Design Patterns for Parser Combinators (Functional Pearl)

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 are a popular and elegant approach for parsing in functional languages. The design and implementation of such libraries are well discussed, but having a well-designed library is only one-half of the story. In this paper we explore several reusable approaches to writing parsers in combinator style, focusing on easy to apply patterns to keep parsing code simple, separated, and maintainable.

CCS Concepts: • **Software and its engineering** → *Software design engineering*; **Functional languages**; **Parsers**; *Domain specific languages.*

Keywords: parser combinators

ACM Reference Format:
Jamie Willis and Nicolas Wu. 2021. Design Patterns for Parser Combinators (Functional Pearl). In *Proceedings of the 14th ACM SIGPLAN International Haskell Symposium (Haskell '21), August 26–27, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3471874.3472984

1 Introduction

Design patterns, popularised by Gamma et al. [9], are software design principles that are not necessarily rigid or must be adhered to but are a guide for solving common problems and structuring large bodies of code. Their most prolific use is within the Object-Oriented Programming (OOP) community; within the Functional Programming (FP) community, many patterns are simply implemented using higher-order functions. In fact, one example of the strength of higher-order functions is the development of combinator libraries, and in particular *parser combinators* [7, 14, 15, 19, 27, 29, 31, 32]. But while FP provides many of these beautiful abstractions, not enough is said about how to actually use them in a maintainable and scalable way. Indeed, this paper aims to highlight several parser combinator design patterns; these patterns, certainly not exhaustive, should:

- Structure and organise larger parsers
- Separate the various concerns of different parts of parsers
- Keep the intention and shape of the grammar clear
- Create informative error messages
- Guide the implementation with strong types

As mentioned, parser combinators are an elegant functional approach to performing parsing of grammars, including context-sensitive, embraced by many in the Haskell community. Compared to parser generator tools, like Haskell's `Happy`, parsers developed with combinators are written in pure Haskell as a Domain-Specific Language. This paper assumes some knowledge of parser combinators and the tutorial by Swierstra [28] serves as a good introduction.

The most ubiquitous family of combinators in Haskell is the `parsec` [19] family: consisting of the libraries `parsec`, `attoparsec`, and `megaparsec`. This family is primarily characterised by their shared semantics for backtracking, where alternative parts of a grammar may only be taken when input has not been consumed in another; in particular they all leverage the `try` combinator to opt-in to backtracking, as opposed to a cut combinator to opt-out. Practically, this means there are some considerations when implementing some of our patterns within this family, concerning `try`, that do not occur in other libraries.

The presentation of our patterns will be anchored around a main running example developed in a generic and simple implementation of a `parsec` family library in Haskell. The aim is to provide a solid foundation without having to focus on the specifics of any particular API or their individual quirks. Importantly, our patterns are not only useful for `parsec` or necessarily Haskell and apply generally.

1.1 An Introductory Example

The classic example used to demonstrate how to use parser combinators is some variant of an expression language, the grammar for which is shown in Figure 1. To start with, this language supports: the standard arithmetic operators, each of which is left-associative, denoted by the left recursion; numbers; variables; parenthesised expressions; and prefix negation operator `negate`, modelled by the following AST:

Haskell '21, August 26–27, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8615-9/21/08.
https://doi.org/10.1145/3471874.3472984

⟨*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*⟩

Figure 1. Grammar

```
data Expr = Num Int          | Var String
          | Neg Expr         | Mul Expr Expr
          | Add Expr Expr    | Sub Expr Expr
```

The parser will directly construct this datatype: combinator libraries can incorporate semantic actions directly into the parser itself. For now, the AST is monolithic and homogeneous, but ideally, it would itself mirror the grammar. The corresponding parser for the grammar is given in Figure 2. This parser maps closely to the original grammar: rule alternatives are expressed with the ‹|› "or" combinator, and semantic actions and sequencing are applied with the standard Applicative ‹$› and ‹*› combinators, pronounced "fmap" and "ap" respectively. The Kleene star and plus operations are implemented with Alternative many and some respectively.

The Combinators. For reference, here is a summary of the combinators used in this paper, along with their types:

```
(<$>)   :: Functor f => (a -> b) -> f a -> f b
(<$)    :: Functor f => a -> f b -> f a

pure    :: Applicative f => a -> f a
(<*>)   :: Applicative f => f (a -> b) -> f a -> f b
(<*)    :: Applicative f => f a -> f b -> f a
(*>)    :: Applicative f => f a -> f b -> f b
(<**>)  :: Applicative f => f a -> f (a -> b) -> f b
(<:>)   :: Applicative f => f a -> f [a] -> f [a]
```

This first set of combinators are responsible for sequencing actions (in this case parsing actions) and combining their results in a way that follows the combinator's type signature: ‹*›, for instance, will apply the function returned by the first action to the value returned by the second. Notably, pure does nothing in a parsing sense except return a result.

```
(<|>)   :: Alternative f => f a -> f a -> f a
many    :: Alternative f => f a -> f [a]
some    :: Alternative f => f a -> f [a]
choice  :: Alternative f => [f a] -> f a
```

```
digit  = oneOf ['0'..'9']
number = foldl addDigit 0 <$> some digit
ident  = alpha <:> many alphaNum
expr   = Add <$> expr <*> (char '+' *> term)
       <|> Sub <$> expr <*> (char '-' *> term)
       <|> term
term   = Mul <$> term <*> (char '*' *> negate) <|> negate
negate = Neg <$> (string "negate" *> negate) <|> atom
atom   = char '(' *> expr <* char ')'
       <|> Num <$> number <|> Var <$> ident
addDigit n d = n * 10 + digitToInt d
```

Figure 2. Parser

The next set of combinators are responsible for choice and data-independent branching: ‹|› will try parsing its first argument, if it fails then the second argument is tried (in some libraries, only if the first consumed no input). Both many and some are built on top of this and ‹:›, to try performing an action multiple times until it fails, collecting the results into a list, with many requiring zero or more successes, and some requiring one or more. The choice combinator tries each action in the list in turn, until one succeeds, using ‹|›.

The canonical form for terms using these operators is sequencing operations separated by alternative operations: usually achieved by making the ‹|› combinator **infixl** 3 and the rest **infixl** 4, so that ‹|› binds weaker than the others.

```
char   :: Char -> Parser Char
string :: String -> Parser String
oneOf  :: [Char] -> Parser Char
try    :: Parser a -> Parser a
```

The final group of combinators are specific to parsers: char parses a specific character; string parses a specific string of characters one after the other; and oneOf is a character class, using choice to parse any of the provided characters. The try combinator in the `parsec` family undoes input consumption on failure, allowing ‹|› to take its second branch.

1.2 The Common Problems with Combinators

Whilst the example parser perfectly maps to the grammar for the language it, in fact, exhibits several common problems:

Left-Recursive Expressions. The biggest problem with this parser is that it is *left-recursive*. For many parser combinator libraries, this will cause infinite recursion at runtime since the recursion is unguarded by input consumption.

Instead of using traditional grammar transformations on both the grammar and the parser to *left-factor* [1, 21] it, our first pattern addresses it idiomatically, whilst introducing extra type safety (Section 2).

In-Place Lexing. The example parser does not make any attempt to parse whitespace, and there are some counter-intuitive parses possible from poor lexing.

To combat this, whitespace handling and general practices of lexing are discussed (Section 3), and measures are introduced to abstract them, further refining the design.

