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 ⟨*expr*⟩ 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 ⟨*expr*⟩ by first trying to parse ⟨*expr*⟩, 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 encapsulate 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:

```scala
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].

```
val result: Parsley[A] = results match {
  case None    => empty
  case Some(x) => pure(x)
}
val transformed = 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 AST of a Scala source file 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, in a similar manner 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 <*> (pronunced "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:

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

```
val ap = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

def deconstructAp(parser: Term) = parser match {
  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 not relevant 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.

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)
```

Identifying Named Parsers

Finding AST nodes corresponding to the definition sites of named parsers involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be some `Parsley[_]`. This type information is accessed by querying the Scalafix semantic API for 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.

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 can't be further inspected
    case Term.Apply(matcher(_), Term.ArgClause(List(expr), _)) => Pure(expr)
  }
}
```

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:

```scala
// 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
  }
}

val nonTerminals: Map[Symbol, ParserDefn] = 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:

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

This term can then be pretty-printed into a string, and applied as a Scalafix patch.

1.3 Implementing the Left-Recursion Transformation

TODO

1.3.1 Defining Utility Functions

The transformation requires the use of three higher-order functions:

- The identity function `identity[A]: A => A` is defined in the standard library.
- The flip function reverses the order of arguments applied to a function. This isn't 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.

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))
```

This brings these higher-order functions into scope, allowing the transformed code to make use of it.

1.3.2 Core Combinators

Core combinators: NonTerminal, Pure, Empty, <*>, <|>. Character combinators (e.g. string, char, item) are grouped as one as they behave the same. Some composite combinators are supported, and desugared into the core combinators.

Can derive empty string? (good resource from packrat parsing paper) pure(x) – yes, semantic action is x empty – no p <|> q – if p or q can derive empty, peg is ordered so semantic action is pe if it can derive empty, else qe p <*> q – if p and q can derive empty, semantic action is pe(qe) due to pure(f) <*> pure(x) == pure(f(x)) law string – only if given argument "", but this also illegal in parsely – explicitly triggers a runtime error, so basically no [error] java.lang.IllegalArgumentException: requirement failed: ‘string‘ may not be passed the empty string (‘string("")‘ is meaningless, perhaps you meant ‘pure("")‘?) char – no (implemented) item – no (not implemented as a core comb, should do that?) many(p) – yes (not implemented properly?) – semantic action is empty list?? some(p) – if p can derive empty (not a primitive, defined in terms of many – p <::> many(p)) NT – if referenced rule can derive empty

Top-level, call unfold on each non-terminal:

```
def unfold(env: Map[Symbol, ParserDefinition], currentNT: Symbol): UnfoldedParser =
  env(currentNT).parser.unfold(currentNT, env, visited = Set.empty)
```

Non-terminals

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

Ap (<*>)

Choice (<|>)

The remainder

1.3.3 Composite Combinators

The unfold method is defined for every single combinator in the Parser ADT.

Special case of right-factoring in reverse, holds unidirectionally.

```
(p <|> pure f) <*> q   ⇒   (p <*> q) <|> (pure f <*> q)
```

```
(p | pure(f)) <*> q = (p <*> q) | q.map(f)
```

Most important is the <*> combinator, which parsley-garnish uses as the primitive combinator for composing parsers. * When can <*> derive empty? If both p and q can derive empty, then its semantic action is pe(qe) due to the pure(f) <*> pure(x) == pure(f(x)) law. * for nonleftrec and leftrec: if p can accept empty, q is changed to be subjected to the semantic value corresponding to the first empty parser * leftrec: flip re-associates semantic actions back to the left * leftrec: function composition required since type is A => A

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

    val result = if (pe.isDefined && qe.isDefined) Some(q"${pe.get}(${qe.get})") else None
    val nonLefts = {
      val lnl = pn <*> q
      val rnl = pe.map(q"f => qn.map(f)").getOrElse(Empty)
      lnl | rnl
    }
    val lefts = {
      val llr = pl.map(q"flip") <*> q
      val rlr = pe.map(q"ql.map(compose)").getOrElse(Empty)
      llr | rlr
    }

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

1.3.4 Success...?

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 p | empty and empty | p are both actually 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`.

This 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`.