Design Patterns for Parser Combinators in Scala

Jamie Willis
Imperial College London
United Kingdom
j.willis19@imperial.ac.uk

Nicolas Wu
Imperial College London
United Kingdom
n.wu@imperial.ac.uk

Abstract

Parser combinators provide a parsing experience that balances flexibility and abstraction with writing parsers in a style that remains close to the grammar. Parser combinators can benefit from the design patterns and structure of an object-oriented world, however, and this paper showcases the implementation and implications of various design patterns tailored at parsers in an object-oriented and functional world. In particular, features of Scala, such as implicits and path-dependent types, along with general object-oriented design help make it easy to write and maintain such parsers.

Keywords: parser combinators, design patterns

ACM Reference Format:
Jamie Willis and Nicolas Wu. 2022. Design Patterns for Parser Combinators in Scala. In *Scala Symposium (Scala '22), June 6, 2022, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3550198.3550427

1 Introduction

Design patterns are a popular organisational principle for Object-Oriented Programming (OOP). They help target three main classes of issues: the behaviours and interactions of components; the construction of objects; and adapting and connecting interfaces to different systems. The use of the patterns provide programmers with a recipe for how to design their code and work at scale. In languages like Scala, many of the traditional design patterns can be subsumed by language features and design: for example, first-class functions subsume the *Strategy Pattern*. For Domain-Specific Languages (DSLs), language design cannot always be used to circumvent certain library patterns. In particular, writing parsers using *Parser Combinators* has proved to be popular alternative to parser generator libraries, like ANTLR, in the Functional Programming (FP) community [3, 7, 8, 18, 19]. However, the implementation of these libraries directly in Scala means that design patterns start to reappear, since they cannot necessarily be handled by the library's design directly. Although design patterns for parser combinators has been explored in a Haskell setting by Willis and Wu [21], they have not been explored when features of OOP are introduced into the mix. This paper aims to refine a few of the patterns originally presented by Willis and Wu [21], discussing how the unique blending of FP and OOP facilitates clean implementation of parsers in Scala, using parser combinators.

For this paper, we will make use of the library `parsley` [20], but many of our lessons also apply to libraries like `fastparse` or the standard `scala-parser-combinators` [13] library. These libraries are relatively similar to each other, however `parsley` has a design more similar to that of a Haskell parser combinator library, and the addition of an explicit backtracking combinator `attempt` makes the presentation of some of the patterns easier than an implementation using the `~!` (cut) operator found in the latter two libraries. That being said, care will be taken to illustrate the differences in syntax when appropriate. Throughout this paper, **Scala 3** will be the language of choice, as its syntax is more concise than Scala 2's – with improved type inference too.

1.1 The Running Example

This paper will focus discussion around a simple language for expressions, including: arithmetic operators, unary negation, integer literals, parentheses, and variables. Each of the operators is defined to be left-associative, and this is modelled with left-recursion in the grammar:

⟨*digit*⟩ ::= '`0`' .. '`9`'
⟨*number*⟩ ::= ⟨*digit*⟩+
⟨*ident*⟩ ::= ⟨*alpha*⟩ ⟨*alpha-num*⟩*

⟨*expr*⟩ ::= ⟨*expr*⟩ '+' ⟨*term*⟩ | ⟨*expr*⟩ '−' ⟨*term*⟩ | ⟨*term*⟩
⟨*term*⟩ ::= ⟨*term*⟩ '*' ⟨*negate*⟩ | ⟨*negate*⟩
⟨*negate*⟩ ::= '`negate`' ⟨*negate*⟩ | ⟨*atom*⟩
⟨*atom*⟩ ::= '(' ⟨*expr*⟩ ')' | ⟨*number*⟩ | ⟨*ident*⟩

The parser will produce an Abstract Syntax Tree (AST) – though with parser combinators, an evaluation function could be written instead – and for now this is given as the following Abstract Datatype (ADT):

```
enum Expr:
  case Add(x: Expr, y: Expr)
  case Sub(x: Expr, y: Expr)
  case Mul(x: Expr, y: Expr)
```


Scala '22, June 6, 2022, Berlin, Germany

ACM ISBN 978-1-4503-9463-5/22/06…$15.00
https://doi.org/10.1145/3550198.3550427

```
import parsley.character.{digit, letter, alphaNum}
val ident: Parsley[String] =
  (letter <::> many(alphaNum)).map(_.mkString)
val number: Parsley[Int] =
  digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)
```

Figure 1. The `ident` and `number` parsers.

```
lazy val expr: Parsley[Expr] =
      (Expr.Add(_, _)).lift(expr, char('+') ~> term)
  <|> (Expr.Sub(_, _)).lift(expr, char('-') ~> term)
  <|> term
lazy val term: Parsley[Expr] =
      (Expr.Mul(_, _)).lift(expr, char('*') ~> negate)
  <|> negate
lazy val negate: Parsley[Expr] =
      (Expr.Neg(_)).lift(string("negate") ~> negate)
  <|> atom
lazy val atom: Parsley[Expr] =
      (char('(') ~> expr <~ char(')'))
  <|> number.map(Expr.Num(_)) <|> ident.map(Expr.Var(_))
```

Figure 2. The initial expression parser.

```
  case Neg(x: Expr)
  case Num(x: Int)
  case Var(v: String)
```

This AST has a single data type with one case for each operator or value in the language. This does not encode any properties about the grammar, such as its precedence, or indeed the associativity of the operators, which means that bugs that silently change either of these properties in the parser go unchecked. For now, this is fine: the aim here is to show the basic structure of the parser.

To start, the two simple tokens from the grammar – `ident` and `number` – are defined to provide a smaller example of how combinators form full parses: these are shown in Figure 1. These two parsers read the tokens – numbers and identifiers. For `ident`, this makes use of the combinator `<::>`, which prepends the result of the letter on the left to the zero or more alpha-numeric characters from `many(alphaNum)` on the right. The resulting `List[Char]` is converted to a `String` using the `.map` combinator. The `number` parser makes use of `parsley`'s `.foldLeft1` combinator, reading one or more digits and then folding them together to form an integer.

The rest of the parser (Figure 2) uses the `.lift` notation, which allows a function of any arity to be applied to parsers returning the correct types, combining their results to make a new parser[1]. Alternatives in the grammar are expressed by the left-biased `<|>` combinator, and parsers can be sequenced with their results ignored using `~>` and `<~` – the combinators point to the interesting result and discard the other. This set of parser must use `lazy val`s as they are recursive.

[1]This is like the `(p ~ q).map {case (x, y) => f(x, y)}` idiom commonly to other libraries, which would be `f.lift(p, q)` in `parsley`.

Patterns. While parsers have been written in this style for years, knowing the libraries alone is not sufficient to navigate the minefield of practical parser design: this is the role of the design patterns. Our presentation first describes a problem, describes a way to solve the problem which is not idiomatic, then describes one or more patterns that address the problem idiomatically and how to implement them. As an example, the initial presentation of the parser makes use of the `char` and `string` combinators to recognise specific characters and strings in the input. This is fine, but it does make the implementation more verbose, and detaches it from the presentation in the grammar.

Problem 0: Extraneous Lifting of Literals
Even though host language literals are often present in BNF formulations, the incorporation of these primitive tokens in the implementation of parsers is often clunky.