Bookkeeping Information. The design does not lend itself well to changing requirements: suppose position information must be added to the AST, the parser would be altered in a way that obscures its underlying purpose and structure.

As a result, further separation is introduced between the semantic action of the parser (in this case AST construction) and the parsing logic that represents the grammar (Section 4). The result of this pattern will be code that is robust to changes in AST requirements and more effectively separates the concerns of the code. This will be illustrated by extending the grammar to handle assignments and statements.

Helpful Errors. To show how to improve errors, the grammar is extended again with conditional statements, and both positive and negative lookahead will be leveraged to make bespoke error messages for the user (Section 5).

Related work is discussed (Section 6) and the parser's development is summarised and reflected on at the very end of the journey, with some closing remarks (Section 7).

2 Expression Parsing

Expression parsers have a very standardised shape in a grammar. Consider the following grammar:

⟨*pred*⟩ ::= ⟨*comp*⟩ '&&' ⟨*pred*⟩ | ⟨*comp*⟩
⟨*comp*⟩ ::= ⟨*expr*⟩ '<' ⟨*expr*⟩ | ⟨*expr*⟩ '=' ⟨*expr*⟩ | ⟨*expr*⟩
⟨*expr*⟩ ::= ⟨*expr*⟩ '+' ⟨*term*⟩ | ⟨*expr*⟩ '-' ⟨*term*⟩ | ⟨*term*⟩
⟨*term*⟩ ::= ⟨*term*⟩ '*' ⟨*atom*⟩ | ⟨*atom*⟩
⟨*atom*⟩ ::= '(' ⟨*expr*⟩ ')' | ⟨*number*⟩ | ⟨*ident*⟩

This consists of various rules each referring to the next, denoting the precedence of each operator (⟨*term*⟩ is tighter than ⟨*expr*⟩). When two operators appear in the same rule they share the same precedence (like < and =; or + and -). Usually, the location of recursion – or its absence – denotes the associativity – or lack thereof – of the operator. Recursion on the left is left-associative (as in ⟨*expr*⟩), on the right is right-associative (as in ⟨*pred*⟩), and no self-recursion is non-associative (as in ⟨*comp*⟩). Most parser combinator libraries operate as recursive descent including the `parsec` family: this means that left recursion results in non-termination.

> ***Problem 1: Left-Recursive Expressions.*** Expressions with left recursion cannot be encoded by recursive descent parsers and will diverge.

One solution to handling left recursion in a grammar is to change the grammar using left-factoring, however, it is preferable to leave the grammar alone: the grammar should not have to be tailored to the implementation.

> ***Anti-pattern 1: Grammar Refactoring.*** Modifying the grammar to remove left recursion exposes implementation details and complicates the grammar.

2.1 The Homogeneous Chain Combinators

Hutton and Meijer [15] discuss the classic technique of replacing left recursion with iteration or recursion with an accumulating parameter. The resulting idiomatic combinators are known traditionally as the chain combinators. Conventionally, parser combinator libraries usually define two sorts of chains: `chainl1` and `chainr1`.

```
chainl1 :: Parser a → Parser (a → a → a) → Parser a
chainr1 :: Parser a → Parser (a → a → a) → Parser a
```

The chain*x*1 p op combinator should parse *one* or more ps, separated by ops, applied *x*-associatively.

> ***Pattern 1a: Homogeneous Chains.*** For binary operators where the associativity is not specified, use `chainl1` or `chainr1` to combine operands with their operators.

```
expr = chainl1 term (Add <$ char '+' <|> Sub <$ char '-')
term = chainl1 negate (Mul <$ char '*')
```

This parser, in conjunction with the original definitions for negate, atom, number, and ident now works correctly and also does not backtrack. However, the use of the *Homogeneous Chains* pattern implies that the grammar does not specify the associativity like, for example, the following:

⟨*func*⟩ ::= ⟨*func*⟩ '.' ⟨*func*⟩ | ⟨*lambda*⟩

The recursion on both sides of the "." operator in the ⟨*func*⟩ rule means that it is associative, without specifying whether it is to the left or the right. This makes the rule a great candidate for the *Homogeneous Chains* patterns, since the concrete associativity is left as an implementation detail: since `chainl1` and `chainr1` share the same type, changing the associativity would be seamless. With our example, however, the grammar does specify the exact associativity of the operators, so the ease of exchanging one chain for the other can allow the parser to be unfaithful to the grammar.

2.2 The Heterogeneous Chain Combinators

The problem with using the conventional chains introduced in Section 2.1 for our grammar is that they fail to distinguish (except lexically) between each other. It is very easy to accidentally use the wrong one and silently change the meaning of the parser and make it unfaithful. Instead, we offer two new chains with refined types along with their definitions:

```
infixl1 :: (a → b) → Parser a
        → Parser (b → a → b) → Parser b
infixl1 wrap p op = (wrap <$> p) <**> rest
  where rest = flip (·) <$> (flip <$> op <*> p) <*> rest
               <|> pure id
```

```
infixr1 :: (a → b) → Parser a
        → Parser (a → b → b) → Parser b
infixr1 wrap p op =
  p ‹**› (flip ‹$› op ‹*› infixr1 wrap p op ‹|› pure wrap)
```

The type of the chains in this formulation make it much clearer which is which: the operators produce bs more tightly in the correct position. This comes at an ergonomic cost, since a wrapping function that transforms the terminal item in the chain into the correct type must be provided. Like chainl1 and chainr1, their definitions generalise the shape of left- and right-factored parsers, providing a recipe for how to transform the grammars by hand, though a strength of combinators is allowing these higher-order parsing recipes to be defined and used instead. These new chains are related to the classic versions:

```
chainl1 = infixl1 id
chainr1 = infixr1 id
```

Additionally, postfix and prefix serve as a natural extension:

```
postfix :: (a → b) → Parser a
        → Parser (b → b) → Parser b
postfix wrap p op = (wrap ‹$› p) ‹**› rest
  where rest = flip (·) ‹$› op ‹*› rest ‹|› pure id
prefix :: (a → b) → Parser (b → b)
       → Parser a → Parser b
prefix wrap op p = op ‹*› prefix wrap op p ‹|› wrap ‹$› p
```

These chain combinators handle many applications of postfix operators or prefix operators to a terminal item. In particular, the definition of postfix is very close in shape to infixl1's, indeed, infixl1 can be easily given in terms of postfix:

```
infixl1 wrap p op = postfix wrap p (flip ‹$› op ‹*› p)
```

> ***Pattern 1b: Heterogeneous Chains.*** For associative operators where operand types may differ, use infixl1 or infixr1 to combine operands with their operators, in conjunction with strongly typed semantic actions.

To properly leverage the additional type-safety provided by the new heterogeneous chains, the AST itself must change:

```
data Expr   = Add Expr Term | Sub Expr Term
            | OfTerm Term
data Term   = Mul Term Negate | OfNegate Negate
data Negate = Neg Negate | OfAtom Atom
data Atom   = Num Int | Var String | Parens Expr
```

This datatype more accurately describes the shape of the grammar; this is good because it provides a second layer with which to check that the parser is correct. As an added benefit, functions that consume this datatype can rely on the shape of the AST being left- or right-associated. The parser now has to be adapted to the new type:

```
expr  = infixl1 OfTerm term
                (Add ‹$ char '+' ‹|› Sub ‹$ char '-')
term = infixl1 OfNegate negate (Mul ‹$ char '*')
negate = prefix OfAtom (Neg ‹$ string "negate") atom
atom   = char '(' *› (Parens ‹$› expr) ‹* char ')'
       ‹|› Num ‹$› number ‹|› Var ‹$› ident
```

This new parser conforms to the new datatype and, as such, if the programmer accidentally switches an infixl1 for an infixr1, this parser would fail to typecheck. Unfortunately, the same level of guarantee is not present for prefix and postfix, other than their reversed arguments. Unlike the previous, homogeneous, version of the parser, the "recursive" point in atom has to be wrapped in the Parens constructor.

