0.1 Simplifying Parsers

This is where the deep embedding approach comes to shine: simplifications are easily expressed by pattern matching on `Parser` constructors.

- `parsley` performs rewrites on the parser AST to produce more optimised *code*.
- `parsley-garnish` performs rewrites on the parser AST to produce a more readable *textual representation of code*.

0.1.1 Parser Laws

Willis, Wu, and Pickering [2020] note that parser combinators are subject to *parser laws*, which often form a natural simplification in one direction. Both `parsley` Scala [Willis and Wu 2018] and `parsley` Haskell [Willis 2023] use these laws as the basis for high-level optimisations to simplify the structure of deeply-embedded parsers. These same principles can be used by `parsley-garnish` to simplify parser terms to be more human-readable.

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 by 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.

Simplifying the Example Parser

This section provides a worked example of how the parser in **??** is simplified using parser laws. Most of the noise in **??** comes from the large number of `empty` combinators. These can be eliminated using eqs. (4), (5), (7), and (8):

```
lazy val expr: Parsley[String] = chain.postfix(string("b"))(
  (pure(identity).map(compose((_ + _).curried))).map(flip) <*> string("a")
)
```

This already looks a lot better, but the second parameter to `postfix` can be further simplified as follows:

```
    (pure(identity).map(compose((_ + _).curried))).map(flip) <*> string("a")
=       { eqs. (2) and (3) }
    pure(compose((_ + _).curried)(identity)).map(flip) <*> string("a")
=       { eqs. (2) and (3) }
    pure(flip(compose((_ + _).curried)(identity))) <*> string("a")
```

```
=      { eq. (3) }
    string("a").map(flip(compose((_ + _).curried)(identity)))
```

The most simplified form of the parser is then:

```
val f = flip(compose((_ + _).curried)(identity))
lazy val expr: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

The parser has now been expressed in a much simplified form, in a similar style to how it would be written by hand. The remaining challenge is to simplify the contents of the expression `f`, which is tackled in **??**.

0.1.2 Implementing Rewrites on the Parser AST

Lawful simplifications are applied by a bottom-up transformation over the recursively defined `Parser` AST. Since there are many parser cases, this inevitably leads to repetitive and error-prone boilerplate code which simply exists to recursively propagate the transformation through each case. To avoid this, the recursive traversal itself can be decoupled from the definition of the transformation function. Although the traversal is still hand-written, this implementation is inspired by the generic traversal patterns offered by Haskell's `uniplate` library [Mitchell and Runciman 2007].

The traversal is realised as a `transform` method on the `Parser` trait, which takes a partial function and applies it to nodes where it is defined. The transformation is applied via a bottom-up traversal:

```
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
}
```

A `rewrite` method can then be defined in terms of `transform`, applying the partial function everywhere and re-applying it until it no longer makes a change. This has the effect of applying a transformation exhaustively until a normal form is reached.

```
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 recursion boilerplate: the act of traversal itself is fully abstracted away and encapsulated within the `transform` method. 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, compose(g, f))
  // pure(f) <*> pure(x) == pure(f(x))
  case Pure(f) <*> Pure(x) => Pure(App(f, x))
  // u <|> empty == u
  case u <|> Empty => u
```

```
  // pure(f) <|> u == pure(f)
  case Pure(f) <|> _ => Pure(f)
  ...
}
```

Extensibility and Safety Further design considerations are made to ensure the extensibility of this approach: the `Parser` trait is sealed, which enables compiler warnings if a new `Parser` case is added and the `transform` method is not updated. Although this formulation of the traversal is inspired by generic traversals, it still manually defines the traversal for each case: a safer approach would be to generically derive this. In Scala, this would require the use of an external dependency such as `shapeless`[1], which is frankly overkill given the relative simplicity of the `Parser` ADT.

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