Some of Scala's most powerful features for DSL design are the various forms of *implicits*: implicit *values* can be used to provide proofs that relate different type parameters and control the language; implicit *conversions* can be used to promote regular Scala values so that they can form values in the language; and *extension methods* can be used to provide additional functionality to already existing types, and are capable of introducing laziness into the receiver of a method call. These three features are crucial to our presentation of the parsing design patterns.

Pattern 0: Implicit Conversions
Use implicit conversions to lift values to the parser level, separating the implementation from the parser while retaining the same style as a BNF grammar.

In particular, two implicit conversions can be used here:

```
given Conversion[Char,   Parsley[Char]]   = char(_)
given Conversion[String, Parsley[String]] = string(_)
```

These conversions allow regular Scala literals to serve as parsers for those literals, via `parsley`'s `char` and `string` combinators. In short, when string or character literals appear within any parser fragment in the paper, they are in fact, themselves, parsers. As an example, the regular expression `a(b|c)` can now be expressed as `'a' ~> ('b' <|> 'c')`.

1.2 The Problems with the Parser

Section 2 showcases idiomatic left-recursion handling using chains and strongly typed ASTs

Although the parser in Figure 2 does compile, it does not execute properly: following the grammar, both `expr` and `term` are written using *left-recursion*. This is problematic for recursive descent parsers, as indefinite and unproductive recursion will occur. The creation of `chain` combinators and strongly-typed ASTs help tackle left-recursion and ensure the associativities encoded by the grammar are

faithfully represented by the parser. Our contribution here is to show how the presence of subtyping and implicits can enrich the chain-combinators, as well as show how to formulate precedence combinators using path-dependent types as a substitute for existentials.

Section 3 showcases the use of objects and implicits to encapsulate lexing

Even with left-recursion fixed, no attempt is made in the original parser to handle whitespace, or correctly deal with keyword-identifier interaction. The creation of lexing combinators isolated to their own objects cleanly decouples lexing and parsing, whilst enforcing conventions for implicit whitespace consumption in the parser. Our contribution here is highlighting the benefits of properly encapsulating the implicit conversions within different scopes.

Section 4 showcases the use of the classic OOP *Bridge Pattern* to encapsulate metadata requirements

Some real workloads for parsing, including for compilers, may require additional metadata about the parse to be stored in the AST: for example, the position information. Alternatively, the programmer may wish to enforce invariants like "integers literals may not overflow" within the parser. These tasks can make the parser harder to read, obscuring the details; by making bridge constructors for the parser, these concerns can be cleanly abstracted away. In addition, our contribution shows that using traits and inheritance can exploit the common structure of ADT companion objects to reduce boilerplate.

Related work is discussed in Section 5, and the changes made to the parser are summarised and discussed in Section 6.

2 Handling Left Recursion

Problem 1: Left-Recursive Expressions
Expression grammars with left recursion cannot be encoded by recursive-descent based parser combinators and parsers implementing them loop indefinitely when ran.

Although left-recursion cannot be represented by recursive-descent parsers, grammars can be left-factored [1, 12] to remove the recursion and allow for a productive parser to be written. However, such refactoring of the grammar is less than ideal, as it exposes implementation details, and may obscure the original intention [21].

Anti-Pattern 1: Left-Factoring
Grammar transformations like left-factoring expose implementation details and are laborious.

However, the transformations required to left-factor a grammar actually manifest as the so-called chain combinators [8]; the definition is given as follows:

```
def chainl1[A](p: Parsley[A], op: =>Parsley[(A, A) => A]) =
  lazy val rest: Parsley[A => A] =
        (op, p, rest).zipped {
          (f, y, next) => (x: A) => next(f(x, y))
        }
    <|> pure(identity[A])
  p <**> rest
```

```
def chainr1[A](p: Parsley[A], op: =>Parsley[(A, A) => A]) =
  lazy val go: Parsley[A] =
      p <**> ((op, go).zipped {
          (f, y) => (x: A) => f(x, y)
      } <|> pure(identity[A]))
  go

def prefix[A](op: Parsley[A => A], p: =>Parsley[A]) =
  lazy val go: Parsley[A] = op <*> go <|> p
  go

def postfix[A](p: Parsley[A], op: =>Parsley[A => A]) =
  lazy val rest: Parsley[A => A] =
          (op, rest).zipped(_ andThen _)
      <|> pure(identity[A])
  p <**> rest
```

Figure 3. The `chainr1`, `prefix`, and `postfix` combinators

The idea of `chainl1(p, op)` is that it will parse many `p`s, which return values, separated by `op`s, which return functions, and then will apply the values from `p` to the functions from `op` in sequence with left-associative application. The `.zipped` notation is functionally equivalent to `.lift`, but the function is on the right-hand side to improve Scala's type inference for the lambda. The combinator works by reading a `p` and applying its result to the function returned by `rest` using `<**>`. The `rest` parser reads `op`s and `p`s and composes the functions together to ensure that the result was left-associated, even though the parse was right-associated. This combinator is commonly found in many parser combinator libraries, including `scala-parser-combinators`, `parsley`, and many of the traditional Haskell libraries. For convenience, a similar, right-assocative, combinator called `chainr1`, along with `prefix` and `postfix`, which handle repeated unary operator application to a handle value, are also provided in Figure 3.

Pattern 1a: Chain Combinators
Use `chainl1` or `chainr1` to combine values and operators without explicit left recursion. The combinator should match the associativity required.

The use of these combinators allows the parser to be reformulated without using left recursion at all:

```
lazy val expr: Parsley[Expr] =
  chainl1(term, '+' #> Expr.Add(_, _)
            <|> '-' #> Expr.Sub(_, _))
lazy val term: Parsley[Expr] =
  chainl1(negate, '*' #> Expr.Mul(_, _))
```

The `#>` combinator replaces the result of a parser with a constant. The remainder of the parser is unchanged, though `negate` can use `prefix` to be consistent with the other rules.

By using `chainl1`, the parser is no longer left-recursive and executes fine for input without whitespace.

2.1 Generalising Chains

Although the version of chains presented do fix the left-recursion problem present in the original parser, they are not robust in the sense that, with no non-lexical distinction between the two, there is nothing to stop the parser writer from accidentally using `chainr1` here and no longer representing the grammar [21]. To guard against this mistake, a more strongly-typed AST can be used, which encodes the precedence and associativity of the grammar by introducing a new type for each grammar rule, with a case class per alternative. The best way to encode this type of hierarchy is to use sub-typing, since this allows higher precedence parts of the AST to represent values of a lower level by transitivity. The new AST is structured as follows:

```
sealed trait Expr
sealed trait Term   extends Expr
sealed trait Negate extends Term
sealed trait Atom   extends Negate
object Expr:
  case class Add(x: Expr, y: Term)   extends Expr
  case class Sub(x: Expr, y: Term)   extends Expr
  case class Mul(x: Term, y: Negate) extends Term
  case class Neg(x: Negate)          extends Negate
  case class Num(n: Int)             extends Atom
  case class Var(v: String)          extends Atom
  case class Parens(x: Expr)         extends Atom
```

By using sub-typing, most of the parser can remain the same, so long as the `Parens` case is used to wrap up the recursive point of `Expr` in the corresponding `atom` parser. There is a trade-off to be made here: by increasing the granularity of the types used to construct the tree, it becomes harder to construct incorrect parsers, but does increase the burden on later parts of the processing pipeline. That said, the use of sub-typing means that changes to pattern-matching based tree traversal is minimal and the `Parens` node is the only major change. To make use of this new AST the type of each chain needs to change:

```
def chainl1[A, B >: A](p: Parsley[A],
                       op: =>Parsley[(B, A) => B])
def chainr1[A, B >: A](p: Parsley[A],
                       op: =>Parsley[(A, B) => B])
def prefix[A, B >: A](op: Parsley[B => B], p: =>Parsley[A])
def postfix[A, B >: A](p: Parsley[A], op: =>Parsley[B => B])
```

The type-safety of the combinator is now apparent in the type of the operator parser: the recursive site `B` appears as the first argument for `chainl1` and as the second argument for `chainr1`. By constraining `B` to be the super-type of the next layer down, the terminal value can be safely upcast from type `A` to type `B`, to fit into the corresponding hole in the operator. This is possible since the `Parsley` type is covariant, such that `A <: B` implies that `Parsley[A] <: Parsley[B]`. These combinators still work for the original datatype, since `A <: A`.