2.3 Generalising to Precedence Tables

Section 2.1 introduced chainl1 as a way of managing left recursion in a grammar, refactored the parser to eliminate left recursion, and introduced heterogeneous chains to support datatypes that encode the grammar more precisely. This is a good first step but there is still a lot of mechanical busywork to encode the precedence and associativity of operators, not to mention the daunting prospect of running out of four-letter parser names as the grammar expands! Parser generator tools often expose a more concise and scalable representation of expression parsers: this same experience can be developed for parser combinators.

A precedence combinator accepts a table of operator precedence along with a base "atom". This is a combinator that is found, in some shape or form, in most parser combinator libraries – including the parsec family – as well as in the literature [3, 12, 13]. This table is normally implemented as a list of some Op datatype that expresses one or more operators of some associativity and fixity. However, [Op a] would imply a homogeneous table, and this is not desirable for the heterogeneous parser in Section 2.2. A heterogeneous list will fit better:

```
data Fixity a b sig where
  InfixL  :: Fixity a b (b → a → b)
  InfixR  :: Fixity a b (a → b → b)
  InfixN  :: Fixity a b (a → a → b)
  Prefix  :: Fixity a b (b → b)
  Postfix :: Fixity a b (b → b)
data Op a b where
  Op :: Fixity a b sig → (a → b) → Parser sig → Op a b
data Prec a where
  Level :: Prec a → Op a b → Prec b
  Atom :: Parser a → Prec a
(⋊) = Level      --infixl 5
(⋉) = flip (⋊)   --infixr 5
```

The Fixity datatype relates the input a with the output b of an operator, given by the type sig. The InfixN constructor represents non-associative operators, which can appear at most once[1]. The Fixity datatype is useful since it detaches any potential functions building on Op from needing to worry about considering every specialised fixity. The Op datatype is a defunctionalised representation of a heterogeneous chain, partially applied to the operator but not the atom. The list-like Prec structure combines smaller precedence tables with a new layer, connected by the Op. To make this more ergonomic, the (⤚) and (⤙) operators allow the table to be built from strongest to weakest *or* weakest to strongest[2]. A precedence combinator is a "fold" over a table:

```
precedence :: Prec a → Parser a
precedence (Atom atom) = atom
precedence (Level lvls ops) = con (precedence lvls) ops
  where con :: Parser a → Op a b → Parser b
        con p (Op InfixL wrap op)  = infixl1 wrap p op
        con p (Op InfixR wrap op)  = infixr1 wrap p op
        con p (Op InfixN wrap op) =
          p ‹**› (flip ‹$› op ‹*› p ‹|› pure wrap)
        con p (Op Prefix wrap op)  = prefix wrap op p
        con p (Op Postfix wrap op) = postfix wrap p op

precHomo :: Parser a → [Op a a] → Parser a
precHomo atom = precedence · foldl (⤚) (Atom atom)
```

The idea is to traverse the table from the deepest layer outwards, converting each operator in turn into the corresponding chain. The homogeneous precedence parser precHomo can easily be recovered as a fold over a regular list.

Using helper functions can make the Op datatype easier to use by providing common wrapping functions: these are id and functions that implement a sort of sub-typing.

```
gops :: Fixity a b sig → (a → b) → [Parser sig] → Op a b
gops fixity wrap = Op fixity wrap · choice

ops :: Fixity a a sig → [Parser sig] → Op a a
ops fixity = gops fixity id

class sub < sup where
  upcast   :: sub → sup
  downcast :: sup → Maybe sub

sops :: a < b ⇒ Fixity a b sig → [Parser sig] → Op a b
sops fixity = gops fixity upcast
```

The gops function takes many operators at the same level and combines them into one with choice. The ops function supports homogeneous operators with id. The sops function uses upcasting to perform wrapping: in Functional-OOP languages, datatype hierarchies are often made using subtype polymorphism to avoid explicit wrapper constructors. This is mimicked by a < b: a function is designated as the cast.

> ***Pattern 1c: Precedence Tables.*** For expressions, use the precedence combinator to deal with both fixity and precedence concisely.

By giving each layer of the AST its own (<) instance – using the Of*X* constructors – a simpler and more concise definition of expr can be given using precedence and sops:

```
expr = precedence $
  sops InfixL [Add ‹$ char '+', Sub ‹$ char '-'] ⤙
  sops InfixL [Mul ‹$ char '*']                  ⤙
  sops Prefix [Neg ‹$ string "negate"]           ⤙
  Atom atom
```

By using sops, the wrapper constructors in the original parser are avoided as they are resolved implicitly. Like the version with heterogeneous chains, adding, removing, or altering any layers will generate a type error[3]. Any alterations to the parser need to be reflected in the datatype itself.

2.4 Aside: Folds for Parsers

The left chain (Section 2.1) is a conversion from recursion to iteration or, in this case, recursion using an accumulating parameter (realised here by composing functions). The relation between iteration with fold and recursion with accumulation is an instance of deforestation [10, 30]. The idea is to fuse the building and the consumption of an intermediate structure – in this case lists – to eliminate it. In fact, in the existing parser, there is an example of a structure that is built up and immediately crushed back down:

```
number = foldl addDigit 0 ‹$› some digit
```

The combinator some returns a list of results, but this list is then folded immediately: this is wasteful. Happily, the chains are useful for trimming the forest, in particular postfix, prefix, and infixl1 can be used to create so-called parser folds.

```
manyr :: (a → b → b) → b → Parser a → Parser b
manyr f k p = prefix id (f ‹$› p) (pure k)

manyl :: (b → a → b) → b → Parser a → Parser b
manyl f k p = postfix id (pure k) (flip f ‹$› p)

somer :: (a → b → b) → b → Parser a → Parser b
somer  f k p = f ‹$› p ‹*› manyr f k p

somel :: (b → a → b) → b → Parser a → Parser b
somel  f k p = infixl1 (f k) p (pure f)
```

Just as foldr (:) [ ] is the identity fold, manyr (:) [ ] ≡ many and somer (:) [ ] ≡ some. Here, the heterogeneous infixl1 can be used with the wrapping function f k representing the initial application of the accumulator to the first item in the fold: this would not be possible with the homogeneous

[1] Notice that b appears in neither the operator's left nor right positions.
[2] The operators are eating the levels with the higher precedence.
[3] In fact, this mechanism successfully caught a typo in this example parser!

chainl1. The relation between a deforested parser fold and the "forested" fold with iterative combinator is as follows:

```
manyr f k p = foldr f k ‹$› many p
manyl f k p = foldl f k ‹$› many p
somer f k p = foldr f k ‹$› some p
somel f k p = foldl f k ‹$› some p
```

With this in mind, the definition of number can be simplified:

```
number = somel addDigit 0 digit
```

While there is little aesthetic difference between the old version and the new one, the second is more efficient as it does not have to build a list, instead consuming its elements *in situ*. Really, it serves to highlight the flexibility of chains and their applicability in a variety of different scenarios.

