# Frequently Asked Questions

### What is the deal with "lazy" vs "strict" positions?

In `parsley-4.x.y` and up, combinators are no longer totally lazy and adopt the idea of lazy and strict arguments. In general, all combinators have strict receivers.

- A strict argument/receiver indicates that the combinator intends for that argument to be parsed "immediately" without a possibility of consuming input before-hand
- A lazy argument/receiver indicates that the combinator acknowledges that input may be consumed by another argument (or the combinator itself) before the lazy argument will be executed.

Why is this important? Recursive parsers are implemented in Parsley with "knot-tying" using `lazy val`. When a parser appears recursively in its own definition, it needs to be kept lazy otherwise it will be evaluated during its own definition, leading to an infinite loop. Lazy positions in combinators indicate probably safe places for recursion to occur. On the other hand, the absence of input-consumption before a strict position *probably* indicates that recursion in that position will result in a *left-recursive* parser: the necessary condition for left-recursion is that a recursive call is made without having consumed input since the last recursive call. Technically, this is known as *unguarded* (by input consumption) recursion, with correct parsers making use of *guarded recursion*. This is not bullet-proof:

- Strict positions sometimes occur in combinators where they *should* be lazy but cannot be due to language limitations: `choice`, the `zipped` syntax, etc all fall victim to this.
- Lazy positions suggest the previously executed parsers *could* have consumed input, but it is not guaranteed that they actually *do*. As an example, `LazyParsley.unary_~` can be used to make a parser lazy in an otherwise strict position, but doesn't itself consume input, so care must still be taken to ensure there is input consumed by previous arguments!

Note that a strict parser in a lazy position is still lazy!

For an example, `lazy val q = (p <~ ':', q).zipped(f)` would be problematic because the recursive call is used naked in a totally strict combinator (`zipped`). There are two ways to fix this problem:

1. Reshuffle the surrounding context to move it into a lazy position: `lazy val q = (p, ':' ~> q).zipped(f)` would be fine, since the right-hand side of ~> is a lazy position guarded by the input consumption of `':'`.
2. Use `LazyParsley.unary_~` to introduce laziness:

```
import parsley.Parsley.LazyParsley
lazy val q = (p <~ ':', ~q).zipped(f)
```

Here, the `unary_~` is used to just make `q` lazy: this is simply defined as `unit ~> q`, which places `q` into a *unguarded* lazy position, since `unit` does not consume input. The guardedness in this case comes from `p <~ ':'`, which is guaranteed to consume a `:` on success.

## Frequently Encountered Problems

### My parser seems to infinite loop and doesn't run, even with the `debug` combinator

This sounds like you've run into the above issue with *left-recursion*. This means that a parser appears as the first possible route of exploration from itself:

```
lazy val bad = bad ~> ...
lazy val badExpr = Add.lift(badExpr, '+' ~> badExpr) | number
lazy val goodExpr = atomic((number <~ '+', goodExpr).zipped(Add)) | number
```

The first two parsers are both examples of left-recursive parsers, which are not allowed in Parsley. The third is caused by the above problem of a genuine recursive call in a strict position, and can be fixed with `unary_~` or by rearranging the `'+'` parser as above:

```
lazy val goodExpr = atomic((number, '+' ~> goodExpr).zipped(Add)) | number
lazy val goodExpr = atomic((number <~ '+', ~goodExpr).zipped(Add)) | number
```

### My parser seems to infinite loop and the `debug` combinator shows it spinning forever

As above you've probably made a *left-recursive* parser, but this time hidden the bad recursion inside an *unguarded* lazy position:

```
import parsley.Parsley.LazyParsley
lazy val badExpr = Add.lift(~badExpr, '+' ~> badExpr) | number
```

Perhaps you tried to fix the above bad parser using `unary_~` even though its an unsafe recursive position?

### My parser throws a `BadLazinessException`, what gives!

This error is thrown by the Parsley compiler when Scala would otherwise have thrown a `NullPointerException` during what's known as "let-finding". This is the very first phase of the pipeline, and, since Parsley does not use `null` *anywhere* in the front-end, its a symptom of a parser having been demanded before its actually been defined. So what does mean for you, the user?

Well, unfortunately, Scala doesn't give us any indication about which parser is at fault (and the Java 14 Helpful NPEs don't help much either). But here are the possible causes:

1. A parser references one that has been defined below it and it hasn't been marked as `lazy val`
2. A combinator doesn't have the right laziness characteristics (conventionally, Parsley defines all its combinators arguments using by-name parameters and `lazy val`s for those that appear multiple times in the body)
3. Be careful: builder combinators to abstract position tracking (as per the Parsley guide) *also* will require the same care as (2)

Now, the solution to (1) is simple, just either add a `lazy val`, or reorder the grammar clauses so that there is no forward referencing. Similarly, (2) is simple to fix by adding in the right laziness to the parameter. However, because Parsley is careful to make as much as possible lazy (be careful of `parsley.syntax.zipped.{Zipped2, Zipped3}`, however, neither of them are lazy!) you may find that you can define an entire parser without ever running into this problem, even if nothing is marked `lazy`: lucky you! My advice is to try and keep things ordered nicely, or mark everything as lazy; of course, laziness will have a slight runtime penalty, so its worth seeing how much of the laziness you can eliminate by just reordering.

You may be helped out by Scala here, either by an error about "crossed-initialisation" or a warning about "Reference to uninitialized value". If you see either of these, check the order and laziness of the parsers concerned!

# Parser Combinator Cheatsheet

Until you're more comfortable using parser combinators and have a sense for how to piece things together, the sheer choice can be daunting. This page is designed to help bridge this gap by bringing attention to some of the more common and useful combinators, as well as a few common idioms for getting stuff done.

## Quick Documentation

This section is designed to give a quick reference for some of the most common combinators, their types as well as their use.