Pattern 1b: Sub-typed Hierarchies

Use hierarchial ASTs in conjunction with chains that are generalised for sub-typing to preserve the correct associativity of operators in a grammar.

This is not the most general form of the chain combinators, however, as they are constrained to values that are in a subtype relation with each other. The implicit covariant upcasting from `Parsley[A]` to `Parsley[B]` performed in `chainl1` can be thought of as `.map[B]((x: A) => x)`; more generally however, this can be done for any function `A => B`. This supports the chaining of more ad-hoc structures that are not related by sub-typing. These more general chains are formulated by implicitly providing a function of type `A => B`:

```
def infixl1[A, B](p: Parsley[A], op: =>Parsley[(B, A) => B])
                 (using wrap: A => B) = .. p.map(wrap) ..
def infixr1[A, B](p: Parsley[A], op: =>Parsley[(A, B) => B])
                 (using wrap: A => B) = .. p.map(wrap) ..
def chainl1[A, B >: A]
  (p: Parsley[A], op: =>Parsley[(B, A) => B]) =
    infixl1(p, op)
def chainr1[A, B >: A]
  (p: Parsley[A], op: =>Parsley[(A, B) => B]) =
    infixr1(p, op)
```

The definition of the original chains in term of the new ones works because `A <:< B`, the value-level proof that `A` is a subtype of `B`, can serve as `A => B` and is provided implicitly.[2]

Pattern 1c: Heterogeneous Chains

When an AST cannot rely on sub-typing, use heterogeneous chains such as `infixl1` and `infixr1` to explicitly connect each layer of the structure.

Whilst `infixl1` and `infixr1` are more general, type inference can easily instantiate the types incorrectly. This is discussed, and a fix provided, in Appendix A, but it is handled naturally by patterns introduced in Section 4 so will not be discussed further here.

2.2 Precedence Parsers

Whilst chain combinators are useful in their own right, it can be more natural to express precedence hierarchies like those in our grammar as precedence tables. This is something that is found in *some* parser generator libraries, but notably not in ANTLR – the most common JVM parser generator library. The key advantage of parser combinators over parser generators is that when something is missing in the library it is possible to implement it in terms of other combinators, with the full functionality of Scala to do it.

Precedence parsers have been found in Haskell libraries like `parsec` [11] for a long time relying instead on the classic `chainl1`, and stronger guarantees – like those of `infixl1` – can be gained with *heterogeneous* precedence parsers [21].

[2]For performance, a good library implementation of the `infix` combinators may check to see if `wrap` has type `A <:< B`, skipping the `.map` if it does.

A direct formulation of heterogeneous precedence tables in Scala would be hard to use, since it either requires *existentials* (which introduces unsoundness in Scala 3's type system, or runtime reflection in Scala 2), or many explicit type ascriptions. Instead, we introduce a formulation of these heterogeneous precedence tables leveraging Scala's *path-dependent types* to make a robust and easy to use precedence system.

The sources of the path-dependent types in the system are attached to a `Fixity`, which describes the operator's fixity (and associativity), a datatype given as follows:

```
sealed trait Fixity { type Op[A, B] }
object InfixL extends Fixity{ type Op[-A, B] = (B, A) => B }
object InfixR extends Fixity{ type Op[-A, B] = (A, B) => B }
object InfixN extends Fixity{ type Op[-A, B] = (A, A) => B }
object Prefix extends Fixity{ type Op[A, B] = B => B }
object Postfix extends Fixity{ type Op[A, B] = B => B }
```

The base trait `Fixity` has an associated type `Op[A, B]`, where `A` is the input to the operator, and `B` is the output and recursive input. For each of the five cases, the type `Op` is instantiated with the same types found in the chains. The `InfixN` case denotes non-associative operators, like <, which have no recursive point at all. In the infix cases, the type `-A` can be made contra-variant, but as it is not referenced in `Prefix` and `Postfix`, using contra-variance will make type-inference less robust. The associated types are used to package up a level of the precedence table by combining its operators – instantiated at `A` and `B` – with the conversion function and the fixity that describes the operator's structure:

```
case class Ops[A, B]
  (fixity: Fixity, ops: Parsley[fixity.Op[A, B]]*)
  (using val conv: A => B)
```

This packaging is implemented by the `Ops` case class. Strictly, dependent case classes of this form are not yet allowed in Scala 3; there is hope to lift this restriction in future versions of Scala: see Appendix B for the workaround.

There are three different usage patterns for `Ops`: all the types are equal, types are all related by sub-typing, or types are related by a conversion function. These can be abstracted into objects leveraging each usage pattern:

```
object GOps:
  def apply[A, B](fixity: Fixity)
    (ops: Parsley[fixity.Op[A, B]]*)(using A => B) =
      Ops(fixity, ops: _*)
object SOps:
  def apply[A, B >: A](fixity: Fixity)
    (ops: Parsley[fixity.Op[A, B]]*) = Ops(fixity, ops: _*)
object Ops:
  def apply[A](fixity: Fixity)
    (ops: Parsley[fixity.Op[A, A]]*) = Ops(fixity, ops: _*)
```

The `GOps` object builds `Ops` objects using a conversion function, mapping to the most general formulation of the chains (`infixl1` and `infixr1`) from Section 2.1. The `SOps` object instead captures the case where the input type `A` is a subtype of the output type `B`: this corresponds to the original chain combinators with sub-typing. The `Ops` object handles the case where all the types are the same: the original use-case.

```
sealed trait Prec[+A]:
  def :+[A' >: A, B](ops: Ops[A', B]): Prec[B] =
    Level(this, ops)
  def +:[A' >: A, B](ops: Ops[A', B]): Prec[B] =
    Level(this, ops)
case class Level[A, B](lvls: Prec[A], ops: Ops[A, B])
  extends Prec[B]
case class Atoms[+A](atoms: Parsley[A]*) extends Prec[A]
```