Discussion. The issues of left recursion and organising expression parsers can be cleanly eliminated with the help of precedence and chains. On the surface, it appears as if precedence is a clear win, however, as illustrated by Section 2.4, chains are more versatile than they might first appear, and, arguably, precedence is overkill for only a single layer. This appears in practice from time to time: grammars where ";" is considered an operator, for instance.

3 Effective Lexing

When writing parsers with a parser generator tool, there is often a distinction between the lexical analysis and the parsing stages. This is often exemplified by having two distinct tools: `alex` and `happy`, for instance. With parser combinators, however, the distinction is much less clear but no less important to consider. Even though the tool used for lexing and parsing is the same, clean and well-separated code leads to more maintainable and readable parsers.

The parser refined in the previous section still has some issues. Some easy ones to see are the following (where Left represents failure and Right represents success):

```
parse expr "x + 7"          ≡ Right (Var "x")
parse (expr ‹* eof) "x + 7" ≡ Left "(1, 2): unex[..]"
parse expr "negatex"        ≡ Right (Neg (Var "x"))
```

All three of these problems are caused by the absence of proper lexing parsers and whitespace handling: `"negatex"` should be treated as a single identifier, the parser is not greedy, and it cannot consume spaces. The naïve solution would be to insert whitespace consuming logic everywhere it is necessary straight into the parser: this is very noisy.

> ***Problem 2: In-Place Lexing.*** Dealing with tokens while parsing is intrusive to the overall structure of the parser and introduces clutter.

Traditionally, parsers are designed with two phases: lexing and then parsing. The idea is to build up a stream of tokens instead of working on raw characters. This allows reading whitespace and chunking the input to be abstracted away from the grammar. Running a lexing pass upfront has the disadvantage that tokens will be generated greedily and missing any contextual information about where the token may lie relative to the grammar: a `'-'` might represent a subtraction or be part of an integer literal but the lexer cannot know which, so the choice is often deferred to the parser and unary negation typically replaces negative literals.

> ***Anti-pattern 2: Lexing then Parsing.*** Preprocessing the input with a dedicated lexer has no contextual awareness with which to selectively construct tokens.

3.1 Dealing with Whitespace

The most pressing issue to fix with the parser is whitespace. This is not difficult, but there are a couple of considerations:

1. Whitespace should be read uniformly
2. Whitespace should be unintrusive to the rest of the parser

In particular, (2) will be properly addressed in Section 3.3, however (1) can be addressed now. The meaning of *uniform* in this context is to establish a convention on exactly where whitespace is read. A common temptation that newcomers make when writing parsers is to always consume whitespace *around* any given token (or worse between any two combinators); this is not ideal, since, if done "properly", this technique will double up on reading whitespace. This is fairly benign as it is only wasteful computationally – the second attempt at reading whitespace will always consume nothing; but, for efficiency, there are two options: always consume leading whitespace, or always consume trailing whitespace (and ultimately consume in the opposite direction exactly once at the end or beginning respectively).

At first glance, it might appear as if both approaches are equivalent to each other, however, that is not the case. In fact, reading leading whitespace has a few flaws that trailing whitespace does not. Firstly, it is inefficient since it leads to excessive backtracking. Secondly, in the `parsec` family this falls afoul of (2): taking the second branch of an ‹|› is conditional on the first having not consumed any input so to perform backtracking `try` combinator must be used. Ultimately, this means that whitespace is already intrusive to the rest of the parser, since `try` combinators must be inserted on the first branch of every ‹|› where whitespace is consumed. The third reason to avoid using the leading approach is that it can make acquiring position information from the parser more difficult, since the reported positions will be *before* the whitespace in front of the relevant token.

As a result, it is much better to deal with trailing whitespace and read whitespace once at the very beginning of the parser. If every token parser consumes whitespace after performing its duties, whitespace will be handled correctly.

Pattern 2a: Whitespace Combinators. Build a lexeme combinator dedicated to consuming *trailing* whitespace after a given lexeme. Build a fully combinator to consume initial whitespace and the end of input.

For the expression grammar, actually parsing whitespace will be easy, since there are no comments or special whitespace described in the language:

```
whitespace :: Parser ()
whitespace = () <$ many (satisfy isSpace)

fully      :: Parser a → Parser a
fully p    = whitespace *> p <* eof

lexeme     :: Parser a → Parser a
lexeme p   = p <* whitespace
```

The whitespace parser is responsible for reading zero or more space-like characters and comments[4]. The fully combinator is used to treat the top-level parser as a greedy unit that is required to reach the end of input, whilst also consuming the first chunk of whitespace. The lexeme combinator should be used to wrap up any token parsers to ensure they consume the *trailing* spaces. Ideally, these parsers should be kept in a module separately from the main grammar, to cleanly encapsulate them: after all, the token parsers exposed to the main parser should all deal with whitespace themselves.

3.2 Tokens

When working with a lexer where tokens are generated on demand, lexing can be context-aware meaning that certain tokens are only demanded within certain grammar rules only. The disadvantage to this approach, however, is it may require several lexes of a single token when the parser backtracks. However, ideally, parsers should be constructed to minimise backtracking, so the favoured approach here will be to combine lexing and parsing into a single pass.

Tokens and Atomicity. The key consideration when creating lexing parsers with parser combinators is that they should be *atomic* so that if reading a token fails it consumes no input at all: it is either all or nothing. This allows the parser to try matching a similar token without the grammar needing to worry about performing any backtracking. In practice, backtracking is still performed, but it should be performed immediately as opposed to waiting until an alternative branch is taken. For libraries with some form of cut operation, this means cuts can often be inserted after tokens. In the `parsec` family, however, backtracking of this kind is done with the `try` combinator. Thankfully, `try` is very cheap when the parser succeeds, so the lexing parsers will use it liberally: it does not even matter if the token is already atomic (which is the case for single-character tokens). With that in mind here is the first relevant combinator:

```
token :: Parser a → Parser a
token = lexeme · try
```

This combinator can be used to make a given parser into a token, which is to say that it consumes any trailing whitespace and is completely atomic. In the `parsec` family, it is important to not put the whitespace parsing within the scope of the `try`: this would mean that if there were a syntax error caused by the whitespace itself, it would needlessly backtrack for a while before the parser finally dies. This could be the case, for instance, if the language contained comments.

Pattern 2bi: Tokenizing Combinators. Annotate terminals with a token combinator, built on lexeme, to atomically parse them with whitespace consumed.

