0.1 Parser Representation

This is an INTERMEDIATE SYMBOLIC REPRESENTATION (?) more specialised than general-purpose scala ast
This section is about simplifying in our semantic domain (parsers)

Several of the more complex lint rules, most notably **??**, require manipulating parser combinators in a high-level manner. Scalafix works with code represented as data, in the form of a AST-like representation i.e. Scalameta `Tree` nodes. It would be possible, but extremely cumbersome, to work directly with these AST nodes.

Left-recursion factoring is based on lawful transformations and rearrangements of parsers. There are many instances where given two parsers `p` and `q`, the transformation requires combining them with another combinator, such as `<*>`. We can use quasiquotes `q"p <*> q"` which expands to a `Term` node resembling `ApplyInfix(lhs = p, op = Term("<*>"), args = q)` (in pseudocode for readability purposes). This is ok in terms of ergonomics, but we lose the static inspectability of the individual parsers `p` and `q`.

Instead, represent parsers as an algebraic data type ADT in the same way that Parsley itself uses a deep embedding to represent combinators as objects. Methods on these objects can then be used to manipulate them, and the resulting object can still be pattern matched, maintaining the static inspectability of the parsers. So then it's just like writing parsers in Parsley itself: `p <*> q` constructs a `Ap(p, q)` node which can still be pattern matched on. And similar to Parsley, representing everything as objects makes it easy to optimise using pattern matching on constructors. This representation also then gives us for free the implementation for lint rules such as *Simplify Complex Parsers* rule, which applies parser laws to simplify parsers.

This section explores the motivation behind this and the design choices made in the implementation. Use the **??** rule as a basis/context to demonstrate the utility of this representation.

Running example to motivate all requirements for the parser representation – removing left recursion from the following simple parser:

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

0.1.1 Detecting Top-Level Parsers

Each source file is represented by Scalafix as a large AST. `parsley-garnish` would benefit from a higher-level abstraction. A more useful representation for `parsley-garnish` is a map of all parsers defined within that file, indexed by the symbol of the definition.

To achieve this, it is necessary to identify the AST nodes of interest corresponding to Parsley parsers. This involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be of form `Parsley[_]`. Consider `expr`, which is a top-level definition of a parser:

```
lazy val expr: Parsley[String] = (expr, string("a")).zipped(_ + _) | string("b")
//       ^^^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//         key: expr.symbol          RHS of definition: convert to a Parser
```

The right hand side of the definition is a Scalameta `Term` node representing the definition of the parser. During traversal, the right hand side of the definition is converted to a `Parser` instance. This is then accumulated in a map, with the symbol of the definition as the key. A full traversal through a source file obtains all the non-terminals in the grammar defined within that file.

0.1.2 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 `Term` to determine which parser combinator it represents, and then constructing the appropriate `Parser` instance.

Each Parser defines a partial function `fromTerm` which creates an instance of that parser from the appropriate Scalameta `Term`. These `fromTerm` methods define a `toParser` extension method on Scalameta `Term`s to transform them to the appropriate parser. Use Scalafix's `SymbolMatcher` to match tree nodes that resolve to a specific set of symbols. This makes use of semantic information from SemanticDB, so we are sure that a `<*>` is actually within the `parsley.Parsley` package, rather than some other function with the same name. This is much more robust compared to HLint, which suffers from false positives due to its reliance on syntactic information only.

The top-level combinator that makes up `expr`'s definition is the choice combinator, `|`. Taking a look at the Scalameta `Term` representing this definition `(expr, string("a")).zipped(_ + _) | string("b")` (this is cleaned up to hide the large term representing the LHS of the choice):

```
Term.ApplyInfix(
  Term.Apply(...), // (expr, string("a")).zipped(_ + _)
  Term.Name("|"),
  Type.ArgClause(List()),
  Term.ArgClause(
    List(
      Term.Apply(
        Term.Name("string"),
        Term.ArgClause(List(Lit.String("b")), None)
      )
    ),
    None
  )
)
```

The goal is to pattern match on this term to construct a `Choice` node, which represents the choice combinator. Thus, `Choice.fromTerm` should be defined to match on an infix application on a `|` (or `<|>`) combinator:

```
object Choice {
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`|`", "parsley.Parsley.`<|>`")

  def fromTerm(implicit doc: SemanticDocument): PartialFunction[Term, Choice] = {
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
      Choice(p.toParser, q.toParser)
  }
}
```

If it finds such a term, it constructs a `Choice` node, and recurses on `p` and `q` to also convert them to parser nodes. After all sub-terms also get converted, the `expr` parser is represented as:

```
Choice(
  Zipped(Function(_ + _), List(NonTerminal(expr), Str(a))),
  Str(b)
)
```

For now we are only concerned with the parser representation; we will touch upon how `Function`s are represented in the next **??**.

0.1.3 Building New Parsers From Existing Parsers

Now that we have parsers represented as an ADT, we can easily build new parsers from existing parsers. This is crucial for the left-recursion factoring rule, which "unfolds" parsers into separate parsers representing the left-recursive and non-left-recursive parts. These are then recombined to form parsers which are free from left recursion.

Make this even easier by utilising Scala's ability to define infix operators, define them as extension methods on the `Parser` trait. For example:

```
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: Function): Parser = FMap(p, f)
}
```

This makes it more ergonomic to manipulate parsers, it's like we're writing Parsley code itself. A small example snippet from the `unfold` method on the `Ap` parser:

```
val lefts = {
  val llr = pl.map(flip) <*> q
  val rlr = pe.map(f => ql.map(composeH(f))).getOrElse(Empty)
  llr <|> rlr
}
```

Notice how the code closely resembles the high-level description of the transformation, using `<*>`, `<|>`, `map`, operators.