### Basic Combinators

These are the basic combinators for combining smaller parsers into bigger parsers, or adjusting their results. These combinators are sometimes lazy in their arguments, which is denoted here by the regular *by-name* (`=>A`) syntax. If an argument is strict, it means that it will be parsed immediately on entry to the combinator before any input can be consumed.

| Combinator | Type | Use | Pronounciation |
|---|---|---|---|
| `pure(_)` | `A => Parsley[A]` | return a value of type `A` without parsing anything. | "pure" |
| `_ *> __ ~> _` | `(Parsley[A], =>Parsley[B]) => Parsley[B]` | sequence two parsers, returning the result of the **second**. | "then" |
| `_ <* __ <~ _` | `(Parsley[A], =>Parsley[B]) => Parsley[A]` | sequence two parsers, returning the result of the **first**. | "then discard" |
| `_.map(_)` | `(Parsley[A], A => B) => Parsley[B]` | use a function to change the result of a parser. | "map" |
| `_ <#> _` | `(A => B, Parsley[A]) => Parsley[B]` | use a function to change the result of a parser. (*Requires* `import parsley.extension.Ha` | "map" |

| Combinator | Type | Use | Pronounciation |
|---|---|---|---|
| `_ #> __.as(_)` | `( Parsley[A], B) => Parsley[B]` | replace the result of a parser with a fixed value. | "as" |
| `liftN(_, .., _)` | `( (A1, A2, .., An) => B, Parsley[A1], =>Parsley[A2], .., =>Parsley[An]) => Parsley[B]` | use a function to combine the results of *n* parsers, sequencing them all together. | "lift n" |
| `_ <\|> __ \| _` | `( Parsley[A], =>Parsley[A]) => Parsley[A]` | try one parser, and if it fails *without consuming input* try the second | "or" |
| `atomic(_)` | `Parsley[A] => Parsley[A]` | perform a parser, but roll-back any consumed input if it fails, use in conjunction with <\|>. | "atomic" |
| `lookAhead(_)` | `Parsley[A] => Parsley[A]` | execute a parser, and roll-back any consumed input if it *succeeded*. | "look-ahead" |
| `notFollowedBy(_)` | `Parsley[A] => Parsley[Unit]` | execute a parser, never consuming input: succeed only if the parser fails. | "not followed by" |
| `empty` | `Parsley[Nothing]` | fails when executed. | "empty" |

## Character Combinators

These combinators, found in `parsley.character` are useful for dealing with *actually* consuming input.

| Combinator | Type | Use |
|---|---|---|
| `char(_)` | `Char => Parsley[Char]` | Reading a single specific character. That character is returned. |

| Combinator | Type | Use |
|---|---|---|
| `string(_)` | `String => Parsley[String]` | Reading a single specific string. That string is returned. |
| `satisfy(_)` | `(Char => Boolean) => Parsley[Char]` | Read any single character for which the provided function returns `true`. The character returned is the one read. |
| `oneOf(_*)` | `Char* => Parsley[Char]` | Read any *one* of the provided characters (which are varargs). The character returned is the one read. |
| `noneOf(_*)` | `Char* => Parsley[Char]` | Read any single character that is *not* one of the provided characters. The character returned is the one read. |

## Lifty Combinators

These combinators can all be implemented in terms of `lift2` (see `liftN` above), but are considered useful enough to have their own syntax and name. These combinators are lazy in their arguments (but not receivers),denoted here by the regular *by-name* (`=>A`) syntax. As such, the receiver is parsed first and may consume input, which means the argument may contain a recursive position (and must therefore be lazy).

| Combinator | Type | Use | Pronounciation |
|---|---|---|---|
| `_ <~> _` | `(Parsley[A], =>Parsley[B]) => Parsley[(A, B)]` | combine the results using `(_, _)` | "zip" |
| `_ <*> _` | `(Parsley[A => B], =>Parsley[A]) => Parsley[B]` | combine the results using `(f, x) => f(x)`. | "ap" |
| `_ <**> _` | `(Parsley[A], =>Parsley[A => B]) => Parsley[B]` | combine the results using `(x, f) => f(x)`. | "reverse ap" |

| Combinator | Type | Use | Pronounciation |
|---|---|---|---|
| `_ <::> _` | `(Parsley[A], =>Parsley[List[A]]) => Parsley[List[A]]` | combine the results using `_ :: _`. | "cons" |

## Composite Combinators

These combinators tackle more common complex tasks. In particular `many` and `some` are **very** important. They are all found in `parsley.combinator`. These combinators are sometimes lazy in their arguments, which is denoted here by the regular *by-name* (`=>A`) syntax. Care should be taken with the combinators with variadic arguments, as they are totally strict, even in normally lazy positions: `LazyParsley.unary_~` can be used to restore laziness in these positions.

| Combinator | Type | Use |
|---|---|---|
| `many(_)` | `Parsley[A] => Parsley[List[A]]` | run one parser many times until it fails, collecting all the results in a list. |
| `some(_)` | `Parsley[A] => Parsley[List[A]]` | as above, but the parser must succeed at least once. |
| `eof` | `Parsley[Unit]` | check if there is any remaining input: it will succeed if there is none. |
| `choice(_*)` | `Parsley[A]* => Parsley[A]` | try each of the given parsers in turn until one succeeds: uses `<\|>`. |
| `option(_)` | `Parsley[A] => Parsley[Option[A]]` | try a parser, if it succeeds wrap the result in `Some`, and if it fails *without consuming input* return `None`. |
| `optional(_)` | `Parsley[A] => Parsley[Unit]` | optionally parse something (but if it fails, it must not consume input). |

| Combinator | Type | Use |
|---|---|---|
| `sepBy1(_, _)` | `(Parsley[A], =>Parsley[_]) => Parsley[List[A]]` | parse one thing separated by another, collecting all the results. Something like comma-separated arguments in a function call. |
| `endBy1(_, _)` | `(Parsley[A], =>Parsley[_]) => Parsley[List[A]]` | same as above, but the sequence must be ended by the separator again. Something like semi-colon separated statements in C-like languages. |
| `sepEndBy1(_, _)` | `(Parsley[A], =>Parsley[_]) => Parsley[List[A]]` | same as above, but the terminal separator is optional. Something like semi-colon separated statements in Scala. |

## Building Values and ASTs

This section covers the common ways you might build a result value or Abstract Syntax Tree (AST) with your parsers.

