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)

Scalafix runs at the meta-level, outside of the phase distinction of compile- and run-time. Staged metaprogramming applies optimisations at compile-time, whereas these "optimisations" at applied post-compilation

For example, given two AST nodes `Term.Name("p")` and `Term.Name("q")` corresponding to named parsers `p` and `q`, suppose a transformation involves combining them with the *ap* combinator `<*>`. One may consider using quasiquotes to achieve this: `q"p <*> q"` would automatically expand to `Term.ApplyInfix(Term.Name("p"), Term.Name("<*>"), Type.ArgClause(Nil), Term.ArgClause(List(Term.Name("q")), None))`. However, this loses the static inspectability of the individual parsers `p` and `q` – although quasiquotes can be used as extractor patterns to recover the original AST nodes, their usage as such is discouraged as they can easily result in unintended match errors. The recommended approach is to pattern match on the AST nodes directly, which is obviously unergonomic even for this small example: to extract the RHS term `q`, one would have to perform a nested pattern match on the `Term.ApplyInfix` term and its `Term.ArgClause` node representing the arguments of the infix function application.

It is hopefully obvious that this would a very painful process for the rule author. It would be desirable to abstract away from the low-level syntactic AST representation, and instead treat these AST nodes as what they semantically represent – parsers.

Instead, fig. 1 shows how parser terms can be represented as an algebraic data type ADT, in the same way `parsley` itself uses a deep embedding to represent parsers as pure data objects. The reasoning behind this approach is the same as that for PARSLEY – this representation allows parsers to be easily inspected and analysed via pattern matching on constructors.

```
trait Parser
case class NonTerminal(ref: Symbol) extends Parser
case class Pure(f: Function) extends Parser
case object Empty extends Parser
case class Choice(p: Parser, q: Parser) extends Parser
case class Ap(p: Parser, q: Parser) extends Parser
...
```

Fig. 1: A subset of the core combinators in the `Parser` ADT.

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

Consider the following left-recursive parser:

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

The transformations necessary to remove left recursion from this parser (**??**), will serve as a running example to motivate the requirements for the parser representation.

0.1.1 Detecting Named Parsers

Before any analysis on parsers can be performed, it is first necessary to identify which AST nodes correspond to parsers. `parsley-garnish` builds a map of all parsers defined within a source file, indexed by the unique symbol of its name.

Identifying these AST nodes of interest involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be some `Parsley[_]` – this information is accessed by querying the Scalafix semantic API for the node's symbol information. Consider the labelled `ast` structure of the `expr` parser:

```
Defn.Val(
  mods = List(Mod.Lazy()),
  pats = List(Pat.Var(Term.Name("expr"))),
  decltpe = Some(
    Type.Apply(Type.Name("Parsley"), Type.ArgClause(List(Type.Name("String"))))
  ),
  rhs = Term.ApplyInfix(...)
)
```

The qualified symbol `expr` is used as the key in the map, and the `rhs` term is lifted the intermediate parser representation for analysis. A reference to the original AST node is also kept so any lint diagnostics or code rewrites can be applied to the correct location in the source file. Thus, a full traversal through the source file builds a map of all named parsers, representing all 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 `scala.meta.Term`. These `fromTerm` methods are combined to define a `toParser` extension method on `scala.meta.Term` – this is where AST nodes are lifted to their corresponding `Parser` representation.

The top-level combinator that makes up `expr`'s definition is the choice combinator, `|`. Scalameta represents this infix application of the `|` operator as so:

```
Term.ApplyInfix(
  lhs = Term.Apply(...), // AST node for (expr, string("a")).zipped(_ + _)
  op = Term.Name("|"),
  targClause = Type.ArgClause(List()),
  argClause = Term.ArgClause(
    List(
      Term.Apply(
        Term.Name("string"),
        Term.ArgClause(List(Lit.String("b")), None)
      )
    ),
    None
  )
)
```

This structure therefore guides the implementation of the pattern match in `Choice.fromTerm`:

```
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)
  }
}
```

The definition of this method is fairly self-explanatory: it matches on a `ApplyInfix` term where the operator is the `|` combinator, and recursively applies `toParser` to its LHS and RHS nodes. Finishing off, the `expr` parser is therefore converted to the following `Parser` instance:

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

For now, the exact representation of the `Function` is not important – this is covered in the next **??**. The remaining code snippets in this section will simplify and grey out the function representations for brevity.

0.1.3 Building New Parsers From Existing Parsers

Now that raw AST terms can be lifted to the higher-level parser representation, it is easy to build new parsers from existing parsers. This is crucial for left-recursion factoring, 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.

Smart constructors are used to make manipulating parser terms resemble writing `parsley` code itself. These are defined as infix operators, which are written as extension methods on the `Parser` trait:

```
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)
}
```

Parser terms can now be manipulated in a manner that looks almost indistinguishable from writing `parsley` code. For example, the `unfold` method on the `Ap` parser contains this snippet, where `pl`, `ql`, and `q` are parsers (`pe` is not a parser, but rather an `Option` value):

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

Other than the capitalised `Empty` constructor, this would be perfectly valid `parsley` code.

0.1.4 Simplifying Parsers Using Parser Laws

After left-recursion factoring is complete, the recombined parsers from the unfolding process are very noisy and difficult to read. Consider the resulting parser from factoring out the left-recursion in `expr`:

```
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(Identity).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. 2 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of the laws in fig. 2 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. 2: 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:

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

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