Whilst `Ops` represents a single layer of the precedence table, many levels are tied together using the `Prec` type. This is built out of the base case `Atoms` and a recursive case `Level`, which takes an `Ops` along with a sub-table that can generate its input type. The `Prec` type is constructed using the traditional Scala operators `:+` and `+:`, which allows the table to be ordered from either tightest-to-weakest or weakest-to-tightest. To allow a `Prec` to be covariant, the type `A'` is used to avoid a variance mismatch. The `precedence` function is required to fold a `Prec` value into the end product:

```
def precedence[A](lvls: Prec[A]): Parsley[A] =
  def convertOps[A, B]
    (atom: Parsley[A], lvl: Ops[A, B]): Parsley[B] =
    given (A => B) = lvl.conv
    lvl match
    case Ops(InfixL, ops)  => infixl1(atom, choice(ops: _*))
    case Ops(InfixR, ops)  => infixr1(atom, choice(ops: _*))
    case Ops(Prefix, ops)  => prefix(choice(ops: _*), atom)
    case Ops(Postfix, ops) => postfix(atom, choice(ops: _*))
    case Ops(InfixN, ops)  =>
      val op = choice(ops: _*)
      atom <**> ((op, atom).zipped((f, y) => f(_, y))
            <|> pure(lvl.conv))
  lvls match
  case Atoms(atoms: _*) => choice(atoms: _*)
  case Level(lvls, ops) => convertOps(precedence(lvls), ops)
```

The `precedence` function recursively pattern matches on `lvls` using `convertOps` to convert each `Ops` value into a parser along with the fully folded levels below. The `choice` combinator used here tries each parser in a sequence until one of them succeeds: it allows for variadic arguments to each layer, for convenience. The case for `InfixN` is similar to the implementation of `infixr1` with no recursion: the parser is left-factored to eliminate any potential backtracking.

Pattern 1d: Precedence Combinators

For a more concise representation of a collection of operators with precedences, use the `precedence` combinator to encode a precedence and associativity table.

Now the example parser can be reformulated with `precedence`:

```
lazy val expr: Parsley[Expr] = precedence(
  SOps(InfixL)('+' #> Expr.Add(_, _),
               '-' #> Expr.Sub(_, _)) +:
  SOps(InfixL)('*' #> Expr.Mul(_, _)) +:
  SOps(Prefix)("negate" #> Expr.Neg(_)) +:
  Atoms('(' ~> expr.map(Expr.Parens(_)) <~ ')',
        number.map(Expr.Num(_)),
        ident.map(Expr.Var(_))))
```

Here, `SOps` has been used for all operators to indicate their sub-typing relation, however in this case `GOps` also works without any type error.

Discussion. The use of precedence tables here has introduced flexibility in the way that the parser can be ordered, and avoids some of the type-inference fragility of using `infixl1` and `infixr1` directly, whilst being just as powerful. That is not to say that the chains are useless: for cases when there is only one level of precedence, using the full `precedence` infrastructure is overkill!

3 Token Parsing

The parser defined up to this point now executes correctly, but only for a subset of the valid inputs recognised by the grammar. The problem is that the parser, so far, has not made any attempt to deal with the consumption of whitespace, nor has it dealt with the correct tokenisation of keywords. Here are some examples that illustrate the issues (here `eof` can be used to force the parser to consume all available input):

```
scala> expr.parse("x + 7")
val res0: Result[String, Expr] = Success(Var(x))

scala> (expr <~ eof).parse("x + 7")
val res1: Result[String, Expr] = Failure((line 1, column 2):
  unexpected space
 expected "*", "+", "-", alpha-numeric character, or end of input
  >x + 7
    ^)

scala> (expr <~ eof).parse("negatex")
val res2: Result[String, Expr] = Success(Neg(Var(x)))
```

Problem 2: In-Place Lexing
Handling tokenisation and whitespace inside the main parser is obstructive and brittle.

Here, it is evident that whitespace is not being consumed: in the first example, the parse stops at the first space, and by forcing a parse to end-of-file, a parse error pointing to the first space is generated. The third example shows that parsing keywords is improper: the result of this parse should have been `Success(Var("negatex"))`.

With traditional parsing techniques, the solution is often to split parsing into two phases: lexing and parsing. This allows the grammar to make reference to tokens as opposed to strings, and the job of whitespace parsing and token discrimination can be left to the lexer. However, this is not often done with parser combinators, which offer more flexibility: the lexing combinators can be naturally expressed themselves using combinators. By building the right abstractions, the lexing problem can still be factored away from the parser without relying on a separate phase or tool.

Anti-Pattern 2: Separate Lexing Phase
Using a separate lexer with parser combinators loses context-aware lexing, making token selection harder.

An advantage of a single-pass parser is that it allows the lexer to adopt elements of context sensitivity, where tokens are statically ruled out within certain contexts in the grammar: this reduces token overloading, and it is easier to resolve ambiguities this way. An example of this is the distinction between unary negation, binary subtraction, and negative integer literals: it is easy to disambiguate them when the parser only demands them in specific parts of the grammar.

3.1 Whitespace Parsing

Whilst consuming whitespace is relatively straightforward, it is important to maintain consistency about when whitespace should be consumed. Poorly handling whitespace can affect the accuracy of error messages, and can impact the performance (or even semantics!) of a parser – especially for libaries where cutting (via `~!`) is present or backtracking is opt-in (as per `parsley`). In fact, it is important to consume whitespace *after* tokens, and not before (except for once at the very beginning of the parse). To see what can go wrong when leading whitespace is consumed, here is an example:

```
(expr <~ eof).parse("x + 7")
val res0: Result[String, Expr] = Failure((line 1, column 3):
  unexpected "+"
  expected "*"
  >x + 7
     ^)
```

Here, whitespace is consumed before every token, and the parser produces an incorrect result: after reading `"x"`, it tries to parse `"*"`, first reading whitespace; this obviously fails, but input has already been consumed by reading whitespace; the parser is unable to backtrack as a result (cut-semantics are assumed by default in `parsley`). The parser could be fixed by using the `attempt` combinator, which locally permits backtracking, but now the complexity of the parser is no longer linear in the input length! With leading whitespace best avoided, the only other ways of parsing whitespace are reading trailing whitespace and reading *both* leading and trailing whitespace – this is wasted computation.

With the convention settled, how should whitespace be consumed? In the combinator style, it is best to abstract into a set of combinators. For our example language, all whitespace can be treated equally and there are no comments:

```
import parsley.character.{space, endOfLine}
val ws = skipMany(space <|> endOfLine).hide
```

The parser `ws` reads zero or more spaces, tabs, or newlines (including both `LF` and `CRLF`). For error message hygiene, these are hidden from appearing as suggestions for expected tokens in the error messages using `.hide`.

Pattern 2a: Whitespace Combinators

Build a `lexeme` combinator to enforce that only trailing whitespace is consumed. Build a `fully` combinator to consume the initial whitespace and the end-of-file.

To ensure that the convention established above is upheld, two more combinators are constructed to make use of `ws`.

```
def lexeme[A](p: Parsley[A]): Parsley[A] = p <~ ws
def fully[A](p: Parsley[A]): Parsley[A] = ws ~> p <~ eof
```

The `lexeme` combinator first parses a given parser, and then consumes the trailing whitespace. The `fully` combinator is designed to be used at the top-most level, consuming the initial whitespace and ensuring that the parser consumes all the available input. With these two combinators available, using the `ws` parser should be discouraged: this can be done by encapsulating all three combinators in a `lexer` object, with `ws` kept private.

Pattern 2bi: Tokenizing Combinators

Build a `token` combinator to ensure the atomicity of tokens in the language, building on `lexeme`.

In addition to these two combinators, tokens should be parsed atomically: in classical parsing techniques tokens are indivisible. This means that failing to parse a token mid-way through should never consume input. This can be accomplished via the `attempt` combinator.

Pattern 2c: Implicit Lexers

Specialise the *Implicit Conversions* pattern to allow string (and character) literals to parse as tokens.

