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 AST of the source program. However, this AST is a general-purpose representation that becomes hard to work with when trying to do domain-specific manipulations on grammars.

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 reverse 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 the quasiquote 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 deconstruct(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` and `Term.ArgClause`, which are low-level details not relevant to the task of manipulating parsers. This is not an issue for simple one-off transformations, but 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

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

AST representation that is more specialised to the domain of parser combinators. the past 3 sentences all start with "this", reword them

The Parser ADT

parsley-garnish uses a similar deep-embedded parser representation for the intermediate AST as Baars and Swierstra [2004], extended to match parsley's combinators. Fig. 1.1 shows how this is implemented as an algebraic data type (ADT), with extra syntactic sugar introduced by implementing unapply methods in extractor objects.

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

case class Choice(p: Parser, q: Parser) extends Parser
object <|> {
  def unapply(parser: Choice): Option[(Parser, Parser)] = Some((parser.p, parser.q))
}
```

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

All Parser types represent parsley combinators, with the exception of NonTerminal to represent a reference to a named parser. Inspecting parsers is now easily done by pattern matching on constructors and/or using the extractor objects:

```
def deconstruct(parser: Parser) = parser match {
  case Ap(p, q) => (p, q) // using constructor
  case p <|> q  => (p, q) // using extractor object
}
```

As an example, the example parser is then represented as a Parser object resembling the following (where quasiquote notation is used to keep the lambda expression term q"_ + _" concise):

```
// (example, string("a")).zipped(_ + _) | string("b")
Choice(
  Zipped(
    q"_ + _",
    List(
      NonTerminal(Sym("path/to/package/ObjectName.example.")),
      Str("a")
    )
  ),
  Str("b")
)
```

1.1.2 Lifting to the Intermediate Parser AST

Converting the raw Scala AST to the intermediate AST therefore 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 Implementing the Left-Recursion Transformation