Chapter 1
Removing Left-Recursion

1.1 Implementation

section intro

Running example The following left-recursive parser and its transformation into its `postfix` form will serve as a running example:

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

1.1.1 The Need for 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 clumsier to perform 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 *ap* combinator `<*>`. 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 {
  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

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

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 Ap(p: Parser, q: Parser) extends Parser
case class Choice(p: Parser, q: Parser) extends Parser
```

Fig. 1.1: A subset of the core combinators in the Parser ADT.

Deconstructing parsers Syntactic sugar for deconstructing parsers is provided by unapply methods on symbolic extractor objects. This makes pattern matching on parsers feel more natural:

```
object <*> {
  def unapply(parser: Ap): Option[(Parser, Parser)] = Some((parser.p, parser.q))
}

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

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

```
implicit class ParserOps(private val p: Parser) extends AnyVal {
  def <*>(q: Parser): Parser = Ap(p, q)
  def <|>(q: Parser): Parser = Choice(p, q)
  def map(f: Term): Parser = FMap(p, f)
}

implicit class MultiParserOps(private val ps: List[Parser]) extends AnyVal {
  def zipped(f: Term): Parser = Zipped(f, ps)
}
```

This makes working with Parser terms feel closer 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 exNT = NonTerminal(Sym("path/to/package/ObjectName.example."))

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

1.1.2 Lifting to the Intermediate Parser AST

Converting the raw Scala AST to the intermediate 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, as 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 some extra meta-information required for the transformation.

Most importantly, this meta-information includes a reference to a parser's original node in the Scala AST, so that any lint diagnostics or code rewrites can be applied to the correct location in the source file. This is simply defined as:

```
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 of the inferred type of the variable. Therefore, this field will not always be present, so in the general case, the type must be queried via a symbol information lookup like so:

```
tree match {
  case Defn.Val(_, List(Pat.Var(varName)), _, body) =>
    println(s"qualified symbol = ${varName.symbol}")
    varName.symbol.info.get.signature match {
      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 ugly 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 §1.1.1 makes a reappearance in the definition of `Ap.fromTerm`, where the arguments to the `<*>` combinator are recursively lifted to `Parser` objects:

```
// Type signatures in Parsley:
// p: Parsley[A => B], q: =>Parsley[A], p <*> q: Parsley[B]
case class Ap(p: Parser, q: Parser) extends Parser
object Ap {
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

  def fromTerm: PartialFunction[Term, Ap] = {
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
      Ap(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:

```
// 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] = {
    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:

```
object VariableDecl {
  def unapply(tree: Tree): ParserDefn = tree match {
    case Defn.Val(_, List(Pat.Var(varName)), _, body) if isParsleyType(varName) =>
      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 {
  case VariableDecl(parserDef) => parserDefn.name.symbol -> parserDef
}.toMap
```

1.1.3 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.

```
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.

1.1.4 Implementing the Left-Recursion Transformation

TODO

Success...?

Thus, running the transformation on the `example` parser yields the output in fig. 1.2. This is disappointing, to say the least. There are *many* things wrong with the transformed output:

- This output is horrendously complex and unreadable. The intent of the parser is entirely obfuscated in a sea of combinators.
- Having to define the `flip` and `compose` functions is not ideal, but inlining them as lambdas would make the code even worse.
- The parser does not even typecheck – unlike classical Hindley-Milner-based type systems, Scala only supports 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`.

```
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.