the definition of `<*>.fromTerm`, where the arguments to the `<*>` combinator are instead recursively lifted to `Parser` objects:

```
// Type signatures in Parsley:
// p: Parsley[A => B], q: =>Parsley[A], p <*> q: Parsley[B]
case class <*>(p: Parser, q: Parser) extends Parser
object <*> {
  // Match the specific symbol for parsley's <*> combinator
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

  def fromTerm: PartialFunction[Term, <*>] = {
    // Pattern match succeeds only if the term has the structure 'p <*> q'
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
      p.toParser <*> q.toParser
  }
}
```

Where a combinator takes a non-parser argument, this is treated as a black box and kept as a raw AST node of type `scala.meta.Term`:

```
// x: A, pure(x): Parsley[A]
case class Pure(x: Term) extends Parser
object Pure {
  val matcher = SymbolMatcher.normalized("parsley.ParsleyImpl.pure")

  def fromTerm: PartialFunction[Term, Pure] = {
    // expr is an opaque AST node that cannot be further inspected
    case Term.Apply(matcher(_), Term.ArgClause(List(expr), _)) => Pure(expr)
  }
}
```

1.2.2.3 Building the Grammar Map

The overall process of converting the source file AST to a high-level map of the grammar can therefore be expressed as a single traversal over the AST:

```
// Encapsulate all valid pattern matches into a single extractor object
object VariableDecl {
  def unapply(tree: Tree): ParserDefn = tree match {
    // isParsleyType uses symbol info to check if variable type is Parsley[_]
    case Defn.Val(_, List(Pat.Var(varName)), _, body) if isParsleyType(varName) =>
      // If the pattern match is successful, convert the definition body to a Parser
      // Collect metadata and bundle into a parser definition object
```

```
      ParserDefn(
        name = varName,
        parser = body.toParser,
        tpe = getParsleyType(varName),
        originalTree = body
      )
    // ... similar cases for Defn.Var and Defn.Def
  }
}

type Grammar = Map[Symbol, ParserDefn]
val grammar: Grammar = doc.tree.collect {
  // Every AST node that satisfies the pattern match is added to the map
  case VariableDecl(parserDef) => parserDefn.name.symbol -> parserDef
}.toMap
```

1.2.3 Lowering Back to the Scalameta AST

After all necessary transformations have been applied to parser terms, the final step is to convert them back to a textual representation to be applied as a Scalafix patch. Parsers can be lowered back to `scala.meta.Term` nodes by the inverse of the original `fromTerm` transformation. The `Parser` trait defines this transformation as the method `term`, using quasiquotes to simplify the construction of the `scala.meta.Term` nodes. For example:

```
case class Zipped(func: Function, parsers: List[Parser]) extends Parser {
  val term: Term = q"(..${parsers.map(_.term)}).zipped(${func.term})"
}
```

The `Parser` trait can then define a `toString` method based on Scalameta's AST pretty-printer on the `term` field. This "stringified" term is therefore guaranteed to be syntactically well-formed and is the appropriate type expected by Scalafix's `Patch` methods.

1.3 Implementing the Left-Recursion Transformation

With the `Parser` AST defined, it is now possible to implement the left-recursion factoring transformation in a more high-level manner. Using the `grammar` map collected previously, the overall transformation can be summarised as follows (in Scala-like pseudocode for brevity):

```
val transformedGrammar = grammar.to(mutable.Map)
for (currentNT <- grammar.keys) {
  // Unfold each non-terminal parser into its three parts
  val (results, nonLeftRec, leftRec) = unfold(transformedGrammar, currentNT)
  // Update the grammar with the recombined postfix parser if it was left-recursive
  transformedGrammar(currentNT).parser = transform(results, nonLeftRec, leftRec)
}
// Generate rewrite patches for each parser definition
transformedGrammar.mapValues { defn =>
  Patch.replaceTree(defn.originalTree, defn.parser.toString)
}
```

The unfolding transformation is defined as follows, which is dynamically dispatched to the appropriate combinator's `unfold` method:

```
def unfold(env: Grammar, currentNT: Symbol) =
  env(currentNT).parser.unfold(currentNT, env, visited = Set.empty)
```

1.3.1 Unfolding the Core Combinators

The most important core combinators in the left-recursion transformation are `NonTerminal`, `Pure`, `Empty`, `<|>`, and `<*>`. Character combinators such as `Str` (`string`), `Chr` (`char`), and `Item` (`item`), although technically handled by `parsley-garnish` as primitives, are grouped together as they behave in a simple and uniform manner. The rest of the combinators in `parsley` are defined in terms of the primitive combinators, and are desugared into the core combinators during unfolding.