Using the token combinator, the primitive tokens can be wrapped up and the whitespace problems fixed. As a reminder, here is the parser up to this point:

```
alpha    = oneOf (['a'..'z'] ++ ['A'..'Z'])
digit    = oneOf ['0'..'9']
alphaNum = alpha <|> digit

number   = somel addDigit 0 digit
ident    = alpha <:> many alphaNum

expr  = precedence $
  sops InfixL [Add <$ char '+', Sub <$ char '-'] ⪪
  sops InfixL [Mul <$ char '*']                  ⪪
  sops Prefix [Neg <$ string "negate"]           ⪪
  Atom atom
atom = char '(' *> (Parens <$> expr) <* char ')'
     <|> Num <$> number <|> Var <$> ident
```

The question is which of these are tokens, and which of them are not. This is largely subjective, but here alpha, digit, and alphaNum are treated as building blocks whereas number, ident, parentheses, and the operators are tokens.

```
number = token (somel addDigit 0 digit)
ident  = token (alpha <:> many alphaNum)

expr  = precedence $
  sops InfixL [Add <$ token (char '+'),
               Sub <$ token (char '-')]          ⪪
  sops InfixL [Mul <$ token (char '*')]          ⪪
  sops Prefix [Neg <$ token (string "negate")]   ⪪
  Atom atom
atom = Parens <$>
       (token (char '(') *> expr <* token (char ')'))
     <|> Num <$> number <|> Var <$> ident
```

By ensuring all of the terminals of the grammar have been marked with token, no other whitespace handling needs to be performed for the expr in the parentheses. This has addressed the original problem so that parse expr `"x + 7"` now returns a correct successful result.

[4] Learning from Section 2.4, manyl const () (satisfy isSpace) is better.

Token Validation. The previous parser has correctly handled each of the tokens within the grammar. However, using string for `"negate"` is inappropriate: it fails to enforce any separation between tokens. A better approach is to develop a distinct method for handling keywords.

```
keyword    :: String → Parser ()
keyword k = token (string k ∗› notFollowedBy alphaNum)

keys :: [String]
keys = ["negate"]
```

The keyword combinator is simple: it will parse the given string, but then ensure that it is not followed by another valid identifier character. Care must be taken to consume whitespace and make it atomic *after* the validation.

> ***Pattern 2bii: Keyword Combinators.*** Avoid using string with token for keywords. Use a keyword combinator that enforces that the keyword does not form a valid prefix of another token.

By substituting token · string for keyword to handle the `negate` operator, parse expr `"negatex"` correctly returns Right (Var `"negatex"`). A similar system can be devised for longest-match operators, but there is no potential ambiguity with operators in the example parser.

3.3 Using `OverloadedStrings` as a Facade

While Section 3.1 provided a rationale for how to robustly handle whitespace in a grammar, and Section 3.2 neatly encapsulated the combination of whitespace logic with other properties of the tokens, the solutions can justifiably be accused of polluting the parser. Indeed, a suggested property of whitespace parsing was that it should not be intrusive to the main body of parser, but with the current setup, the parser is littered with tokens and keywords.

The Haskell `OverloadedStrings` extension allows the regular Haskell syntax for string literals to represent something else entirely. In other words, if there is an instance of the IsString type class available for a type s, string literals can represent values of type s implicitly.

```
{-# LANGUAGE OverloadedStrings #-}
class IsString s where fromString :: String → s
```

> ***Pattern 2c: Overloaded Strings.*** Hide tokenizing logic by allowing string literals to serve as parsers.

The IsString type class is of particular interest where lexing is concerned, with a valuable instance being one for Parser () (to help GHC's constraint solver, u~() is used):

```
instance u~() ⇒ IsString (Parser u) where
  fromString str
    | elem str keys = keyword str
    | otherwise     = () ‹$ token (string str)
```

This magical instance encapsulates *both* the use of token and the use of keyword in the parser. Except for the tokens number and ident themselves (which should be separated), the rest of the parser can be stripped of its lexing baggage:

```
expr  = precedence $
  sops InfixL [Add ‹$ "+", Sub ‹$ "-"] +<
  sops InfixL [Mul ‹$ "∗"]             +<
  sops Prefix [Neg ‹$ "negate"]        +<
  Atom atom
atom = "(" ∗› (Parens ‹$› expr) ‹∗ ")"
     ‹|› Num ‹$› number ‹|› Var ‹$› ident
```

Discussion. The final result of the lexing transformation is striking as it ends up eliminating noise from before lexing was even incorporated in: the original char combinators have been removed. This keeps it looking closer to the original grammar in form, more so for non-precedence grammars. Preferably, the combinators and non-string tokens themselves should be kept in another module.

In practice, parser combinator libraries often support some form of "lexer combinator generator", where a specification of the language's tokens are given and out pops the combinators to parse them. This mechanism, where it exists, will also handle whitespace just as described in Section 3.1, but there is still value in understanding the justification behind the canonical implementations.

4 Abstracting Asᴛ Construction

A common realisation after writing a working parser is that the Asᴛ produced during parsing may need to be augmented with extra information from the parse: this can be any information that might be required for a later part of the overall pipeline, often the line and column numbers. This can be a frustrating realisation, as the parser and the data type needs to be modified to accommodate the new requirements, and the bookkeeping required to patch the parser is intrusive, undoing all the work of the other patterns!

> ***Problem 3: Bookkeeping.*** Asᴛs built during parsing occasionally require parser metadata.

To demonstrate both the problem and a taste of the damage done, the grammar will be augmented to now include assignments and statements. For the sake of illustration, the requirement is that variables and assignments require position information, so that, at a later point in the program, scoping errors can be reported referencing the original positions of the offenders. The new rules in the grammar are:

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

In the spirit of the type safety adopted in Section 2, the new datatypes will mirror the structure of these rules:

```
data Stmt = Seq Asgn Stmt | OfAsgn Asgn
data Asgn = Asgn String Expr (Int, Int)
data Atom = Num Int | Var String (Int, Int) | Parens Expr
```

As discussed, the Asgn constructor has been given an extra argument to accommodate the extra position information required for scope errors. The same has been done for the Var constructor in Atom. The implementation of the parser is familiar, making use of infixr1 to handle statements:

```
stmt = infixr1 OfAsgn asgn (Seq <$ ";")
asgn = pos <**> (Asgn <$> ident <* ":=" <*> expr)
atom = "(" *> (Parens <$> expr) <* ")"
     <|> Num <$> number <|> pos <**> (Var <$> ident)
```

The atom parser is the only parser that needs to change to accommodate the new requirements. The new intrusion into the parser is the pos :: Parser (Int, Int) combinator, which yields the required information. Perhaps interestingly, the combinator is merged in a somewhat counter-intuitive ordering, applied on the left to the rest of the partial constructor. This is because the position has been placed at the end of the constructor, but positions should be obtained *before* any tokens have been read otherwise they will point at the next token after the variable or assignment. This serves as further justification of the principle that only trailing whitespace should be consumed. This has already introduced noise into the parser, if position information were required for the operators as well, the problem would get much worse.

Anti-pattern 3: Inline Bookkeeping. Incorporating metadata inline into the parser is intrusive and brittle.

4.1 Smart Constructors for Parsers

Smart constructors are a well-known Haskell technique for augmenting a datatype with additional builder functions to:

- Define compound structures built of core constructors
- Perform light-weight validation on constructor inputs to guarantee invariants hold true
- Perform light-weight optimisations on constructors that compose their arguments
- Simplify the API by providing default values
- Processing constructor arguments into normal forms

They are simply regular Haskell functions, often with a similar name to the constructor they are abstracting and the prefix mk (or make). In general, given a regular constructor $C :: A_1 \to \ldots \to A_n \to T$, a smart constructor typically has the form $mkC :: B_1 \to \ldots \to B_m \to T$ where the required arguments $A_1 \ldots A_n$ are synthesised from other values $B_1 \ldots B_m$ and used to instantiate C to make a T. A simple but effective concept, they appeared in Section 2.3: the smart constructors ops and sops both helped to simplify the API by providing default wrappers; and gops extended the Op constructor by combining a list of parsers with choice.

