Parser Core

We require ApplicativeDo, DeriveFunctor, and LambdaCase for this module; while DeriveFunctor and LambdaCase is just for convenience, ApplicativeDo is a pretty major powerhouse. Parsing is an applicative action, and ApplicativeDo allows us to use do-notation for parsing actions instead of needing to define a monad instance.

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE LambdaCase #-}

module Zl.Parser.Core where
import Control.Applicative
```

The Parser data type

The following line is the core data type that powers all parsing:

```
data Parser a = Parser { runParse :: String -> [(a, String)] } deriving (Functor)
```

All it is is a simple wrapper around a `String -> [(a, String)]` function - any parser is a function that takes a string, and returns a list of possible matches, where each match is a tuple containing a matched object, and the rest of the string that hasn't been matched yet. An empty list output means that the parser could not get any matches.

Here's an example of the most primitive parser we can get: the following function takes a char, and creates a Parser that matches against that char.

```
char :: Char -> Parser Char
char x = Parser $
  \case (c:cs) | x == c -> [(c, cs)]
        _               -> []
```

Typeclass instances

It turns out that this definition satisfies a number of useful properties that allow it to be legal Haskell/mathematical constructs. These constructs are the core plumbing in how we chain parsers together.

For instance, the Alternative type class allows us to chain parsers together in a `or` like fashion - we define `f <|> g` to return a new parser that attempts to parse `f`, but if that fails, trys to parse `g` instead.

```
instance Alternative Parser where
    empty = Parser $ pure []
    f <|> g = Parser $ \s ->
      case runParse f s of []  -> runParse g s
                           res -> res
```

Alternative also gives us `some` and `many` for free: they match one or more, or zero or more, of a given parser respectively.

The Applicative type class is the big thing powering sequencing parsers together. Though it may not be obvious in its definition, it allows us to write `f *> g` to create a parser that parses f, and then parses g from the rest of the string after f's parse. In general, this is a very flexible set of operations that allows us to do almost any other chaining action with ease.

```
instance Applicative Parser where
    pure x = Parser $ \s -> [(x, s)]
    p <*> q = Parser $ \s -> do
        (f, s1) <- runParse p s
        (a, s2) <- runParse q s1
        pure $ (f a, s2)
```

We can also make a monoid out of this. There are a number of possible legal ways to make Parsers into a monoid, but since some of them (like sequencing or alternating) have been already covered, I will make the monoid instance into something that combines the result of two parsers (i.e. it will make the rest of the parsing sequence after those two run twice, once for the result of each parser).

```
instance Semigroup (Parser a) where
    f <> g = Parser $ \s -> runParse f s <> runParse g s
instance Monoid (Parser a) where
    mempty = empty
```

Finally, don't forget we got a Functor instance for free from DeriveFunctor. This lets us modify the return type of a Parser.

Parsing Combinators

Now that we have the basics, we can chain these together to form higher level combinators. The following parser matches a given string:

```
string :: String -> Parser String
string [] = pure []
string (x:xs) = do
    char x
    string xs
    pure (x:xs)
```

This combinator lets us parse one option out of a list of parsers. We can then specialize the combinator into variants for chars and strings:

```
oneOf :: [Parser a] -> Parser a
oneOf       = foldr (<|>) empty

oneOfChar   ::  String  -> Parser Char
```

```
oneOfString :: [String] -> Parser String
oneOfChar   = oneOf . fmap char
oneOfString = oneOf . fmap string
```

This lets us, for instance, define a parser that catches alphabetic characters:

```
alphabetic :: Parser Char
alphabetic = oneOfChar ['A'..'z']
```

We can make a combinator that matches zero or one of a given parser:

```
zeroOrOne :: Parser a -> Parser (Maybe a)
zeroOrOne x = Just <$> x <|> pure Nothing
```

This one negates a parser - it fails a parse if the given parser succeeds, and vice versa.

```
is_not :: Parser a -> Parser ()
is_not f = Parser $ \s -> case runParse f s of
                            [] -> [((), s)]
                            _  -> []
```

The ApplicativeDo interface lets us define a parser wrapper like this to match parenthesis-wrapped parsers:

```
parens :: Parser a -> Parser a
parens f = do
    char '('
    x <- f
    char ')'
    pure x
```

And finally, don't forget that `Control.Applicative` gives us `some` and `many` for free, which are akin to regex `+` and `*` respectively.

Parsing numbers

Using the above infrastructure and combinators, we can define a sample parser that parses numbers. We start with digit parsers:

```
digit, leading_digit :: Parser Char
digit         = oneOfChar ['0'..'9']
leading_digit = oneOfChar ['1'..'9']
```

Using these, we can already form a natural number parser:

```
natural :: Parser Integer
natural = read <$> do
    lead <- leading_digit
    rest <- many digit
    pure $ lead:rest
```

Then to extend to integers, we can include zero and parse a potential minus sign too:

```
zero :: Parser Integer
zero = char '0' *> pure 0

integer :: Parser Integer
integer = zero <|> do
    minus <- zeroOrOne $ char '-'
    num <- natural
    pure $ case minus of
             Nothing -> num
             _ -> negate num
```

And to parse floating point numbers, we parse an integer and a decimal point followed by one or more digits.

```
floating :: Parser Double
floating = read <$> do
    lead <- integer
    char '.'
    decimals <- some digit
    pure $ show lead <> "." <> decimals
```