0.1 Parser Combinators

Parsing is the process of extracting structured information from a flat, unstructured representation of the data. Parsers are programs that perform this process, using a specified grammar to determine the structure of the data. They are utilised in a variety of applications such as compilers, interpreters, and processing of data storage formats such as JSON and XML.

Traditionally, parsers have either been written by hand or by using parser generator frameworks such as ANTLR [Parr 2013]. Hand-rolling a parser is a tedious process, requiring the programmer to manually implement the parsing algorithm for the grammar. However, this approach is the most powerful and flexible and can provide excellent performance. Alternatively, parser generators lift the burden of implementing the parsing algorithm, instead requiring the programmer to specify the grammar in the format of a domain-specific language (DSL) similar to a high-level grammar. The grammar is then compiled by the parser generator tool to produce a parser in a target language. This approach is less flexible but can be more convenient and less error-prone.

Parser combinators [Hutton 1992], which stem from a functional programming background, are a middle ground between the two approaches. They take the form of an embedded DSL written directly in a general-purpose language, rather than the parser generator approach where the DSL is a separate language. With a parser generator, the provided DSL is often limited in its expressiveness. This is not the case with parser combinators, as the full power of the host language is available to the programmer. This approach also reduces boilerplate code: for example, the programmer does not need to convert between the AST produced by the parser generator and their own AST.

A downside of parser combinators, however, is that they are unstandardised compared to parser generators. Across different implementations, parser combinator APIs can vary significantly, making it difficult to transfer knowledge between different libraries. Experienced users of parser combinators may approach a new library with prior knowledge of general concepts but may have misconceptions about the specifics of the API which can lead to confusion and frustration. This is another motivating reason for the development of `parsley-garnish`, to lower the barrier of entry for new users of the `parsley` library.

0.1.1 Parsley

TODO: proper, worked example showcasing relevant design patterns and stuff which will be picked up by the linter

`Parsley` [Willis and Wu 2018] is a parser combinator library for Scala that provides an API inspired by the `parsec` [Leijen and Meijer 2001] style of parser combinators. This section provides an illustrative example of a simple expression parser to demonstrate what a parser written in `parsley` looks like.

Consider the EBNF grammar for a simple expression language shown in fig. 1a. The parser in fig. 2 will parse an expression into the AST represented by the Scala datatype in fig. 1b.

Notice how the parser closely resembles the high-level EBNF grammar. The main differences of note include the use of:

- `map` to transform the result of a parser to help construct tree nodes consisting of a single value.
- `zipped` to combine the results of two parsers to help construct tree nodes consisting of multiple values.
- `<~` and `~>` operators to guide the direction of parsers.

Except for the possibly cryptic-looking implementation of `num` to parse a series of digits into an integer, the parser is relatively straightforward to understand.

Willis and Wu [2022] describe several design patterns for writing maintainable parsers using parser combinators in Scala. They identified common problems and anti-patterns in parser design, and proposed solutions in the form of design patterns. This provides a guideline for writing idiomatic `parsley` code for practical parser design, which enables opportunities for the development of linting and refactoring rules.

```
ident ::= "x" | "y" | "z"
num ::= digit+
expr ::= factor "+" expr
factor ::= atom "*" factor
atom ::= ident | num | "(" expr ")"
```

(a) The grammar in EBNF.

```scala
sealed trait Expr
case class Ident(name: String) extends Expr
case class Num(x: Int) extends Expr
case class Add(x: Expr, y: Expr) extends Expr
case class Mul(x: Expr, y: Expr) extends Expr
```

(b) The Scala AST to parse into.

Fig. 1: The grammar and AST for our simple expression language.

```scala
val ident = "x" | "y" | "z"
val num: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)

lazy val expr: Parsley[Expr] = (factor, "+" ~> expr).zipped(Add)
lazy val factor: Parsley[Expr] = (atom, "*" ~> factor).zipped(Mul)
lazy val atom: Parsley[Expr] =
  ident.map(Ident) | num.map(Num) | "(" ~> expr <~ ")"
```

Fig. 2: A parser for our simple expression language.

This thesis hopes to explore how these common problems can be formalised into code smells and suspicious code patterns that can be automatically detected using linting rules. Some of the design patterns are also theoretically amenable to automated refactoring, which we hope to explore and implement in `parsley-garnish`.