0.1.4 Simplifying Parsers Using Parser Laws

Once all the unfolded parsers have been recombined, the raw output is very noisy and difficult to read. Again, ignore the functions, these will be covered in **??**.

```
lazy val expr: Parsley[String] = chain.postfix(
  empty | (empty.map(a => b => a + b) | empty <*> expr) <*> string("a")
    | string("b") | empty
)(
  (empty.map(FLIP) <*> expr | pure(ID).map(COMPOSE(a => b => a + b)))
    .map(FLIP) <*> string("a")
    | empty | empty
)
```

This is obviously unacceptable and completely obfuscates the intent of the parser. For human readability of the transformed output, it is therefore important to simplify the parser as much as possible. Now that the parsers are represented as objects, it is easy to pattern match on their constructors. This improved static inspectability allows us to perform simplifications using the laws that govern parser combinators – these often form a natural

simplification in one direction. This is similar to the high-level optimisations performed in the Parsley backend as described by Willis, Wu, and Pickering [2020], using the same parser laws.

Fig. 1 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of the laws in fig. 1 have already been shown to hold for Parsley in [Willis and Wu 2018]; an additional proof for eq. (8) can be found in **??**.

```
p.map(f).map(g) = p.map(g compose f)                (1)
  pure(f) <*> pure(x) = pure(f(x))                  (2)
      pure(f) <*> x = x.map(f)                      (3)
           empty | u = u                            (4)
           u | empty = u                            (5)
        pure(x) | u = pure(x)                       (6)
         empty <*> u = empty                        (7)
         empty.map(f) = empty                       (8)
```

Fig. 1: Functor (1), Applicative (2, 3), and Alternative (4–8) laws.

In the previous example, it is evident that the most noise results from the `empty` combinators. These can be eliminated using eqs. (4), (5), (7), and (8):

```
lazy val expr: Parsley[String] = chain.postfix(string("b"))(
  (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
)
```

The complicated term in the postfix operator can then be simplified as follows:

```
    (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
=       { eqs. (2) and (3) }
    pure(COMPOSE(a => b => a + b)(ID)).map(FLIP) <*> string("a")
=       { eqs. (2) and (3) }
    pure(FLIP(COMPOSE(a => b => a + b)(ID))) <*> string("a")
=       { eq. (3) }
    string("a").map(FLIP(COMPOSE(a => b => a + b)(ID)))
```

This results in the most simplified form of the parser:

```
val f: Function = FLIP(COMPOSE(a => b => a + b)(ID))
lazy val expr: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

Implementation

These simplifications are applied akin to peephole optimisations in a bottom-up traversal of the recursive `Parser` ADT. There are many instances of Parsers, so this leads to a lot of boilerplate code recursing through each case, which is prone to error when we implement it. To avoid this, we decouple the application of a generic transformation function from the recursive traversal through the datatype. This is still a hand-written traversal, but heavily inspired by the generic traversal patterns in [Mitchell and Runciman 2007]. Bottom-up transformation

takes a partial function, applying the transformation at nodes where it is defined. The resulting `transform` method on parsers resembles the following (only a few cases shown for brevity):

```
def transform(pf: PartialFunction[Parser, Parser]): Parser = {
  val p = this match {
    case Ap(p, q)      => Ap(p.transform(pf), q.transform(pf))
    case Zipped(f, ps) => Zipped(f, ps.map(_.transform(pf)))
    case Pure(f)       => Pure(f)
    ...
  }
  if (pf.isDefinedAt(p)) pf(p) else p
}
```

There is also a need for a `rewrite` method to apply a transformation exhaustively until a normal form is reached. This is implemented in terms of `transform`, applying the partial function everywhere and re-applying it until it no longer makes a change.

```
def rewrite(pf: PartialFunction[Parser, Parser]): Parser = {
  def pf0(p: Parser) = if (pf.isDefinedAt(p)) pf(p).rewrite(pf) else p
  this.transform(pf0)
}
```

Therefore, any transformation on parsers can be defined without having to worry about any recursive traversal boilerplate. Using `rewrite`, parser simplification can then be expressed in a clean and maintainable manner:

```
def simplify: Parser = this.rewrite {
  // p.map(f).map(g) == p.map(g compose f)
  case FMap(FMap(p, f), g) => FMap(p, composeH(g, f))
  // u <|> empty == u
  case Choice(u, Empty) => u
  // pure(f) <|> u == pure(f)
  case Choice(Pure(f), _) => Pure(f)
  ...
}
```

Additionally, the `Parser` trait is sealed, so there will be compiler warnings if a new case is added and the `transform` method is not updated. Overall, this approach still requires a hand-written traversal so it is more error-prone than a generic derivation. However, that would require usage of an external library such as `shapeless`[1], which is not desired as the complexity of the ADT is not high enough to warrant bringing in an extra dependency just for this purpose.

0.1.5 Converting Parsers Back to Scalameta Terms

After the transformations on parsers are complete, they need to be converted back to a textual representation to be applied as a Scalafix patch. It is actually rather trivial to do so, by borrowing the pretty-printing capabilities of Scalameta terms. This transformation is thus the inverse of the `fromTerm` transformation. This can be written using Scalameta quasiquotes to construct the `Term` nodes. The `Parser` trait defines this transformation as the method `term`, for example:

[1] https://github.com/milessabin/shapeless

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