The original parser allows string literals to serve as parsers by using a `Conversion[String, Parsley[String]]`. This kept the structure of the parser as close as possible to the BNF. In the same vein, the `lexer` object can expose a new version of the conversion, allowing whitespace and tokenisation to be incorporated into the original parser without changing it. The other combinators and machinery can be kept private, and both `number` and `ident` should be moved to `lexer` also:

```
object lexer:
  private val ws = skipMany(space <|> endOfLine).hide
  private def lexeme[A](p: Parsley[A]) = p <~ ws
  def fully[A](p: Parsley[A]) = ws ~> p <~ eof
  private def token[A](p: Parsley[A]) = lexeme(attempt(p))

  private given Conversion[Char, Parsley[Char]] =
    c => char(c)
  private given Conversion[String, Parsley[String]] =
    s => string(s)

  val ident =
    token(letter <::> many(alphaNum)).map(_.mkString)
  val number = token(digit.foldLeft1(0)(_ * 10 + _.asDigit))

  object implicits:
    given Conversion[Char, Parsley[Char]] =
      c => lexeme(char(c))
    given Conversion[String, Parsley[String]] =
      s => token(string(s))

import lexer.{number, ident}
import lexer.implicts.given
val parser: Parsley[Expr] = fully(expr)
...
```

By importing the components of `lexer` into scope and wrapping the whole parser using `fully`, the parser will successfully handle any and all whitespace. Notice that the lexer itself has access to the original implicit conversions that do not deal with whitespace: this allows for literals to still be used in the lexer whilst ensuring explicit whitespace handling is performed (this is not required for this example). The original conversions are kept private so that the parser must pull the new whitespace-aware conversions from a dedicated scope; this extra scope ensures that the lexer does not accidentally make use of whitespace-handling combinators for its own definitions. Of course, since all the logic involving tokens has been factored into the exposed `Conversion`s, no changes are necessary to the rest of the parser.

3.2 Correct Keyword Handling

The final problematic input given at the start of Section 3: `"negatex"`, does still not parse correctly, despite whitespace handling having been fixed. As a reminder, the parser treats it as an application of `negate` to `x`, as opposed to an identifier in its own right. Traditionally, tokens are formed in lexers using *maximal munch*, for longest-match behaviour: this approach would have found the longer continuous token `Ident("negatex")`. Parser combinator libraries, however are not based on this approach: they instead greedily pick the first successful branch, which in this case is the operator.

Pattern 2bii: Keyword Combinator

In addition to `token`, build a `keyword` combinator that will implement longest-match lexing behaviour.

To correctly parse these cases – where a keyword is directly connected to more letters – a separate combinator `keyword` should be developed. This combinator ensures that a keyword can only be parsed if it is not directly proceeded by another valid identifier letter by using negative lookahead:

```
object lexer:
  ...
  private def keyword(key: String): Parsley[String] =
    token(string(key) <~ notFollowedBy(alphaNum))
```

It is important here to ensure that whitespace and atomisation is performed *after* the negative lookahead has taken place. This combinator has been left private, and its use can be incorporated into the implicit conversion from string literals to parsers. To do this, the string to be converted must be checked to see if it is part of the valid keywords, deferring to `keyword` if this is the case:

```
object lexer:
  ...
  private val keys = Set("negate")
  object implicits:
    given Conversion[String, Parsley[String]] = s =>
      if keys.contains(s) then keyword(s)
      else                     token(string(s))
```

With this change made, the parser works as intended on all valid and invalid inputs. A similar system can be implemented to handle operators with ambiguous prefixes, such as not parsing + given the input "++", for instance; this is not needed for this grammar, however, so is omitted for brevity.

Discussion. Thanks to the early support of the *Implicit Conversions* pattern, the work required to handle whitespace, tokenization, and keyword handling has no effect on the structure of the parser from the end of Section 2. However, by shifting the `Conversion`s used by the parser into the lexer object a clean encapsulation of the behaviours is achieved. By further nesting the implicits inside another object, the original plain string and character literal parsers could be used inside the lexer, if desired. This highlights the flexibility that OOP offers in this space, with the principles of encapsulation shining through.

4 Handling Metadata

Often, parsers need to attach various pieces of metadata to generated syntax trees. Commonly, this would be position information, so that semantic errors about the structure can refer back to locations in the original input. When this is required, it has a tendency to be invasive to the parser. The `pos` combinator can be used to acquire line and column information during a parse.

Problem 3: Parsing Metadata
Parsers often need to collect parsing metadata and incorporate such information into the results of the parse.

To make use of position information, assignment and a yield statement will be added to the grammar, and position information will be attached to variables, the assignments, and the yields. The new grammar is as follows:

⟨*stmts*⟩ ::= ⟨*stmt*⟩ ';' ⟨*stmts*⟩ | ⟨*stmt*⟩
⟨*stmt*⟩ ::= ⟨*asgn*⟩ | ⟨*yield*⟩
⟨*asgn*⟩ ::= ⟨*ident*⟩ ':=' ⟨*expr*⟩
⟨*yield*⟩ ::= 'yield' ⟨*expr*⟩

The new AST nodes are similar to those used for `Expr`, however they also have an additional position attached:

```
type Pos = (Int, Int)
sealed trait Stmts
sealed trait Stmt extends Stmts
object Stmt:
  case class Seq(stmt: Stmt, stmts: Stmts) extends Stmts
  case class Asgn(v: String, x: Expr)
                 (val pos: Pos)            extends Stmt
  case class Yield(x: Expr)(val pos: Pos)  extends Stmt
object Expr:
  ...
  case class Var(v: String)(val pos: Pos)  extends Atom
```

The position is kept out of the first set of brackets since it is more auxilliary information than it is a key part of a pattern match. The parser defined naïvely as follows:

```
lazy val stmts =
  infixr1(stmt, ';' #> Stmt.Seq(_, _))
lazy val stmt = asgn <|> `yield`
lazy val asgn =
  pos <**> (Stmt.Asgn(_, _)).lift(ident <~ ":=", expr)
lazy val `yield` =
  pos <**> ("yield" ~> expr.map(Stmt.Yield(_)))
...
      pos <**> ident.map(Expr.Var(_)))
```

This parser makes use of `infixr1` to parse the right-associative semi-colons and the rest of the parser is routine, and the new conversion introduced in Section 3.2 ensures `yield` is parsed as a keyword. This works fine, however, the addition of `pos: Parsley[(Int, Int)]` is distracting from the overall shape of the parser: the grammar itself does not mention positions, why should the parser! The writer must also put the `pos` before the first token of the structure is parsed, otherwise it will point in the wrong place. In a similar way, the `f(_, _).lift` is quite distracting to the reader.

Anti-Pattern 3: Inline Metadata Extraction
Handling metadata collection in the main parser obscures the grammar, and can make ergonomics suffer.

These can both be resolved concisely with another pattern: the *Bridge Pattern* [5]. The traditional *Bridge Pattern* is concerned with decoupling an abstraction from any concrete implementations. This means that the use of the abstraction can vary independently from the specifics of the implementation. In our interpretation, AST nodes can be constructed by a function whose job is to handle the complex metadata and class-selection behaviours—the implementation of which varies from node to node; and the combination of results of parsers can be kept abstract and separate. By being able to also change which specific kind of AST node may be generated, some bridges are similar to instances of the *Factory Method Pattern* [5].

Pattern 3a: Parser Bridges
Separate the construction of an AST node and metadata collection by using bridge constructors in the parser.