When parsing it is convenient to have a *lifted* smart constructor of shape $mkC :: \text{Parser } A_1 \to \ldots \to \text{Parser } A_n \to \text{Parser } T$, since the value C is built during parsing. When the arguments are plainly parsed then combined in sequence, the constructor can be implemented by $\text{lift}A_n\ F$, which abstracts away the <$> and <*> normally required to combine results. However, smart constructors can also be made to perform any of three common tasks:

- Perform bookkeeping as the AST is built
- Perform semantic validation on the AST nodes
- Perform normalisation on the AST nodes

All of these are properties of the underlying constructor, and may or may not be required. An example of bookkeeping may be extracting position information (this is possible since the smart constructor operates in the parsing effect), an example of semantic validation might be to ensure that integers do not overflow and normalisation might arbitrate between two ambiguous constructions without backtracking.

Pattern 3a: Lifted Constructors. Use smart constructors to decouple bookkeeping logic from the parser.

The first of the three advertised properties above can immediately address the issues with the current parser. The smart constructors for Asgn and Var should both handle position tracking, but the others, for instance, Num, need do nothing but lift the underlying constructor:

```
mkAsgn :: Parser String → Parser Expr → Parser Asgn
mkAsgn var body = pos <**> (Asgn <$> var <*> body)
mkVar var = pos <**> (Var <$> var)
mkNum n = Num <$> n
mkParens x = Parens <$> x
asgn = mkAsgn (ident <* ":=") expr
atom = "(" *> mkParens expr <* ")"
     <|> mkNum number <|> mkVar ident
```

The parsers now have been simplified further: the work needed to combine various results has been abstracted into the smart constructors, and the parser does not need to be aware of what additional work needs to be performed on these sub-parts. The advantage to this approach is that if the position information was no longer needed, it can be removed without changing the parser: this adheres nicely to the *Single-Responsibility Principle* of Software Engineering [4, 20, 23] as the parser only cares about building results, not how they are built, which is the sole job of the constructors.

The *Lifted Constructors* pattern works well for linear segments of parser, where the arguments to constructors are situated next to the application of the constructor itself. But this is not always the case: consider the constructors for the operators in the precedence table. In these instances, the same principle can be applied, but in a different shape.

Pattern 3b: Deferred Constructors. Defer the construction of an AST node to abstract the bookkeeping when its arguments are not immediately available.

Here constructors are returned by parsers so that their arguments can be applied to them later but without any required metadata in the type. In the basic case, the constructor can be returned with pure:

```
mkAdd :: Parser (Expr → Term → Expr)
mkAdd = pure Add

mkNeg :: Parser (Negate → Negate)
mkNeg = pure Neg
```

While it might appear like this is not particularly useful, it still means that if position information needs to be added to a node, the parser does not need modification, for example:

```
mkMul :: Parser (Term → Negate → Term)
mkMul = (λp x y → Mul x y p) <$> pos
```

In this case, the position can be read immediately and applied ahead of time, with the other arguments deferred to later. The remainder of the parser is modified as follows:

```
expr = precedence $
  sops InfixL [mkAdd <* "+", mkSub <* "-"] ⊣
  sops InfixL [mkMul <* "*"]               ⊣
  sops Prefix [mkNeg <* "negate"]          ⊣
  Atom atom
```

The intrusion here is minimal, but the flexibility increased: this is more robust to change than any previous version.

Discussion. The smart constructor pattern is ultimately a simple one, but very flexible. It allows the parser maintainer to improve the separability of their code and removes yet more combinator noise to bring the parser closer still to the original look of the grammar.

It can be improved further, however: in other languages, for instance, the constructor can easily be overloaded so that the name does not require any prefix attached; and, in Haskell, the mechanical nature of the position tracking variant can be leveraged by Template Haskell to automatically generate the smart constructors for a datatype. Pattern synonyms [25] can be used as an alternative to smart constructors to construct compound structures, however the result must be pattern matchable.

5 Improving Errors: Anticipating Mistakes

An important part of a parser (up till now overlooked by this paper) is generating meaningful and *helpful* error messages. Writing good error messages is incredibly subjective as an exercise and so the focus here is not so much on what makes good error messages but instead on providing tools that the parser writer can keep in mind when considering their errors. Many libraries have a few combinators in common for errors that will be leveraged here:

```
(<?>)          :: Parser a → String → Parser a   --infix 0
unexpected     :: String → Parser a
fail           :: String → Parser a
lookAhead      :: Parser a → Parser a
notFollowedBy :: Parser a → Parser ()
```

The <?> combinator, pronounced "label" assigns a name to a parser to identify it in an error message. As an example:

```
digit = oneOf ['0'..'9'] <?> "digit"
```

As an aside, adding whitespace is hardly ever the solution to a syntax error (with the exception of indentation-sensitive grammars), so ideally it should be hidden from any syntax errors to reduce noise. This can often be accomplished by using (<?>"") to hide the label of the whitespace.

The unexpected and fail combinators both fail immediately, as indicated by their ability to seemingly produce a value of any type a out of thin air if they were to succeed. Normally, unexpected changes the part of the error message representing the problematic token, and fail adds a bespoke message to the error[5]. The specific combinators required for each technique may differ depending on the library.

The lookAhead and notFollowedBy combinators are positive and negative look-ahead respectively. If lookAhead succeeds, it does so without consuming any input, and if notFollowedBy's argument fails, then the combinator succeeds, and vice-versa; notFollowedBy never consumes input.

Conditionals. To provide a basis for the upcoming discussion, the grammar is extended one final time with conditional statements and basic comparisons:

⟨*stmts*⟩ ::= ⟨*stmt*⟩ ';' ⟨*stmts*⟩ | ⟨*stmt*⟩
⟨*stmt*⟩ ::= ⟨*asgn*⟩ | ⟨*ifStmt*⟩ | 'skip'
⟨*ifStmt*⟩ ::= 'if' ⟨*comp*⟩ '{' ⟨*stmt*⟩ '}' 'else' '{' ⟨*stmt*⟩ '}'
⟨*comp*⟩ ::= ⟨*expr*⟩ '<' ⟨*expr*⟩

Since there is more than one type of statement now, the sequence in ⟨*stmt*⟩ has been split out into a ⟨*stmts*⟩ rule instead. This change, along with the new components, must be incorporated into the AST:

```
data Stmts = Seq Stmt Stmts | OfStmt Stmt
data Stmt  = Asgn String Expr (Int, Int)
           | If Comp Stmt Stmt | Skip
data Comp = Less Expr Expr
```

It is perhaps a little overkill to create a new If datatype when Stmt will do, but, other than that, the datatype aligns perfectly with the grammar. The parser makes use of the same techniques used so far in the paper[6]:

[5] It may or may not also remove other information from the error.
[6] Assume that the additional keywords have been added to the lexer and new smart constructors have been created.

```
stmts  = infixr1 OfStmt stmt (mkSeq ‹* ";")
stmt   = asgn ‹|› ifStmt ‹|› mkSkip ‹* "skip"
ifStmt = mkIf ("if" *› comp) ("{" *› stmt ‹* "}")
              ("else" *›      ("{" *› stmt ‹* "}"))
comp   = mkLess expr ("<" *› expr)
```