The most primitive combinators for reading input all have a tendency to return the thing they parsed, be it a single character or a string. For the most part, this is not the useful output you'd like your parser to have.

### Transforming a single value with `map`

The quickest way to change the result of a parser is by using `.map` or the `#>` combinator (see the above quick documentation). This is really useful for changing the result of a *single* parser, but provides no way of combining multiple.

```
import parsley.Parsley, Parsley.some
import parsley.character.digit

case class Num(n: Int)

// A preferred method is to use `digit.foldLeft1` to avoid creating a List.
val digits: Parsley[List[Char]] = some(digit)
// `map` here is using a function of type `List[Char] => Int`
val int: Parsley[Int] = digits.map(_.mkString.toInt) // equivalently
 `digits.map(_.mkString).map(_.toInt)
// `map` here is being used to wrap the `Int` in the `Num` class
val num: Parsley[Num] = int.map(Num)
```

But when you need to combine the results of two parsers more options open up.

### Combining multiple results with `lift`, `<::>`, and friends

Let's suppose we want to rule out leading zeros in the above parser. We'll need to read one non-zero digit before we read zero or more digits. In this case, we want the first digit to be added to the list of remaining digits. This task is quite common, so the `<::>` combinator is designed specially for it:

```
import parsley.Parsley, Parsley.many
import parsley.character.{digit, oneOf, char}

case class Num(n: Int)

val nonzero = oneOf('1' to '9')

// <::> adds the leading non-zero char onto the other digits
val digits: Parsley[List[Char]] = nonzero <::> many(digit)
// Using #> here to handle the plain ol' zero case
val int: Parsley[Int] = char('0') #> 0 | digits.map(_.mkString.toInt)
val num: Parsley[Num] = int.map(Num)
```

But more generally, we could reach for the `lift` functions:

```
import parsley.Parsley, Parsley.many
import parsley.character.{digit, oneOf, char}
import parsley.lift.lift2

case class Num(n: Int)

val nonzero = oneOf('1' to '9')

val digits: Parsley[List[Char]] = lift2[Char, List[Char], List[Char]](_
 :: _, nonzero, many(digit))
// Using #> here to handle the plain ol' zero case
val int: Parsley[Int] = char('0') #> 0 | digits.map(_.mkString.toInt)
val num: Parsley[Num] = int.map(Num)
```

Sadly, to do this, it's sometimes necessary to specify all the types, in particular for anonymous functions that can have many possible type-instantiations, like `_ :: _`. The reason is that Scala doesn't infer the types of arguments, only return values, so on its own `_ :: _` has no known type. As such, the fix is to let other type-instantations help give the argument types (as above) or to specify the types in the function manually:

```
lift2((c: Char, cs: List[Char]) => c :: cs, nonzero, many(digit))
```

Notice that this didn't seem to be a problem with `map`. This is because the function is type-checked after the receiver of the method: it gets given the right argument type straight away. Parsley has a syntax for leveraging this property:

```
import parsley.syntax.zipped.Zipped2

(nonzero, many(digit)).zipped(_ :: _)
```

The `zipped` syntax, unlike the `liftN` combinators or `lift` syntax, is not lazy in *any* of its arguments, so care may be needed to use `LazyParsley.unary_~` to restore laziness to those arguments that need it.

Use this form of lifting when type-inference fails you. Otherwise, for clarity, use a regular `liftN`, or the syntactic sugar for it:

```
import parsley.syntax.lift.{Lift2, Lift1}

val charCons = (c: Char, cs: List[Char]) => c :: cs

charCons.lift(nonzero, many(digit))
Num.lift(int)
```

The `lift` functions work all the way up to 22 arguments (which is the Scala 2 limit on function arguments). The same goes for the `zipped` syntax and `lift` syntax. Don't forget about `<::>` as well as its friends `<~>`, `<*>`, and `<**>`! They all provide a concise way of combining things in (common) special cases.

**A note for Haskellers**

In Scala, curried application is not as favoured as it is in Haskell for performance reasons. The classic `f <$> p <*> .. <*> z` pattern that is common in Haskell is unfavourable compared to the scala `liftN(f, p, .., z)`. For the latter, `f` is uncurried, which is the norm, and so it is almost always more efficient. Both `<*>` and `<**>` should be, therefore, used sparingly in idiomatic `parsley` code instead of liberally like in Haskell.

However, it goes without saying that `lift2[A => B, A, B]((f, x) => f(x), pf, px)` is no more efficient than `pf <*> px` so the latter is favoured for that use case!

# Understanding the API

## Main Classes and Packages

In `parsley`, everything resides within the **parsley** package, and the major entry point is `parsley.Parsley`. There are a few modules of note:

- `parsley.Parsley`: contains some of the basic and primitive combinators (at least those that aren't methods on parsers).
- `parsley.combinator`: contains handy combinators, this should be your first port of call when you want to do something but are not sure a combinator exists for it. At the very least, the `eof` combinator is very common.
- `parsley.character`: contains a variety of combinators which deal with characters, key ones include `char`, `satisfy` and `string`.
- `parsley.syntax`: contains the very useful implicit conversion combinators. In particular, importing `charLift` and `stringLift` allows you write character and string literals as if they were parsers themselves. There are also implicit classes here which extend functions of any arity with a corresponding `.lift` method, instead of using the `liftN` functions.
- `parsley.expr`: contains the machinery needed to generate expression parsers for you based, at its simplest, on a table of operators in order of precedence. This is *well* worth a look (this is covered in detail in **Building Expression Parsers**.
- `parsley.token`: contains a bunch of functionality for performing common lexing tasks, which is *very* configurable. These parsers may also be optimised for performance.