In Scala, this is best achieved with the `.apply` method, which acts as function call overloading, treating an object as a function. The idea is to create a new overloaded constructor that operates on parsers instead of values; this boils down to lifting the original constructor using `.lift`[3] and applying the position when required.

[3] In practice, `.zipped` is used as it aids types inference.

In addition, there are some kinds of bridges that share a common structure across many different AST nodes. In these cases, the process of creating the bridges can be abstracted: in Scala, all `case class`es have a convenience `.apply` method with the same shape as the constructor generated in the companion object. It is possible to hook into this method and create uniform `ParserBridge` traits that supply the desired bridge constructor via the *Template Method Pattern* [5]:

```
trait ParserBridge1[-A, +B]:
  def apply(x: A): B // the abstract hook method
  // template method for the bridge constructor
  def apply(px: Parsley[A]): Parsley[B] = px.map(apply)

trait ParserBridgePos1[-A, +B]:
  def apply(x: A)(pos: Pos): B
  def apply(px: Parsley[A]): Parsley[B] =
    pos <**> px.map(apply)

trait ParserBridgePos2[-A, -B, +C]:
  def apply(x: A, y: B)(pos: Pos): C
  def apply(px: Parsley[A], py: Parsley[B]): Parsley[C] =
    pos <**> (px, py).zipped(apply)
```

Each of these traits provides a template for the general case of lifting a constructor and handling auxilliary position information when required. The bridges become available for the AST by mixing in the appropriate `ParserBridge` *generic bridge trait* into the companion object:

```
object Stmt:
  ...
  object Asgn extends ParserBridgePos2[String, Expr, Stmt]
  object Yield extends ParserBridgePos1[Expr, Stmt]

object Expr:
  ...
  object Num extends ParserBridge1[Int, Num]
  object Var extends ParserBridgePos1[String, Var]
  object Parens extends ParserBridge1[Expr, Parens]
```

By mixing in the generic bridge traits to each of the AST nodes, the position tracking is be fully encapsulated: it is no longer necessary for the rest of the parser to have to worry about whether or not positions must be tracked. In fact, no changes need to be made to the parser to either add or remove position tracking: it suffices to change the generic bridge trait used. The parser can be changed as follows:

```
lazy val stmts = infixr1(stmt, ';' #> Stmt.Seq(_, _))
lazy val stmt = asgn <|> `yield`
lazy val asgn = Stmt.Asgn(ident <~ ":=", expr)
lazy val `yield` = Stmt.Yield("yield" ~> expr)
lazy val expr: Parsley[Expr] = precedence(
  SOps(InfixL)('+' #> Expr.Add(_, _),
               '-' #> Expr.Sub(_, _)) +:
  SOps(InfixL)('*' #> Expr.Mul(_, _)) +:
  SOps(Prefix)("negate" #> Expr.Neg(_)) +:
  Atoms(Expr.Parens('(' ~> expr <~ ')'),
        Expr.Num(number),
        Expr.Var(ident)))
```

This is more concise than the previous versions, presenting a natural syntax for constructing AST nodes from parsers. Notice, however, that the parts of the parser that are interacting with `infixr1` or the `precedence` machinery still refer directly to constructors. The problem is that the arguments to these constructors are not fully saturated: they are provided by the `precedence` combinator. This does not mean that the bridges cannot be applied, but it does mean that the pattern must be extended.

Pattern 3b: Singleton Bridges
When arguments to bridge constructors are not immediately available, turn the companion object itself into a parser, encapsulating any administration within.

Instead, the companion objects of the AST nodes must themselves serve as parsers, or they support a combinator that allows them to be combined with other parsers. This is an example of the *Singleton Pattern* [5]. This can be written with the help of a more general `ParserBridge` trait:

```
trait ParserBridge[+A]:
  val parser: Parsley[A]
  final def <#(p: Parsley[_]): Parsley[A] = parser <~ p
```

An implementation of this trait provides a parser (the hook) that yields a value of the desired type. It uses this parser to complete a template for the `<#` combinator, which mirrors the `#>` for substituting the results of a parser with a value. In this variant of the combinator, however, it is a full parser, allowing it to incorporate metadata, and so on, as required. The `ParserBridge` trait can serve as the base trait for each of the more specialised generic bridge traits since each can implement its `parser` in terms of its own abstract `.apply` hook method:

```
trait ParserBridge1[-A, +B] extends ParserBridge[A => B]:
  ..
  override val parser = pure(apply)

trait ParserBridgePos1[-A, +B]
  extends ParserBridge[A => B]:
  ..
  override val parser = pos.map(p => apply(_)(p))

trait ParserBridgePos2[-A, -B, +C]
  extends ParserBridge[(A, B) => C]:
  ..
  override val parser = pos.map(p => apply(_, _)(p))
```

Now, each companion object can be used as a lifted constructor or as a singleton. The remaining companion objects for `Add`, `Sub`, `Mul`, and `Neg` can now all have the appropriate generic bridge trait mixed in, and the parser can be simplified using the companions directly. The parser's final form is as follows:

```
lazy val stmts = infixr1(stmt, Stmt.Seq <# ';')
lazy val stmt = asgn <|> `yield`
lazy val asgn = Stmt.Asgn(ident <~ ":=", expr)
lazy val `yield` = Stmt.Yield("yield" ~> expr)
lazy val expr: Parsley[Expr] = precedence(
  SOps(InfixL)(Expr.Add <# '+', Expr.Sub <# '-') +:
  SOps(InfixL)(Expr.Mul <# '*') +:
```

```
SOps(Prefix)(Expr.Neg <# "negate") +:
Atoms(Expr.Parens('(' ~> expr <~ ')'),
      Expr.Num(number),
      Expr.Var(ident)))
```

Discussion. This is the complete, final, version of the parser for the whole grammar. As mentioned before, any additional work that might want to be incorporated into the parser can be done by tweaking the bridges and singletons. Again, the key advantage of these patterns is to help improve the separation between the various components of the parser: in this case, administrative work involving the construction of AST nodes has been encapsulated by the bridge constructors, and the main body of parser corresponding to the grammar remains in its purest form. An effect of the bridge constructors in particular is that the types can be readily fixed to forms that are automatically more amenable for `infixl1` and `infixr1`: this fixes the type-inference issues presented way back in Section 2.1.

One issue, however, is that the generic bridge traits cannot capture all of possible nuances of an AST node's construction: from time to time, bridge constructors may have to be written manually to accomodate, and abstract, more complex requirements. For example, it may be desirable to enforce that numeric literals are not too big to fit within the type they will inhabit within the AST; or the AST itself may have additional type-indices to support a tree decoration frame like "Trees that Grow" [14]. In either case the generic bridge traits described in this section would not apply, but the general technique of using parser bridge constructors for separation of concerns does. Library authors may wish to include the position-unaware generic bridge traits as part of their library to save on user boilerplate.