Happily, this new syntactic extension offers some non-conventional syntax ripe for exploration. One example is that the body of an `if` statement cannot be empty:

```
parse (fully stmts)
  "if 0 < 1 {} else { x := 10 }"

(1, 11): unexpected "}"
  expected identifier, if, skip
  > if 0 < 1 {} else { x := 10 }
               ^
```

This error could be improved by marking it using ‹?› to give the three alternatives the name "statement". This is fine for experts, but it would be nice to explain what that means for those not versed in such terminology. This can be achieved by adding a fail combinator:

```
stmt = (asgn ‹|› ifStmt ‹|› mkSkip ‹* "skip"
    ‹?› "statement")
    ‹|› fail "statements consist of skips, [..]"
```

This produces a more friendly error:

```
(1, 11): unexpected "}"
  expected statement
  statements consist of skips, [...]
  > if 0 < 1 {} else { x := 10 }
               ^
```

As another example, unlike many mainstream languages, `if` statements not only require an `else` branch, but a trailing semicolon too:

```
parse (fully stmts)
  "if 0 < 1 { skip } else { skip }\nskip"

(2, 1): unexpected "s"
  expected ";", end of file
  > skip
    ^
```

Here, the user has fallen afoul of this restriction, assuming that a semicolon is not needed after braces. Unfortunately, the error message gives no indication that this is the cause of the issue, other than suggesting adding a semicolon.

> ***Problem 4: Helpful Errors.*** The parser may wish to provide guidance about avoiding common mistakes.

This example will be revisited later. Instead, suppose that the user now realises a semicolon is necessary, and now incorrectly assumes that all braces require a semicolon after:

```
(1, 18): unexpected ";"
  expected else
  > if 0 < 1 { skip }; else { skip }
                       ^
```

Again, it is clear that writing an `else` instead of the semicolon is the way to fix the problem, but fails to explain to the user *why* in a more useful way. The user has fallen afoul of the choice to make semicolons an operator instead of a delimiter, and again, there is no indication in the error message that would help the user understand *what* the rules are and *why* they have gotten them wrong.

A first attempt may use the same strategy used with statements, by adding a bespoke fail firing if `else` is not parsed:

```
elseClause = "else" ‹|› fail "semicolons are not [..]"
```

Whilst this seems fine at first glance (and works for the "true-positive" input above), it generates nonsensical messages in the presence of "false-positive" instances:

```
(1, 18): unexpected end of input
  expected else
  semicolons are not allowed between if and else
  > if 0 < 1 { skip }
                     ^
```

> ***Anti-pattern 4: Unconditional Errors.*** Addressing a common issue with a fixed error message can produce misleading errors.

5.1 Using Positive Lookahead

The problem with the naïve approach using solely fail on its own is that it has no awareness of the surrounding input: clearly, reporting that semicolons are illegal when found between `if` and `else` is only valid when there actually *is* a semicolon between them. The previous attempt failed to respect that idea, instead always emitting the message.

> ***Pattern 4a: Verified Errors.*** Use lookAhead to verify that contextual obligations are met before raising an error.

In contrast, lookAhead can be used to inspect the next part of the input to determine whether or not the conditions are met to make the error make sense. In the previous scenario, it is important to ensure that a semicolon has been written before referencing them, so the fail will be guarded. To help ensure that the added *error widgets* are not seen as part of the grammar, prefix their name with an underscore.

```
elseClause = "else" ‹|› _semi
_semi      = lookAhead (";" *› "else"
           *› fail "semicolons are not [..]" ‹?› "")
```

By predicating the fail behind the parsing of a semicolon and an `else` the message only appears when appropriate. The widget should be given the "hidden" error label.

5.2 Using Negative Lookahead

While the *Verified Errors* pattern is useful, it is less appropriate when the context of the error depends on non-local information in the grammar or when there are multiple places it can occur. As an example, informing the user that assigning comparisons to variables is illegal cannot be performed with lookAhead, since the valid continuations of assign: end of input, semicolons, or closing braces are valid elsewhere. If a lookAhead was placed at each of these places, then bad input like `"skip <"` may be reported as a bad assignment and, even if worked properly, it would duplicate the logic around the parser. Placing a lookAhead in the assignment, on the other hand, would only work with lookAhead comp… ‹|› expr, which always parses valid input twice, so it is not ideal.

> ***Pattern 4b: Preventative Errors.*** Use notFollowedBy to rule out illegal input or else raise an error.

A reasonable substitute is to use notFollowedBy to ensure that the right-hand side of an assignment is an expression that does not form part of an otherwise valid comparison. As with the *Verified Errors* pattern, the error widget can be distinguished by prepending its name with _no.

```
_noComp =
      notFollowedBy ("<" *> expr)
  <|> unexpected "<"
  <|> fail "\"<\" cannot be used in assignments"
  <?> "end of assignment"
asgn = mkAsgn (ident <* ":=") expr <* _noComp
```

This widget ensures that only when a < appears after an assignment with another expression will the user be informed that comparisons are illegal in assignments.

This technique was applied in Section 3.2 to prevent keywords from being followed by a letter. It can also be used after a closing brace to guard against another statement being written before a semicolon, resolving the very first example. Finally, it can also report that non-associative operators cannot be chained in the precedence combinator.

Discussion. These final two patterns are more situational than the rest presented in this paper but no less useful. However, the formulation using lookAhead and notFollowedBy is not perfect, and depends on how a library arbitrates between different messages. But they are a good heuristic for generating more bespoke, and applicable, errors.

6 Related Work

Object-oriented design patterns [9] are ubiquitous in the OOP world, being widely applied and embraced. There are instances of their use for the design and creation of parsers themselves: Nguyen et al. [22] uses several of the classic OOP design patterns to create easily extensible LL(1) parsers, where the elements of the grammar are decoupled to isolate the scope of any changes; Schreiner and Heliotis [26] leverage many of the same patterns in the internal design of a parser generator tool called `oops3`.

The ANTLR parser generator tool [24] uses the *Visitor* pattern to separate the building of a semantic action from parsing by exposing a visitor to process a generic parse tree; this contrasts with our *Lifted Constructors* pattern to decouple the concrete construction of ASTs from the parser which generates them. The difference in approach is a form of deforestation where the parser itself acts as a fold over the parse tree. ANTLR also uses the *Observer* pattern in the form of listeners to allow bespoke errors to be generated depending on the surrounding context, contrasting with our parser-driven *Verified Errors* and *Preventative Errors* patterns.

While there is an abundance of parser combinator tutorials [7, 15, 27, 28], they focus on how to design the libraries themselves, and do not offer any substantial discussion on *reusable patterns* for writing parsers. This is surprising since design patterns find such popularity in other disciplines. Kövesdán et al. [18] do document specific design patterns in the parsing space, but these are directed at an architectural level, describing the differences and applications of techniques such as hand-rolled parsers, parser generators, and so on. In this context, they would describe parser combinators as a pattern in themselves, and not describe their underlying nuances. As far as we are aware, our patterns have not been presented in the true design pattern style before, instead existing in folklore.

Different parsing algorithms such as Earley's algorithm [6], LALR, and LR can all handle left-recursion without any changes to the grammar or the parser [1]. There are also examples of libraries that use memoisation to allow left-recursion [8, 16, 17]. Danielsson [2] presents a parser combinator library using guarded co-induction to ensure that left-recursive grammars are still productive. Devriese and Piessens [5] present a parser combinator library that uses the left-corner transform [21] to automatically remove left-recursion. In all of these cases, the *Chains* pattern is not needed. However, variants of the *Precedence Tables* pattern still appear in many parser generator tools, like Happy [11].