### Using `parsley.token` for Lexing

Unlike Haskell libraries like `megaparsec` and `parsec`, `parsley` does not tie the lexing functionality to the Haskell Report, instead supporting a superset of the functionality. The functionality is provided to the user by **Lexer**, and this must be provided an value of type **LexicalDesc**, which provides all the configuration necessary to describe the language and make the parsers.

# Parsley (`parsley.Parsley`)

All parsers have type `Parsley`, which has many methods (combinators) for composing parsers together. The companion object also contains some primitive combinators and parsers.

> *The Scaladoc for this page can be found at* `parsley.Parsley` ***(class)*** *and* `parsley.Parsley` ***(object)***.

## Class `Parsley`

The `Parsley` class is the value class responsible for representing parsers. It has methods largely grouped into the following categories:

- Methods to execute a parser, like `parse`.
- Methods to alter the result of a parser, like `map`, `as`, `void`, and `span`.
- Methods to sequence multiple parsers, like `flatMap`, `zip`, <~, and ~>.
- Methods to compose parsers in alternation, like `orElse`, |, and <+>.
- Methods to filter the results of parsers, like `filter`, `collect`, and `mapFilter`.
- Methods to repeat a parser and collapse the results, like `foldLeft`, `reduceRight`, and so on.
- Special methods to help direct `parsley`, like `impure`, `overflows`, and `force`.

### Running Parsers

The `parse` method runs a parser with some given input. The type signature has an additional `Err` parameter, with an implicit requirement for `ErrorBuilder[Err]`. This is discussed in `parsley.errors.ErrorBuilder`, but for the purposes of basic use, you can assume that Scala will automatically infer this type to be `String`, for which an `ErrorBuilder[String]` does implicitly exist.

By importing `parsley.io._`, another `parseFromFile` method is added to `Parsley`, which works similarly, but loads the input from a file first. In future versions of `parsley`, this import will no longer be needed, and this will be an overloading of `parse`.

## Altering Results

The results of an individual parser can be altered using `map` or combinators derived from it, such as `as` or `void`. Note that the `#>` combinator is a symbolic alias for `as`, though as is now recommended.

> Note that, whilst `as` can be implemented using `map`, it is actually implemented with `~>` and `pure`, which allows it to be better optimised: if a result is not needed, `parsley` will ensure it is not even generated to begin with.

The `span` combinator is special: it allows for the result of a parser to be discarded and instead the input consumed in the process of parsing is returned instead. This can be used to avoid constructing strings again after parsing.

## Composing Parsers

Some combinators allow for multiple parsers to be composed together to form a new one. This comes in two forms: sequencing them one after another and combining their results in some way, or combining them in parallel as two independent choices.

### Sequencing

The `zip` combinator is the most permissive way of combining two parsers together whilst retaining both of their results; `p zip q` will parse `p`, then `q` and then return a pair of their results. The symbolic version of `zip` is called `<~>`. When both results are not needed, `~>` and `<~` point towards a result to keep, and discard the other.

Combinators like `<*>`, `<**>` and `<::>` perform a similar role to `zip`, but combine the two results depending on the specific subtypes of the arguments. Both `<*>` and `<**>` perform function application, and `<::>` adds the result of the first parser onto a list returned by the second. These are all more specific versions of the `parsley.lift._` combinators, but are often useful in practice.

The `flatMap` combinator is another way of sequencing two parsers, but where the second depends on the result of the first. In `parsley`, this operation is **very** expensive, and it (and its derived `flatten` combinator) should be avoided. One way of avoiding the `flatMap` combinator is to use features found in `parsley.state`, or to use `lift` combinators instead.

### Choice

When one of two parsers can be used at a specific point, the `|` combinator (also known as `orElse` or `<|>`) can be used to try one and then the other if the first failed. The result of the successful branch, if any, will be returned.

> ⚠
>
> A parser `p | q` will **only** try `q` if the parser `p` failed having **not** consumed any input. If it does consume input, then the overall parser will fail. The `atomic` combinator can be used to prevent input consumption on failure, however.
>
> When `atomic` is used in this manner, it may cause error messages to be less effective, or increase the runtime or complexity of the parser. Backtracking in this way should be avoided if at all possible by factoring the grammar or employing techniques like ***Disambiguator Bridges***.

## Filtering Results

When the results of the parser need to be verified or conditionally transformed, filtering combinators can be used. Largely, they will try and apply a function to the result of a parser: if the function returns `false` or is otherwise not defined, then the parser will fail, and otherwise, it may perform some transformation:

- `filter` will check if the result of a parser matches some predicate, failing otherwise. If successful, result is returned unchanged
- `collect` will attempt to apply a partial function to the result of the parser: if the function is defined on that input, it is applied and the new result returned; otherwise, the parser fails.
- `filterMap` is similar to `collect`, but works with a function that returns `Option[B]`: if the function returns `None`, the parser fails; otherwise the value inside the `Some` is returned.

Largely, these combinators are more useful in their more advanced formulations, those in `parsley.errors.combinator.ErrorMethods`, which produce detailed error messages about filtering failures.

## Reductions

There are a collection of methods that repeatedly perform a parser, and collapse the generated results into a single value - like regular folds. There are three classes of reductive combinators:

- `foldLeft`, `foldRight`: these combine up the results of repeated parsing of a parser starting with an initial value and combining each result with the running value. When the name of the combinator has a `1` at the end, it means the parser must succeed at least once; otherwise the parser need not succeed and the combinator will return the initial value (so long as no input was consumed in the process).

- `reduceLeft`, `reduceRight`: these combine one or more parses of a parser by a left or right reduction. As the parser will parse successfully at least once, there is no need for a default or initial value.
- `reduceLeftOption`, `reduceRightOption`: like `reduceLeft` or `reduceRight`, but returns `None` if the parser could not successfully parse once, and wraps the result of reduction in a `Some` otherwise.