Non-terminals Non-terminals are the primary base case of the recursive unfolding transformation. There are three cases to consider when a non-terminal symbol is encountered:

- This non-terminal is the same as the current non-terminal being analysed, which means it may be in a left-recursive position. Recall that the `leftRec` result corresponds to a function, where the call to the left-recursive non-terminal is stripped off. Therefore in this case, the left-recursive call is the `identity` function which just returns its argument, i.e. the non-terminal itself. It is not an issue if the non-terminal is not in left-recursive position, as the `leftRec` result will be subsumed by the `nonLeftRec` result during the final recombination.
- The non-terminal refers to a different non-terminal, but has already been visited in the current unfolding process. This means it should not be recursively visited again, as it has already been factored out. It is not left-recursive and the `nonLeftRec` part is just a reference to this non-terminal's name.
- The non-terminal refers to a different non-terminal that has not been visited yet. In this case, the non-terminal is recursively unfolded, and the `visited` set is updated to include this non-terminal.

The three cases can be implemented as so:

```
case class NonTerminal(ref: Symbol) extends Parser {
  def unfold(currentNT: Symbol, env: Grammar) =
    if (ref == currentNT) (None, Empty, Pure(q"identity"))
    else if (visited.contains(ref)) (None, NonTerminal(ref), Empty)
    else env(ref).parser.unfold(currentNT, env, visited + ref)
}
```

Base cases The base cases are mostly straightforward. The unfolded `results` portion can be reasoned with by the combinators' correspondence to PEG expressions [Eichenroth, Rein, and Hirschfeld 2022], as shown in table 1.1. If the combinator can succeed by parsing the empty string, `results` is a `Some` value of its semantic action, otherwise it is `None`. The other two portions are also easy: none are left-recursive, and if the combinator consumes input on success then `nonLeftRec` is just the combinator itself. Thus, the base cases can be summed up as follows:

```
/* Pure */                 (Some(x), Empty, Empty)
/* Empty */                (None, Empty, Empty)
/* p: Str | Chr | Item */  (None, p, Empty)
```

Combinator	Can succeed without consuming input?
`pure(x)`	Yes, semantic action is `x`.
`empty`	No, combinator fails immediately regardless of input.
`string(s)`	Functionally, no. Can theoretically be given the empty string `""`, but this is illegal in `parsley` and triggers a runtime exception.
`char(c)`	No, always consumes the given character.
`item`	No, always consumes any single character.

Table 1.1: Semantic action (`results`) portion behaviour for each base case.

Choice (<|>) This recursive case is also relatively simple: each branch is unfolded, then the `nonLeftRec` and `leftRec` portions are recombined with the `|` combinator. The `results` portion is that of the left branch if it can derive the empty string, otherwise it is the right branch. This behaviour differs slightly from the original presentation of the transformation by Baars and Swierstra [2004], where `|` is based on CFG semantics. The consequence of this is that choice is unbiased in the original transformation, triggering an ambiguity error if both branches can succeed without consuming input. However, since `parsley` uses PEG semantics, this ambiguity is not possible – the choice operation represented by the `|` combinator is *left-biased*, so if the left parser succeeds the right parser is not attempted.

```
case class <|>(p: Parser, q: Parser) extends Parser {
  def unfold() = {
    val ((pe, pn, pl), (qe, qn, ql)) = (p.unfold, q.unfold)
    (pe.orElse(qe), pn | qn, pl | ql)
  }
}
```

Ap (<*>) This is an important case, as `parsley-garnish` chooses to use `<*>` as the primitive combinator for composing parsers. It is also the most complex:

- `p <*> q` can succeed without consuming input only if *both* `p` and `q` can do so. Parser combinators abide by the law that `pure(f) <*> pure(x) = pure(f(x))`, so the resulting semantic action is that of `p` applied as a function to the semantic action of `q`.
- In the `nonLeftRec` and `leftRec` portions, if `p` can succeed without consuming input, then `q` must be subjected to the semantic value corresponding to `p` [Swierstra and Duponcheel 1996]. This is achieved using the result shown in eq. (1.1), which is a special case of grammar right-factoring in reverse.
  - The `leftRec` portion requires re-associating semantic actions back to the left, which necessitates using the `flip` higher-order function. Therefore, the left branch of the RHS in eq. (1.1) is reformulated as `p.map(flip) <*> q`.
  - Since `leftRec` is a parser of a function, the right branch also needs adjusting: the function composition operation must be used in order to apply the function to `q` with the `<*>` combinator.

$$\texttt{(p <|> pure f) <*> q} \quad \Rightarrow \quad \texttt{(p <*> q) <|> (f <\$> q)} \tag{1.1}$$

