Chapter 1
Using the New Parser ast

The ability to simplify parsers, as well as normalising their expression arguments, makes it feasible to use the intermediate `Parser` ast for high-level parser transformations. This chapter first revisits the left-recursion factoring rule from **??**, demonstrating how parser simplification drastically improves the quality of the rule. This is followed by a brief exploration of other rules that can be trivially implemented on top of the intermediate ast, rather than the generic Scalameta ast.

1.1 Removing Left-Recursion: Revisited

?? left off with a disappointing result for the left-recursion factorisation transformation, producing a mostly unreadable output in **??**. The transformation can now be revisited given the improvements made to the `Parser` ast. This section briefly explores some of the changes made to the rule, and the final result of the transformation on the `example` parser.

1.1.1 Pretty Parsers

The need to obtain simpler parser terms was the main driving motivation behind the new `Parser` ast, which now defines the following methods on top of `simplify`:

- `normalise`: Simplifies parsers via parser laws and normalises their expression arguments. This is used to check parser equivalence using their canonical forms.
- `prettify`: Adds an extra step after normalisation, using a set of "resugaring" rules to rebuild the combinators that were desugared during the left-recursion unfolding.

After the left-recursion transformation, `prettify` is used to make parser terms more readable, before applying the final Scalafix patch. The following demonstrates the implementation of these methods, as well as a subset of the resugaring rewrite rules:

```
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
    // 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
    // Replace with explicit 'string' combinator so the compiler does not complain
    case FMap(Str(s, _), f) => FMap(Str(s, implicitSyntax = false), f)
  }
}
```

1.1.2 Evaluating Expressions

The higher-order functions `flip` and `compose` are represented as in **??**, eliminating the need to patch in their definitions as Scala code. Instead, they get partially evaluated as lambda expressions during normalisation.

The currying of functions passed to `LiftLike` combinators are also expressed with the `Expr` ast, to ensure that they are well-formed during the transformation. This also significantly reduces the occurrence of `.curried` method calls in the transformed output, which helps Scala's type inference and reduces syntactic noise in the final output.

```
trait Expr {
  // Curries a lambda expression, flattening into a chain of unary functions
  def curried: Expr = this match {
    case Abs(x, f)        => Abs(x, f.curried)
    case AbsN(x :: xs, f) => Abs(x, AbsN(xs, f).curried)
    case _                => this
  }
}

trait LiftLike extends Parser {
  def unfold() = {
    val curriedFunc = Pure(func match {
      // If the Expr is still opaque, it is treated as before
      case Translucent(term, env) => Translucent(q"($term).curried", env)
      // If statically inspectable, curry the lambda expr properly
      case _ => func.curried
    })
    parsers.foldLeft(curriedFunc)(_ <*> _).unfold
  }
}
```

1.1.3 The Improved Rule

Being able to normalise parsers lets `parsley-garnish` make more informed decisions about the result of the transformation. By normalising the `leftRec` portion of the unfolded parser, it can be determined whether the transformation was successful or not:

- If `leftRec` resolves to `empty`, the original parser was not left-recursive, so `parsley-garnish` does not bother rewriting it.
- If it simplifies to a `pure(x)`, this indicates a deeper problem which rewriting into the `postfix` combinator cannot solve. This generates a lint error diagnostic instead of an auto-fix.
- Otherwise, the parser is left-recursive and can be fixed using the `postfix` combinator. This result is prettified before applying the final patch.

These cases are implemented in the `transform` method to generate the a Scalafix patch per unfolded parser:

```
def transform(results: Option[Expr], nonLeftRec: Parser, leftRec: Parser) = {
  val result = results match {
    case None    => Empty
    case Some(t) => Pure(t)
  }

  leftRec.normalise match {
    case Empty   => Patch.empty // not left-recursive, do not rewrite
    case Pure(_) => Patch.lint(LeftRecDerivesEmptyLint) // left-recursive but unfixable
    case _ =>
      val transformed = Postfix(nonLeftRec | results, leftRec).prettify
      Patch.replaceTree(originalTree, transformed.toString)
  }
}
```

The Final Result

Finally, the `example` parser from **??** can be automatically transformed into the following form:

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

This is a *significant* improvement over the original output in **??**, and is visually very similar to the "optimal" hand-written version in **??**. In fact, the two parsers can be proven to be equivalent – **??** explores this further with some different examples.

1.2 Simplify Parsers

The improved `Parser` AST also grants a new auto-fix rule for free: automatic simplification of parsers. At a high-level, the rule works as follows:

- For each parser, get its prettified form and compare it to the original.
- If the result is different, apply the simplified version as a code rewrite.

The implementation of the entire rule is very straightforward, and 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
  }
}
```

This rule therefore lints for overly complex parsers that can be automatically simplified with parser laws. It is unclear how useful this rule would be in practice, but it demonstrates the power of the new `Parser` AST in enabling high-level transformations, and is gained with very minimal effort. For example:

```
/* Before */  val anise = pure("basil").map(add(_, "coriander"))
/* After */   val anise = pure(add("basil", "coriander"))
```

1.3 Avoid Parser Redefinitions

Another rule that can be easily implemented with the new `Parser` AST is to catch cases where a user manually writes out a parser that is already defined in the library. This is similar in spirit to the previous rule, but makes use of the new `rewrite` and `transform` methods on parsers introduced in **??**, which makes it incredibly convenient to write syntax-directed rewrite rules on parsers. Expressing high-level rewrites is much less error-prone than working with the Scalameta AST directly, and prevents re-duplication of efforts. As an example, consider the following rewrite:

```
parser.rewrite { case ManyP(p <~ sep) => EndBy(p, sep) }
```

This catches a case where a user writes a parser in the form of `many(p <~ sep)`, which is already defined in `parsley` as `endBy(p, sep)`. Compare this to the equivalent pattern match if utilising the low-level AST:

```
tree.collect {
  case Term.Apply.After_4_6_0(
    manyMatcher(_),
    Term.ArgClause(
```

```
      List(Term.ApplyInfix.After_4_6_0(
        Term.Name("p"),
        Term.Name("<~"),
        Type.ArgClause(Nil),
        Term.ArgClause(List(Term.Name("sep")), None)
    )), None)) => q"endBy(p, sep)"
}
```

This rule is likely to be much more useful in practice than *Simplify Parsers*, serving as a pedagogical tool to guide users towards using the library's higher-level combinators.

Summary

This chapter demonstrates how the left-recursion factoring rule is made feasible with the simplification machinery in the new `Parser` AST. It also introduces two rules, *Simplify Parsers* and *Avoid Parser Redefinitions*, that are trivial to implement on top of the new `Parser` AST, showcasing the power of the new high-level AST in expressing transformations in a more maintainable and readable way. The improved `Parser` AST machinery makes it easy to extend `parsley-garnish` with more complex rules on top of a parser abstraction, without having to worry about manually simplifying its output. Future work could explore more complex rules, such as detecting ambiguous parsers or conversely, overuse of the `atomic` combinator.