5 Related Work

Design patterns have been used to organise large-scale programs in object-oriented language for decades [5]. They are employed regularly in the design of libraries, and often the usefulness of a pattern is reflected in first-class support for it within a language's design. In the parser design space, the popular parser generator library ANTLR [16] is no exception: it provides support for the generation of the *Visitor Pattern* and the *Observer Pattern* to aid in the processing of a parse tree into an AST. With parser combinators, the creation of the AST is more tightly coupled with the parser itself: the *Parser Builders* pattern in our paper helps separate the administration – which might be performed using a *Visitor* in ANTLR – from the parser itself. Our *Parser Bridge* pattern is itself based in the traditional *Bridge Pattern* and it is abstracted using the *Template Method Pattern*; the *Singleton Bridge* pattern incorporates the *Singleton Pattern* too to provide a uniform interface to the AST. At the very essence of parser combinator design, the *Strategy Pattern* is crucial, taking the form of the first-class functions that are provided to the `.lift` combinator! In `parsley`'s own design, the *Observer Pattern* is used to help build customised parse errors. Design patterns find homes in many other libraries too, including parsing ones [15, 17].

Most of the patterns we discuss in this paper, and indeed the problems, have also been explored in Haskell – a purely functional language – by Willis and Wu [21]. In contrast, we offer additional insights into how these patterns fit into an object-oriented language. In their work, they discuss the *Homogeneous Chains* pattern, which corresponds to the classic version of the chain combinators we first discussed. However, due to sub-type polymorphism being unavailable, they put more focus on *Heterogeneous Chains* as the solution to the grammar associativity problem. In our formulation, the natural structure of sub-typing allows us to lean on the *Subtyped Hierarchies* pattern to automatically provide a seamless typing guarantee whilst still using the homogeneous chains – without explicit wrapping of datatypes. We show how to adapt their *Precedence Combinators* pattern into a language without existential types using Scala's path-dependent typing. We generalise their *Overloaded Strings* pattern to make use of Scala's popular implicit mechanism in both *Implicit Conversions* and *Implicit Lexers* patterns, providing additional, more fine-grained, control over where the conversions may apply; and allow the mechanism to also work on types other than `String`, including `Char` – though it could be extended to many different literal values! Willis and Wu also describe *Lifted Constructors* and *Deferred Constructors*, which are analogous to our *Parser Bridges* and *Singleton Bridges*: we have highlighted the similarity of these patterns to the traditional OOP frameworks, and leveraged the language-level support for these patterns in Scala to provide a cleaner, more idiomatic, interface. An advantage of our presentation of these patterns compared with theirs is that the boiler-plate can be abstracted away by the generic builder traits.

Of course, just as traditional design patterns may be made obsolete by language design, some of the patterns presented here are made obsolete by different library designs. In particular, the problem of left recursion is often tackled by some libraries using either automatic grammar refactoring [2] via a left-corner transform [12]; or memoisation to ensure recursive calls can be resolved [4, 9, 10]. The *Precedence Combinators* pattern is also sometimes encoded first-class in some parser generator libraries such as Happy [6].

6 Conclusion

The full grammar supporting both the original arithmetic operations and the statements added in Section 4 can be found in Figure 4. In addition, the final parser for this complete grammar, along with the `lexer` object can be seen in Figure 5. Originally, the parser suffered from problems with left recursion, causing non-termination, and did not parse whitespace properly. Left recursion can be fixed by using grammar

⟨*digit*⟩ ::= '0' .. '9'
⟨*number*⟩ ::= ⟨*digit*⟩+
⟨*ident*⟩ ::= ⟨*alpha*⟩ ⟨*alpha-num*⟩*
⟨*stmts*⟩ ::= ⟨*stmt*⟩ ';' ⟨*stmts*⟩ | ⟨*stmt*⟩
⟨*stmt*⟩ ::= ⟨*asgn*⟩ | ⟨*yield*⟩
⟨*asgn*⟩ ::= ⟨*ident*⟩ ':=' ⟨*expr*⟩
⟨*yield*⟩ ::= 'yield' ⟨*expr*⟩
⟨*expr*⟩ ::= ⟨*expr*⟩ '+' ⟨*term*⟩ | ⟨*expr*⟩ '-' ⟨*term*⟩ | ⟨*term*⟩
⟨*term*⟩ ::= ⟨*term*⟩ '*' ⟨*negate*⟩ | ⟨*negate*⟩
⟨*negate*⟩ ::= 'negate' ⟨*negate*⟩ | ⟨*atom*⟩
⟨*atom*⟩ ::= '(' ⟨*expr*⟩ ')' | ⟨*number*⟩ | ⟨*ident*⟩

Figure 4. The full grammar

```
object lexer:
  private val ws = skipMany(space <|> endOfLine).hide
  private def lexeme[A](p: Parsley[A]) = p <~ ws
  def fully[A](p: Parsley[A]) = ws ~> p <~ eof
  private def token[A](p: Parsley[A]) = lexeme(attempt(p))
  private def keyword(key: String) =
    token(string(key) <~ notFollowedBy(alphaNum))
  private val keys = Set("negate", "yield")

  val ident =
    token(letter <::> many(alphaNum)).map(_.mkString)
  val number = token(digit.foldLeft1(0)(_ * 10 + _.asDigit))

  object implicits:
    given Conversion[String, Parsley[String]] = s =>
      if keys(s) then keyword(s)
      else            token(string(s))
    given Conversion[Char, Parsley[Char]] =
      c => lexeme(char(c))

import lexer.{fully, ident, number}
import lexer.implicits.given
lazy val parser = fully(stmts)
lazy val stmts = infixr1(stmt, Stmt.Seq <# ';')
lazy val stmt = asgn <|> `yield`
lazy val asgn = Stmt.Asgn(ident <~ ":=", expr)
lazy val `yield` = Stmt.Yield("yield" ~> expr)
lazy val expr: Parsley[Expr] = precedence(
  SOps(InfixL)(Expr.Add <# '+', Expr.Sub <# '-') +:
  SOps(InfixL)(Expr.Mul <# '*') +:
  SOps(Prefix)(Expr.Neg <# "negate") +:
  Atoms(Expr.Parens('(' ~> expr <~ ')'),
        Expr.Num(number),
        Expr.Var(ident)))
```

Figure 5. The final parser

refactoring, however the *Chain Combinators* pattern allows the left-factoring algorithm to be concisely represented as an operator in the combinator language. By strengthening the AST according to *Sub-typed Hierarchies*, the chain combinators enforced the associativies encoded by the grammar in conjunction with stronger types whilst still providing a natural and ergonomic interface. By using *Precedence Combinators*, the parser was more concisely described as a table of precedence and associativity, whilst still maintaining all the benefits of the previous iterations – leveraging path-dependent types to do so.

To tackle the issues of lexing and whitespace, a `lexer` object was devised to help encapsulate the lexing logic, ensuring it is used consistently in the parser only by the provided combinators. Whitespace was handled consistently using *Whitespace Combinators*, and the atomicity and maximal-munch properties of tokens are guaranteed by *Tokenizing Combinators* and the *Keyword Combinator*. The changes introduced to the parser, however, were non-existent: by using the *Implicit Conversions* pattern early on, the method for parsing string and character literals was decoupled from their use in the parser, allowing *Implicit Lexers* to automatically handle the tokenization logic for these literals.

When the grammar was extended to incorporate simple statements, the fact that often parsers need to collect metadata – such as positions – during parsing, was introduced. This added an extra challenge, since the addition of the `pos` combinator pulled the parser further from the BNF representation. To combat this, the *Parser Bridges* and *Singleton Bridges* abstracted the construction of the AST node – and associated metadata – from the parser, leveraging classic OOP design patterns and techniques. This yields a final parser which is clean, well abstracted, and easily extendable and maintainable. In all, we feel this highlights that the interplay between the traditionally functional code and object-oriented principles can produce great results for parser implementation and library design.