### Special Methods

The underlying implementation of `parsley` is more akin to a compiler than a regular parser combinator library. Some methods exist to help direct the internal compiler:

- `force()`: this forces the compiler to optimise and compile the parser right now, as opposed to when the parser is first executed. This should be used on the top-level parser.
- `impure`: this tells the optimiser to not touch the parser, which may be necessary when the parser deals with mutable objects. This is because the optimiser assumes purity, and may factor out or even remove results entirely, which would otherwise change the results. This can be used across the parser.
- `overflows()`: this tells the compiler that the parser it is invoked on will stack overflow during compilation. As a result, the compiler will instead process the parser in a stack-safe trampoline. This should be used on the top-level parser.

## Object `Parsley`

The `Parsley` object contains a collection of primitive combinators that help control a parser or its results. By far the most important ones are `pure` and `atomic`:

- `pure` injects a value into the parsing world without having any other effect on the state of the parser.
- `atomic` ensures that a parser either consumes all its input and succeeds, or none of it and fails. This is important because the `|` combinator will not parse its second argument if the first failed having consumed input - `atomic` helps by ensuring no input was consumed on failure.

In addition to these, it also has `empty` and `empty(Int)`, which fail the parser immediately (but recoverably) and produces a given width of caret (and `0` in the case of `empty`); `lookAhead` and `notFollowedBy` for dealing with positive and negative lookahead; and `fresh`, which can be used like `pure` but evaluates its argument every time, which is useful in the presence of mutable values.

### Iterative Combinators

One of the main classes of combinator are the iterative combinators, which execute parsers multiple times until they cannot match any more; the results of these combinators vary. If the parser being repeated fails

having consumed input, iterative combinators will fail; if no input was consumed on failure, the iteration will stop.

The most commonly used of these are the `many` and `some` combinators, which return a list of the successful results:

```
import parsley.character.digit
import parsley.Parsley.{many, some}

many(digit.zip(digit)).parse("")
// res0: parsley.Result[String, List[(Char, Char)]] = Success(List())
many(digit.zip(digit)).parse("1234")
// res1: parsley.Result[String, List[(Char, Char)]] = Success(List((1,2),
 (3,4)))
many(digit.zip(digit)).parse("12345")
// res2: parsley.Result[String, List[(Char, Char)]] = Failure((line 1, column
 6):
//   unexpected end of input
//   expected digit
//   >12345
//         ^)

some(digit.zip(digit)).parse("")
// res3: parsley.Result[String, List[(Char, Char)]] = Failure((line 1, column
 1):
//   unexpected end of input
//   expected digit
//   >
//    ^)
some(digit.zip(digit)).parse("1234")
// res4: parsley.Result[String, List[(Char, Char)]] = Success(List((1,2),
 (3,4)))
some(digit.zip(digit)).parse("12345")
// res5: parsley.Result[String, List[(Char, Char)]] = Failure((line 1, column
 6):
//   unexpected end of input
//   expected digit
//   >12345
//         ^)
```

# Parsing Characters

The consumption of input is central to parsing. In `parsley`, the only way of consuming input is by consuming characters, as customised token streams are not supported. However, as `parsley` is based on Scala, which relies on 16-bit Basic-Multilingual Plane UTF16 characters, there are two modules for input consumption: `character`, for 16-bit `Char`, and `unicode`, handling 32-bit `Int`s.

*The Scaladoc for this page can be found at* `parsley.character` *and* `parsley.unicode`.

## Character Primitives

There are three key input consumption primitives in `parsley`: `satisfy`, which takes a predicate and parses any character for which that predicate returns true; `char`, which reads a specific character; and `string`, which reads a specific sequence of characters. Technically, `char` and `string` can be built in terms of `satisfy`, but would then lose their enhanced error message characteristics.

The `oneOf` and `noneOf` are based on `satisfy` but with better error messages, based on the kind of range of characters they are provided with. The `strings` and `trie` combinators can be used similarly to match a set of strings: they are likely to parse these more efficiently than a manual construction and may improve further in future.

## String Building

It is possible to efficiently construct a `String` using the `stringOfMany` and `stringOfSome` combinators. This can be used with either a character predicate (which turns out to be called `takeWhileP` in `megaparsec`), or can be given a parser that consumes a character instead. The construction of the underlying string is done efficiently with a `StringBuilder`. Note that, with the version that takes a parser, the characters incorporated into the string are the ones returned by the parser and not necessarily the ones the parser consumed.

## Pre-Built Character Parsers

Some pre-built parsers are bunched into `character` and `unicode`, which parse specific sets of letter, with good error messages. Each will document the specific set of characters they recognise.

# Additional Combinators (`parsley.combinator`)

While many combinators are implemented as methods directly on the `Parsley` type, some are left as functions within the `combinator` module. These mostly involve handling repeating parsers. Not all the combinators are discussed here.

> *The Scaladoc for this page can be found at* `parsley.combinator`.

## Iterative Combinators

One of the main classes of combinator are the iterative combinators, which execute parsers multiple times until they cannot match any more; the results of these combinators vary. If the parser being repeated fails having consumed input, iterative combinators will fail; if no input was consumed on failure, the iteration will stop.

The most commonly used of these are the `many` and `some` combinators (see **Object** `Parsley`), which return a list of the successful results. While `some` will parse one or more times, `manyN` generalises to work for any minimum required parses `n`. When the results are not needed, `skipMany`, `skipSome`, and `skipManyN` can be used instead. To determine how many times the parse was successful, the `count` and `count1` can be used instead.

