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 overall boilerplate code: the programmer does not need to convert between the AST produced by the parser generator and their own AST. Instead, the results of parsers, known as their *semantic actions*, are incorporated as part of the parsers themselves.

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 a motivating reason for the development of `parsley-garnish`: to lower the barrier of entry for new users of the `parsley` library.

0.1.1 The **parsley** Library

`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 expands upon the example parser from the introduction to provide a further demonstration of `parsley`. As opposed to directly evaluating the results of the parser, this parser will instead construct an AST representing the structure of arithmetic expression language.

$$
\begin{aligned}
\langle digit\rangle &::= \text{‘}\mathtt{0}\text{’} \ldots \text{‘}\mathtt{9}\text{’} \\
\langle number\rangle &::= \langle digit\rangle+ \\
\langle expr\rangle &::= \langle expr\rangle\ \text{‘}\mathtt{+}\text{’}\ \langle term\rangle \mid \langle expr\rangle\ \text{‘}\mathtt{-}\text{’}\ \langle term\rangle \mid \langle term\rangle \\
\langle term\rangle &::= \langle term\rangle\ \text{‘}\mathtt{*}\text{’}\ \langle atom\rangle \mid \langle term\rangle\ \text{‘}\mathtt{/}\text{’}\ \langle atom\rangle \mid \langle atom\rangle \\
\langle atom\rangle &::= \text{‘}\mathtt{(}\text{’}\ \langle expr\rangle\ \text{‘}\mathtt{)}\text{’} \mid \langle number\rangle
\end{aligned}
$$

(a) The grammar in EBNF.

```scala
enum Expr {
  case Num(n: Int)
  case Add(x: Expr, y: Expr)
  case Sub(x: Expr, y: Expr)
  case Mul(x: Expr, y: Expr)
  case Div(x: Expr, y: Expr)
}
```

(b) The Scala AST to parse into.

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

Recall the EBNF grammar for the arithmetic expression language introduced in the introduction, shown again in fig. 1a. The following parser will parse an expression into the AST represented by the Scala datatype in fig. 1b. To demonstrate an alternate approach, this parser deals with left-recursion using the lower-level `chain.left1` combinator, rather than the higher-level precedence table API:

```scala
import parsley.Parsley
import parsley.character.digit
import parsley.expr.chain
import parsley.syntax.character.charLift

val number: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)

lazy val expr: Parsley[Expr] = chain.left1(term)('+' as Add(_, _) | '-' as Sub(_, _))
lazy val term: Parsley[Expr] = chain.left1(atom)('*' as Mul(_, _) | '/' as Div(_, _))
lazy val atom: Parsley[Expr] = '(' ~> expr <~ ')' | number.map(Num(_))

// Example usage: this parser constructs an AST, rather than evaluating its input
expr.parse("1+2*3/4") // => Add(Num(1), Div(Mul(Num(2), Num(3)), Num(4)))
```

Notice that although the form of the parser retains high-level similarities to the grammar, it is more verbose because it encodes semantic actions (i.e. constructing the resulting AST) directly in the parser.

- The `digit` combinator used in `number` is a parser that recognises a single digit character.
- The `number` parser uses the `foldLeft1` combinator to parse one or more digits, folding them together to form an integer.
- Left-associative parsing is idiomatically handled by the `chain.left1` combinator. This preserves the left-associative structure of the grammar, while avoiding left-recursion. Since the chain combinator makes it explicit that it is parsing a left-associative binary operator, it can be argued that this is a clearer and higher-level way of expressing the intent of the grammar rather than by left-recursion.
- The `'+' as Add(_, _)` parser consumes a `+` character, discarding its result and replacing it with the semantic action of constructing an `Add` node.
- The `map` combinator is used to transform the result of the `number` parser into a `Num` node, lifting its `Int` result to an `Expr`.
- The `~>` and `<~` operators guide the direction of the parser, ensuring that the parentheses are consumed but not included in the final `Expr`.

Design Patterns for Parser Combinators Willis and Wu [2022] describe several design patterns for writing maintainable parsers using parser combinators in Scala. They identify common problems and anti-patterns in parser design, and propose solutions in the form of design patterns. This provides a guideline for writing idiomatic `parsley` code for practical parser design, enabling opportunities for the development of linting and refactoring rules. These design patterns therefore form the basis for many linting rules implemented in `parsley-garnish`.

The example parser already utilises the *Implicit Conversions* design pattern, eliding the need to explicitly use the `char` combinator to parse characters. This is achieved by bringing `parsley.syntax.character.charLift` into scope, allowing Scala to automatically treat character literals as `char` parsers. It also uses the *Chain Combinators* pattern to parse left-associative binary operators, rather than left-factoring the grammar to remove left-recursion – this is seen as an anti-pattern that obfuscates the intent of the grammar.

Parsing Expression Grammars Formally, the semantics of `parsley` parsers are based on parsing expression grammars (PEGs) [Ford 2004]. They differ from traditional context-free grammars (CFGs) [Mascarenhas, Medeiros, and Ierusalimschy 2014] in that the choice operator is ordered: e_1/e_2 will try e_2 only if e_1 fails. In contrast, the choice operator in a CFG is unbiased, which leads to the possibility of ambiguous grammars. The ordered choice operator means that PEGs cannot be ambiguous.