Acknowledgements

We would like to thank the reviewers for their helpful and constructive feedback. We would also like to thank Rob Chatley for our discussion about the most appropriate name for the *Parser Bridge* pattern. This work was supported by EPSRC grant number EP/S028129/1 on 'Scoped Contextual Operations and Effects'.

A For Library Authors: Improving inference for `infixl1` and `infixr1`

Whilst `infixl1` and `infixr1` are more general, their use (as opposed to the original sub-typed `chainl1`) does cause Scala's type inference to trip up:

```
lazy val term: Parsley[Term] =
  infixl1(negate, '*' #> Expr.Mul(_, _))
// no implicit argument of type Negate => Expr.Mul was found
// for parameter wrap of method infixl1
```

Annoyingly, without explicit sub-typing to guide it, Scala has chosen to instantiate the types with `Negate` and `Expr.Mul` instead of the desired `Negate` and `Term`. This, of course, can be fixed with a type ascription, but it is frustrating that it is not a drop-in replacement for `chainl1`! There is a way of generalising the combinator a little more to help the compiler

instantiate the types correctly: introduce a third type `C >: B`, which serves as the new recursive point in the operator. This encourages the compiler to find `Term >: Expr.Mul`.

```
def infixl1[A, B, C >: B]
  (p: Parsley[A], op: =>Parsley[(C, A) => B])(using A => C)
def infixr1[A, B, C >: B]
  (p: Parsley[A], op: =>Parsley[(A, C) => B])(using A => C)
```

The downside of this, however, is that if type inference were to fail in this formulation, then even more types would have to be ascribed on the combinator, for very little gain.

B For Library Authors: Implementing precedence without Dependent Case Classes

The discussion in Section 2.2 relies on dependent case classes, which are disallowed in early versions of Scala 3 – see Issue #8073. Because of this, implementing `Ops` is a little less clean:

```
class Ops[A, B](val fixity: Fixity)
               (val ops: Parsley[fixity.Op[A, B]]*)
               (using val conv: A => B)
```

In this formulation, the dependent argument must be placed in a separate set of brackets, ruling out any meaningful form of pattern matching on `Ops`. While this seems benign here, the effect is obvious in the `precedence` function:

```
def precedence[A](lvls: Prec[A]): Parsley[A] =
  def convertOps[A, B]
    (atom: Parsley[A], lvl: Ops[A, B]): Parsley[B] =
    given (A => B) = lvl.conv
    lvl.fixity match
    case InfixL  => infixl1(atom, choice(
      lvl.ops
         .asInstanceOf[Seq[Parsley[InfixL.Op[A, B]]]]: _*))
    case InfixR  => infixr1(atom, choice(
      lvl.ops
         .asInstanceOf[Seq[Parsley[InfixR.Op[A, B]]]]: _*))
    case Prefix  => prefix(choice(
      lvl.ops
         .asInstanceOf[Seq[Parsley[Prefix.Op[A, B]]]]: _*),
      atom)
    case Postfix => postfix(atom, choice(
      lvl.ops
         .asInstanceOf[Seq[Parsley[Postfix.Op[A, B]]]]: _*))
    case InfixN  =>
      val op = choice(
        lvl.ops
           .asInstanceOf[Seq[Parsley[InfixN.Op[A, B]]]]: _*)
      atom <**> ((op, atom).zipped((f, y) => f(_, y))
            <|> pure(lvl.conv))
  lvls match
  case Atoms(atoms: _*) => choice(atoms: _*)
  case Level(lvls, ops) => convertOps(precedence(lvls), ops)
```

Here, the relation between the `fixity` and the `ops` is not uncovered by the pattern match (even if `Ops` were a case class in just fixity), so an `asInstanceOf` must be used to convince the compiler of the types correctness.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. https://doi.org/10.5555/1177220

[2] Dominique Devriese and Frank Piessens. 2011. Explicitly Recursive Grammar Combinators. In *Practical Aspects of Declarative Languages*, Ricardo Rocha and John Launchbury (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–98. https://doi.org/10.1007/978-3-642-18378-2_9

[3] R. Frost and J. Launchbury. 1989. Constructing Natural Language Interpreters in a Lazy Functional Language. *Comput. J.* 32, 2 (apr 1989), 108–121. https://doi.org/10.1093/comjnl/32.2.108

[4] Richard A. Frost, Rahmatullah Hafiz, and Paul C. Callaghan. 2007. Modular and Efficient Top-down Parsing for Ambiguous Left-Recursive Grammars. In *Proceedings of the 10th International Conference on Parsing Technologies* (Prague, Czech Republic) *(IWPT '07)*. Association for Computational Linguistics, USA, 109–120. https://doi.org/10.5555/1621410.1621425

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., USA. https://doi.org/10.5555/186897

[6] Andy Gill and Simon Marlow. 1995. Happy: the parser generator for Haskell.

[7] Graham Hutton. 1992. Higher-order functions for parsing. *Journal of Functional Programming* 2, 3 (1992), 323–343. https://doi.org/10.1017/S0956796800000411

[8] Graham Hutton and Erik Meijer. 1996. *Monadic Parser Combinators.* Technical Report NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.1678

[9] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. 2016. Practical, General Parser Combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (St. Petersburg, FL, USA) *(PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2847538.2847539

[10] Mark Johnson. 1995. Memoization in Top-down Parsing. *Comput. Linguist.* 21, 3 (Sept. 1995), 405–417. https://doi.org/10.5555/216261.216269

[11] Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators For The Real World.* Technical Report. Microsoft. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5187

[12] Robert C. Moore. 2000. Removing Left Recursion from Context-Free Grammars. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference* (Seattle, Washington) *(NAACL 2000)*. Association for Computational Linguistics, USA, 249–255. https://doi.org/10.5555/974305.974338

[13] Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Parser combinators in Scala. *CW Reports* 54 (2008).

[14] Shayan Najd and Simon Peyton Jones. 2016. Trees That Grow. *CoRR* abs/1610.04799 (2016). arXiv:1610.04799 http://arxiv.org/abs/1610.04799

[15] Dung "Zung" Nguyen, Mathias Ricken, and Stephen Wong. 2005. Design Patterns for Parsing. *SIGCSE Bull.* 37, 1 (Feb. 2005), 477–481. https://doi.org/10.1145/1047124.1047497

[16] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf. https://doi.org/10.5555/2501720

[17] Axel-Tobias Schreiner and James Heliotis. 2008. Design Patterns in Parsing. (2008). https://scholarworks.rit.edu/other/82/ Presented at Killer Examples, a workshop at OOPSLA '08.

[18] S. Swierstra. 2000. Combinator Parsers: From Toys to Tools. *Electronic Notes in Theoretical Computer Science* 41 (01 2000), 38–59. https://doi.org/10.1016/S1571-0661(05)80545-6

[19] S. Doaitse Swierstra. 2009. *Combinator Parsing: A Short Tutorial*. Springer Berlin Heidelberg, Berlin, Heidelberg, 252–300. https://doi.org/10.1007/978-3-642-03153-3_6

[20] Jamie Willis and Nicolas Wu. 2018. Garnishing Parsec with Parsley. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala* (St. Louis, MO, USA) *(Scala '18)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/3241653.3241656

[21] Jamie Willis and Nicolas Wu. 2021. Design Patterns for Parser Combinators (Functional Pearl). In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell* (Virtual, Republic of Korea) *(Haskell 2021)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/3471874.3472984