The `manyTill` and `someTill` combinators can be used to parse iteratively until some other parse is successful: this can be used, for instance, to scan comments:

```
import parsley.character.{string, item, endOfLine}
import parsley.combinator.{manyTill}

val comment = string("//") ~> manyTill(item, endOfLine)
// comment: parsley.Parsley[List[Char]] = parsley.Parsley@1d4ca6a8
comment.parse("// this is a comment\n")
// res0: parsley.Result[String, List[Char]] = Success(List( , t, h, i, s,  , i,
 s,  , a,  , c, o, m, m, e, n, t))
```

### Separators

There are three variants of the iterative combinators that handle delimited parsing: reading something separated (or ended) by another delimiter:

- `sepBy1`: parses input like `x, y, z`

- `endBy1`: parses input like `x; y; z;`
- `sepEndBy1`: parses input like `x, y, z` *or* `x, y, z,`

## Optional Combinators

The `option`, `optional`, and `optionalAs` combinators can be used to optionally parse something. The `option` combinator will inject the result into an `Option`; the `optional` combinator just returns `Unit`; and the `optionalAs` combinator unconditionally returns any given value.

```
import parsley.character.digit
import parsley.combinator.option

option(digit.zip(digit)).parse("")
// res1: parsley.Result[String, Option[(Char, Char)]] = Success(None)
option(digit.zip(digit)).parse("0")
// res2: parsley.Result[String, Option[(Char, Char)]] = Failure((line 1, column
 2):
//   unexpected end of input
//   expected digit
//   >0
//     ^)
option(digit.zip(digit)).parse("09")
// res3: parsley.Result[String, Option[(Char, Char)]] = Success(Some((0,9)))
```

## Conditional Combinators

The conditional combinators are used for conditional execution of a parser. These include `ifP`, `guard`, `when`, and `whileP`. As may be expected, `ifP` models an *if-else-expression*, `whileP` models a *while-loop*, and `when` models an *if-statement*. They mostly find their utility when used in conjunction with **registers**.

# Generic Bridges (`parsley.generic`)

The *Parser Bridge* pattern is a technique for decoupling semantic actions from the parser itself. The `parsley.generic` module contains 23 classes that allow you to get started using the technique straight away if you wish.

> *The Scaladoc for this page can be found at* `parsley.generic`.

## What are *Parser Bridges*?

Without making use of *Parser Bridges*, results of parsers are usually combined by using `lift`, `map`, or `zipped`:

```
import parsley.syntax.zipped.Zipped2
case class Foo(x: Int, y: Int)
// with px, py of type Parsley[Int]
val p = (px, py).zipped(Foo(_, _))
// p: Parsley[Foo] = parsley.Parsley@270db29f
```

These work fine for the most part, however, there are couple of problems with this:

1. In Scala 3, `Foo(_, _)` actually needs to be written as `Foo.apply`, which introduces some (minor) noise; `zipped` itself is even contributing noise.
2. `Foo` itself is a simple constructor, if it gets more complex, readability rapidly decreases:
   - The result produced may require inspection of the data, including pattern matching (see **Normalising or Disambiguating Data**).
   - Additional information may need to be threaded in, like position information.
   - Data invariances may need to be enforced (see **Enforcing Invariances**).
3. Constructor application is on the right of the data, which people may find harder to read; this can be mitigated with `lift`, but that may run into type inference issues.

For some people (1) or (3) may not be an issue, or can be tolerated, but (2) can get out of hand quickly. For larger parsers, properly decoupling these issues can make a huge difference to the maintainability.

How do bridges help? In short, a bridge is an object that provides an `apply` method that takes parsers as arguments, as opposed to values. This means that they can be used directly in the parser with the logic kept elsewhere. While you can just define a bridge manually with an `apply` method (or even just as a function), it

is more ergonomic to *synthesise* a bridge in terms of a function that does not interact with `Parsley` values. If we assume that the companion object of `Foo` has been turned into such a bridge (definition below), the above example can be written as:

```
val q = Foo(px, py)
// q: Parsley[Foo] = parsley.Parsley@529cc777
```

In this version, the act of constructing the `Foo` value has been abstracted behind the bridge, `Foo`: this means that the underlying implementation can vary without changing the parser.

What are *Generic Bridges*?
Generic bridges are the templating mechanism that allow for the synthesis of an `apply` method that works on values of type `Parsley` from another that does not. While you can define your own bridge templates (see **the associated tutorial** for an explanation), `parsley` provides some basic ones to get you started.

## How to use

The `parsley.generic` module contains `ParserBridge1` through `ParserBridge22` as well as `ParserBridge0`; they all extend `ParserBridgeSingleton`, which provides some additional combinators.

### `ParserBridge1[-T1, +R]` through `ParserBridge22[-T1, .., -T22, +R]`

Each of these traits are designed to be implemented ideally by a companion object for a `case class`. For example, the `Foo` class above can have its companion object turned into a bridge by extending `ParserBridge2` (which is for two argument bridges):

```
import parsley.generic.ParserBridge2
object Foo extends ParserBridge2[Int, Int, Foo]
```

This defines `def apply(px: Parsley[Int], py: Parsley[Int]): Parsley[Foo]`, implementing it in terms of `def apply(x: Int, y: Int): Foo`, which is included as part of Scala's automatic `case class` implementation. By making use of a companion object, this is *all* the boilerplate required to start using the bridge. Of course, it's possible to define standalone bridges as well, so long as you provide an implementation of `apply`, as illustrated by this error:

```
object Add extends ParserBridge2[Int, Int, Int]
// error: object creation impossible.
// Missing implementation for member of trait ParserBridge2:
//   def apply(x1: Int, x2: Int): Int = ??? // implements `def apply(x1: T1,
 x2: T2): R`
//
// object Add extends ParserBridge2[Int, Int, Int]
//        ^^^
```

Implement that `apply` method and it's good to go! Of course, if the traits are mixed into a regular `class`, they can also be parametric:

```
class Cons[A] extends ParserBridge2[A, List[A], List[A]]
// error: class Cons needs to be abstract.
// Missing implementation for member of trait ParserBridge2:
//   def apply(x1: A, x2: List[A]): List[A] = ??? // implements `def apply(x1:
 T1, x2: T2): R`
//
// class Cons[A] extends ParserBridge2[A, List[A], List[A]]
//       ^^^^
```