Given the above observations, the implementation of the `<*>` combinator's unfolding is as follows:

```
case class <*>(p: Parser, q: Parser) extends Parser {
  def unfold() = {
    val ((pe, pn, pl), (qe, qn, ql)) = (p.unfold, q.unfold)

    val result = for {
      f <- pe
      x <- qe
    } yield App(f, x)

    val nonLefts = {
      val lnl = pn <*> q
      val rnl = pe.map(q"f => qn.map(f)").getOrElse(Empty) // f <$> qn
      lnl | rnl
    }

    val lefts = {
      val llr = pl.map(q"flip") <*> q
      val rlr = pe.map(q"$ql.map(compose)").getOrElse(Empty) // f.map(compose) <$> ql
```

```
      llr | rlr
    }

    (result, nonLefts, lefts)
  }
}
```

1.3.2 Composite Combinators

The remaining combinators are defined in terms of the core combinators, and are unfolded by recursively unfolding their constituent parts. For example, the `map` combinator is defined as `p.map(f) = pure(f) <*> p`, so its unfolding is simply implemented as:

```
case class FMap(p: Parser, f: Term) extends Parser {
  def unfold() = (Pure(f) <*> p).unfold
}
```

Further high-level combinators are defined in a similar manner. The improved sequencing combinators `lift` and `zipped`, as well as bridge constructors using the *Parser Bridges* design pattern [Willis and Wu 2022], represent a further generalisation of the `map` combinator:

```
liftN(f, p1, ..., pN) // explicit lift syntax, where f is a function of arity N
f.lift(p1, ..., pN) // implicit lift syntax
(p1, ..., pN).zipped(f) // zipped syntax, improves Scala's ability to infer types
F(p1, ..., pN) // parser bridge pattern, abstracts the construction of f behind bridge F

// All are equivalent to the following desugared form,
// which is idiomatic in Haskell, but not in Scala for performance reasons
= pure(f.curried) <*> p1 <*> ... <*> pN
```

These are handled in `parsley-garnish` as the `LiftLike` family of combinators, which converts them into the desugared form in terms of core combinators `pure` and `<*>`, allowing the unfolding transformation to be applied:

```
trait LiftLike extends Parser {
  // foldLeft builds a parser of the form 'pure(f.curried) <*> p1 <*> ... <*> pN'
  def unfold() = parsers.foldLeft(Pure(q"$func.curried"))(_ <*> _).unfold
}
```

1.3.3 Defining Utility Functions

In various places within the unfolding transformations, three higher-order functions are utilised, which may not be provided by the Scala standard library:

- The flip function reverses the order of arguments applied to a function. This is not defined in the standard library, so it must be defined manually.
- Function composition is defined in the standard library, but a more versatile curried version is required by the transformation, so it is also defined manually.
- The identity function `identity[A]: A => A` is defined in the standard library.

Therefore, `parsley-garnish` will insert the following definitions into the source file as a patch:

```
def flip[A, B, C](f: A => B => C)(x: B)(y: A): C = f(y)(x)
def compose[A, B, C](f: B => C)(g: A => B)(x: A): C = f(g(x))
```

Although this is not very aesthetically pleasing, it is necessary to bring these higher-order functions into scope for the transformed code to make use of them.

Success...?

With the unfolding transformations defined for all core combinators, the left-recursion factoring transformation is now complete. Running the transformation on the `example` parser yields the output in fig. 1.2.

```
def flip[A, B, C](f: A => B => C)(x: B)(y: A): C = f(y)(x)
def compose[A, B, C](f: B => C)(g: A => B)(x: A): C = f(g(x))

lazy val example: Parsley[String] = chain.postfix(
  empty | (empty.map((_ + _).curried) | empty <*> example) <*> string("a")
    | string("b") | empty
)(
  (empty.map(flip) <*> example | pure(identity).map(compose((_ + _).curried)))
    .map(flip) <*> string("a")
    | empty | empty
)
```

Fig. 1.2: The initial attempt at factoring out left-recursion from the `example` parser.

This is... disappointing, to say the least. There are *many* things wrong with the transformed output:

- The parser is horrendously complex and unreadable, its intent entirely obfuscated in a sea of combinators. It's especially frustrating that there are so many `empty` combinators, when both `p | empty` and `empty | p` are just equivalent to `p`.
- Having to define the `flip` and `compose` functions is not ideal, but inlining them as lambdas would make the code even worse.
- Even worse, the parser does not even typecheck – unlike classical Hindley-Milner-based type systems, Scala only has *local* type inference [Cremet et al. 2006]. As a result, the compiler is unable to correctly infer correct types for `flip` and also asks for explicit type annotations in the lambda `(_ + _).curried`.

The result is discouraging especially because it is not impossible to factor out the left-recursion in a nice manner: a hand-written equivalent using `postfix` would resemble the concisely defined parser in fig. 1.3. There is still hope, though – if the `empty` combinators can be removed and something is done about the higher-order functions, perhaps fig. 1.2 could be salvaged into something that looks more like the human-written version.

```
lazy val example: Parsley[String] = chain.postfix(string("b"))(string("a").as(_ + "a"))
```

Fig. 1.3: An idiomatic way to express the `example` parser using `chain.postfix`.