7 Conclusion

The final state of the grammar is shown in Figure 3, and the final state of the parser (with the AST, lexer, and smart constructors omitted) is shown in Figure 4, along with the definitions of the error widgets that are used.

The effect of the *Precedence Table* pattern is that the precedence and fixities of the expression portion of the grammar are clearly represented in the expr rule in a way that leverages strong types to ensure the correctness of the table's fixities and ordering. While this does make the parser diverge from the grammar, it provides an easy way to establish information about each operator. The *Heterogeneous Chains*

⟨*stmts*⟩ ::= ⟨*stmt*⟩ ';' ⟨*stmts*⟩ | ⟨*stmt*⟩
⟨*stmt*⟩ ::= ⟨*asgn*⟩ | ⟨*ifStmt*⟩ | 'skip'
⟨*asgn*⟩ ::= ⟨*ident*⟩ ':=' ⟨*expr*⟩
⟨*ifStmt*⟩ ::= 'if' ⟨*comp*⟩ '{' ⟨*stmt*⟩ '}'
'else' '{' ⟨*stmt*⟩ '}'
⟨*comp*⟩ ::= ⟨*expr*⟩ '<' ⟨*expr*⟩
⟨*expr*⟩ ::= ⟨*expr*⟩ '+' ⟨*term*⟩ | ⟨*expr*⟩ '-' ⟨*term*⟩ | ⟨*term*⟩
⟨*term*⟩ ::= ⟨*term*⟩ '*' ⟨*negate*⟩ | ⟨*negate*⟩
⟨*negate*⟩ ::= 'negate' ⟨*negate*⟩ | ⟨*atom*⟩
⟨*atom*⟩ ::= '(' ⟨*expr*⟩ ')'
| ⟨*number*⟩ | ⟨*ident*⟩

Figure 3. Final Grammar

pattern has been used to handle the separation of statements using semicolons: here, the right-associativity of sequencing is enforced using the stronger types offered by infixr1. If the grammar did not specify the associativity for us, however, the *Homogeneous Chains* pattern could have been used to provide a more flexible implementation.

Using the *Overloaded Strings* pattern, the parser is void of any references to tokens or whitespace. This allows the parser to adopt a form closer to that of the grammar by using string literals. Behind the scenes, the *Whitespace*, *Tokenizing*, and *Keyword Combinators* patterns are used to cleanly manage lexing and consume whitespace consistently.

The *Lifted*, and *Deferred Constructors* patterns have been employed to abstract the creation of the AST from the parser, in the process abstracting away many sequencing combinators. While parts of the AST this parser produces may require position information, that is not evident here and can be easily managed separately.

The *Verified* and *Preventative Errors* patterns have been employed in the form of _semi and _noComp to provide some bespoke errors to the user of the parser. Whilst they distract a little from the overall parser, they can be kept separate and distinguished with their naming convention.

In all, the use of all of these patterns has yielded a clean and maintainable parser that can be easily extended as the language grows, and we have no doubt there are plenty more patterns waiting to be documented!

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments on the draft of this paper. This work has been supported by EPSRC grant number EP/S028129/1 on "SCOPE: Scoped Contextual Operations and Effects".

```
stmts  = infixr1 OfStmt stmt (mkSeq <* ";")
stmt   = asgn <|> ifStmt <|> mkSkip <* "skip"
asgn   = mkAsgn (ident <* ":=") expr <* _noComp
ifStmt = mkIf ("if" *> comp) ("{" *> stmt <* "}")
              (("else" <|> _semi) *> "{" *> stmt <* "}")
comp   = mkLess expr ("<" *> expr)
expr   = precedence $
  sops InfixL [mkAdd <* "+", mkSub <* "-"] +<
  sops InfixL [mkMul <* "*"]               +<
  sops Prefix [mkNeg <* "negate"]          +<
  Atom atom
atom   = "(" *> mkParens expr <* ")"
       <|> mkNum number <|> mkVar ident

_semi = lookAhead (";" *> "else"
      *> fail "semicolons are not [...]" <?> "")
_noComp =
      notFollowedBy ("<" *> expr)
  <|> unexpected "<"
  <|> fail "\"<\" cannot be used in assignments"
  <?> "end of assignment"
```

Figure 4. Final Parser

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/doi/10.5555/1177220

[2] Nils Anders Danielsson. 2010. Total Parser Combinators. *SIGPLAN Not.* 45, 9 (Sept. 2010), 285–296. https://doi.org/10.1145/1932681.1863585

[3] Nils Anders Danielsson and Ulf Norell. 2011. Parsing Mixfix Operators. In *Implementation and Application of Functional Languages*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–99. https://doi.org/10.1007/978-3-642-24452-0_5

[4] Tom DeMarco. 1979. *Structured Analysis and System Specification.* Prentice Hall PTR, USA. https://doi.org/doi/book/10.5555/1102012

[5] 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

[6] Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102. https://doi.org/10.1145/362007.362035

[7] Jeroen Fokker. 1995. Functional Parsers. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text.* Springer-Verlag, Berlin, Heidelberg, 1–23. http://dl.acm.org/citation.cfm?id=647698.734153

[8] 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/doi/10.5555/1621410.1621425

[9] 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/doi/book/10.5555/186897

[10] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

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

[12] Steve Hill. 1994. *Continuation Passing Combinators for Parsing Precedence Grammars*. Technical report. University of Kent, Computing Laboratory, University of Kent, Canterbury, UK. https://kar.kent.ac.uk/21168/

[13] Steve Hill. 1996. Combinators for parsing expressions. *Journal of Functional Programming* 6, 3 (1996), 445–464. https://doi.org/10.1017/S0956796800001799

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

[15] Graham Hutton and Erik Meijer. 1996. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham. https://doi.org/viewdoc/summary?doi=10.1.1.54.1678

[16] 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

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

[18] Gábor Kövesdán, Márk Asztalos, and László Lengyel. 2014. Architectural design patterns for language parsers. *Acta Polytechnica Hungarica* 11, 5 (2014), 39–57. https://doi.org/10.12700/aph.11.05.2014.05.3

[19] Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. Technical Report. Microsoft. https://doi.org/viewdoc/summary?doi=10.1.1.19.5187

[20] Robert C Martin. 2000. Design principles and design patterns. *Object Mentor* 1, 34 (2000), 597.

[21] 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/doi/10.5555/974305.974338

[22] 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

[23] Meilir Page-Jones. 1988. *The Practical Guide to Structured Systems Design: 2nd Edition*. Yourdon Press, USA. https://doi.org/doi/book/10.5555/48039

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

[25] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. *SIGPLAN Not.* 51, 12 (Sept. 2016), 80–91. https://doi.org/10.1145/3241625.2976013

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

[27] Doaitse Swierstra and Luc Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*. Springer-Verlag, London, UK, 184–207. http://dl.acm.org/citation.cfm?id=647699.734159

[28] 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

[29] Philip Wadler. 1985. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–128. https://doi.org/10.1007/3-540-15975-4_33

[30] Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (Jan. 1988), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

[31] 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

[32] Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged Selective Parser Combinators. *Proc. ACM Program. Lang.* 4, ICFP, Article 120 (Aug. 2020), 30 pages. https://doi.org/10.1145/3409002