### `ParserSingletonBridge[+T]`

All the generic bridges extend the `ParserSingletonBridge` trait instantiated to a function type. For example, `trait ParserBridge2[-A, -B, +C] extends ParserSingletonBridge[(A, B) => C]`. This means that every bridge uniformly gets access to a couple of extra combinators in addition to their `apply`:

```
trait ParserSingletonBridge[+T] {
    final def from(op: Parsley[_]): Parsley[T]
    final def <#(op: Parsley[_]): Parsley[T] = this.from(op)
}
```

The implementation of `from` is not important, it will be handled by the other `ParserBridgeN`s. What these two combinators give you is the ability to write `Foo.from(parser): Parsley[(Int, Int) => Foo]`, for instance. This can be useful when you want to use a bridge somewhere where the arguments cannot be directly applied, like in **chain** or **precedence** combinators:

```
import parsley.expr.chain
import parsley.syntax.character.stringLift

val term = chain.left1(px, Add.from("+")) // or `Add <# "+"`
```

They are analogous to the `as` and `#>` combinators respectively.

### `ParserBridge0[+T]`

This trait is a special case for objects that should return themselves. As an example, here is an object which forms part of a larger AST, say:

```
import parsley.generic.ParserBridge0
trait Expr
// rest of AST
case object NullLit extends Expr with ParserBridge0[Expr]
```

The `NullLit` object is part of the `Expr` AST, and it has also mixed in `ParserBridge0[Expr]`, giving it access to `from` and `<#` only (no `apply` for this one!). What this means is that you can now write the following:

```
val nullLit = NullLit <# "null"
// nullLit: Parsley[Expr] = parsley.Parsley@2c646f52
nullLit.parse("null")
// res2: parsley.Result[String, Expr] = Success(NullLit)
```

Without any further configuration, notice that the result of parsing `"null"` is indeed `NullLit`, and `nullLit: Parsley[Expr]`.

Be aware that the type passed to the generic parameter cannot be itself:

```
case object Bad extends ParserBridge0[Bad.type]
// error: illegal cyclic reference involving object Bad
// case object Bad extends ParserBridge0[Bad.type]
//                                       ^^^
```

Resolving this will require introducing an extra type, like `Expr` in the example with `NullLit`, which breaks the cycle sufficiently.

## Additional Use Cases

Other than the natural decoupling that the bridges provide, there are some more specialised uses that can come out of the generic bridges alone.

### Normalising or Disambiguating Data

Occasionally, the shape of an AST can change internally even though the syntax of the language being parsed does not. Bridges are perfectly suited for handling these internal changes while masking them from the parser itself. As an example, assume that a `Let` AST node was *previously* defined as follows:

```
case class Let(bindings: List[Binding], body: Expr)
object Let extends ParserBridge2[List[Binding], Expr, Let]
```

The parser, therefore, can be expected to produce lists of bindings to feed in. However, later it was decided that the ordering of the bindings doesn't matter, so a `Set` is being used. The decoupling of the bridge will allow for this change to happen without changing the parser, so long as the bridge performs the "patching":

```
case class Let(bindings: Set[Binding], body: Expr)
object Let extends ParserBridge2[List[Binding], Expr, Let] {
    def apply(bindings: List[Binding], body: Expr): Let
 = Let(bindings.toSet, body)
}
```

By defining the appropriate "forwarding" in its own `apply`, the bridge has ensured the parser will still work.

Handling ambiguity Another use of this kind of bridge is to allow for the disambiguation of two syntactically similar structures. As an example, consider Scala's tuple syntax:

```
val x  = (6)
val xy = (5, 6)
```

It is clear to us that `x` is not a "singleton pair", whatever that would be, but a parenthesised expression; on the other hand, `xy` is clearly a pair. The problem is that the syntax for these overlap, requiring backtracking to resolve (you can only know if you're parsing a tuple when you find your first `','`).

In practice, arbitrary backtracking in a parser can impact performance and the quality of error messages. Instead of dealing with this ambiguity by backtracking, it is possible to exploit the shared structure in a *Disambiguator Bridge*: this is just a bridge that looks at the provided arguments to decide what to make. For example:

```
import cats.data.NonEmptyList

case class Tuple(exprs: NonEmptyList[Expr]) extends Expr

object TupleOrParens extends ParserBridge1[NonEmptyList[Expr], Expr] {
    def apply(exprs: NonEmptyList[Expr]): Expr = exprs match {
        case NonEmptyList(expr, Nil) => expr
        case exprs                   => Tuple(exprs)
    }
}
```

In the above example, the parser will parse one or more expressions (signified by the `NonEmptyList` from `cats`); the bridge will then inspect these expressions, returning a single expression if only one was parsed, and construct the `Tuple` node otherwise. In the parser, this would just look something like:

```
// from `parsley-cats`, produces `NonEmptyList` instead of `List`
import parsley.cats.combinator.sepBy1
val tupleOrParens = TupleOrParens("(" ~> sepBy1(nullLit, ",") <~ ")")
// tupleOrParens: Parsley[Expr] = parsley.Parsley@364469d9
tupleOrParens.parse("(null)")
// res4: parsley.Result[String, Expr] = Success(NullLit)
tupleOrParens.parse("(null,null)")
// res5: parsley.Result[String, Expr] = Success(Tuple(NonEmptyList(NullLit,
 NullLit)))
```

## Enforcing Invariances

So far, the bridges we've seen have been altering the way that the data itself should be constructed from the results. However, it may also be desirable to override the *templated* `apply` to perform additional checks. This basically means that you can add in a `filter`-like combinator after the data has been constructed to validate that the thing you've constructed is actually correct. As an example, it turns out that Scala only allows tuples with a maximum of 22 elements:

```
val oops =
 (1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3)
// error: tuples may not have more than 22 elements, but 23 given
// val oops = (1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1,
 2, 3)
//
  ^
```

How to adjust the parser to handle this? One possible approach is to use the `range` combinator:

```
def nonEmptyList[A](px: Parsley[A], pxs: Parsley[List[A]]) =
    lift2(NonEmptyList(_, _), px, pxs)
val tupleOrParensObtuse =
    TupleOrParens("(" ~> nonEmptyList(nullLit, range(0, 21)("," ~> nullLit))
 <~ ")")
```

This works, but it's very obtuse. Not to mention that the error message generated isn't particularly good. Instead, we can hook some extra behaviour into the generated `apply`:

```
import parsley.errors.combinator._

object TupleOrParens extends ParserBridge1[NonEmptyList[Expr], Expr] {
    def apply(exprs: NonEmptyList[Expr]): Expr = exprs match {
        case NonEmptyList(expr, Nil) => expr
        case exprs                   => Tuple(exprs)
    }

    override def apply(exprs: Parsley[NonEmptyList[Expr]]): Parsley[Expr] =
        super.apply(exprs).guardAgainst {
```

```
            case Tuple(exprs) if exprs.size > 22 =>
                Seq(s"tuples may not have more than 22 elements,
 but ${exprs.size} given")
        }
}
```

This bridge invokes the templated `apply` with `super.apply` first, then after processes it with the `guardAgainst` combinator to generate a bespoke message:

```
tupleOrParens.parse("(null,null,null,null,null,null,null,null,null,null,null,null,null,nul
// res7: parsley.Result[String, Expr] = Failure((line 1, column 1):
//   tuples may not have more than 22 elements, but 23 given
//
 >(null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,nu
//
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

To be clear, this is using the original definition of `tupleOrParens` and *not* `tupleOrParensObtuse`.

> ⚠
>
> While this use of bridges does retain the unsaturated application from `ParserSingletonBridge`, using the `from` combinator will not perform the additional validation: be careful!

## When *not* to use

Simply put, generic bridges have one major limitation: they cannot interact with additional metadata that might be required in the parser. One excellent example of this is position information. While `parsley` could take a stance on how this should be done, I'd prefer if the users can make that decision for themselves. The previously linked tutorial demonstrates how to *make* templating bridges from scratch, which you would need to do to support something like position tracking.

# Synactic Extensions (`parsley.syntax`)

The `parsley.syntax` package contains several modules that enable new "syntax" on parsers or other values. There are currently four such modules:

- `parsley.syntax.character`: contains conversions that allow for character and string literals to serve as parsers.
- `parsley.syntax.lift`: enables the `lift` method on functions to allow them to work on parsers.
- `parsley.syntax.zipped`: enables the `zipped` method on tuples of parsers to sequence and combine their results with a single function.

## Implicit Conversions

The `charLift` and `stringLift` conversions in `parsley.syntax.character` allow for Scala character and string literals to work directly as parsers for those specific literals. For example:

```
import parsley.syntax.character._

val p = 'a' ~> "bc"
// p: parsley.Parsley[String] = parsley.Parsley@79dc2e0b
p.parse("abc")
// res0: parsley.Result[String, String] = Success(bc)
p.parse("axy")
// res1: parsley.Result[String, String] = Failure((line 1, column 2):
//   unexpected "xy"
//   expected "bc"
//   >axy
//     ^^)
```

In the above, `'a': Parsley[Char]`, and `"bc": Parsley[String]`.

If you see an error like this, when you otherwise have the implicit imported:

```
val p = "cb" <~ 'a'
p.parse("cba")
// error: value <~ is not a member of String
// val p = "cb" <~ 'a'
//         ^^^^^^^
```

Then this likely means that you have *another* conversion in scope and the ambiguity is not resolved. If the arguments reversed, this will become more evident:

```
val p = 'a' ~> "bc"
p.parse("abc")
// error: type mismatch;
//  found   : String("bc")
//  required: parsley.Parsley[?]
// Note that implicit conversions are not applicable because they are
 ambiguous:
//  both method stringLift in object character of type (str: String):
 parsley.Parsley[String]
//  and method implicitSymbol in class ImplicitSymbol of type (s:
 String): parsley.Parsley[Unit]
//  are possible conversion functions from String("bc") to
 parsley.Parsley[?]
```

In this case, a `lexer.lexeme.symbol.implicits` is imported and is clashing.

## Improved Sequencing

Both the `lift` and `zipped` modules within `parsley.implicits` enable new ways of sequencing parsers in an idiomatic way. The `lift` syntax is perhaps more natural, where the function to apply appears to the left of the arguments:

```
import parsley.character.char
import parsley.syntax.lift._

val add = (x: Int, y: Int) => x + y
// add: (Int, Int) => Int = <function2>
add.lift(char('a').as(5), char('b').as(6)).parse("ab")
// res4: parsley.Result[String, Int] = Success(11)
```

However, while `lift` works well when the function has its type fully elaborated, it does not infer well:

```
(_ + _).lift(char('a').as(5), char('b').as(6)).parse("ab")
// error: missing parameter type for expanded function ((<x$2: error>, x$3) =>
 x$2.$plus(x$3))
// (_ + _).lift(char('a').as(5), char('b').as(6)).parse("ab")
//      ^
// error: missing parameter type for expanded function ((<x$2: error>, <x$3:
 error>) => x$2.$plus(x$3))
```

This is where `zipped` comes in: by placing the function to the right of its arguments, it can infer the type of the function based on the arguments. This may appear slightly less natural, however:

```
import parsley.syntax.zipped._
(char('a').as(5), char('b').as(6)).zipped(_ + _).parse("ab")
// res6: parsley.Result[String, Int] = Success(11)
```

Both `lift` and `zipped` work for up to 22-argument functions.

# Position Combinators (`parsley.position`)

During a parse, position information is tracked and recorded by `parsley`. The `parsley.position` module contains combinators that can access this information so that it can be used in the construction of the results.

> *The Scaladoc for this page can be found at* `parsley.position`.

## Position Extraction

The `line` and `col` parsers will extract the line and column respectively. In `parsley`, the first line and column are both 1. For convenience, the `pos` parser is defined as `line.zip(col)`.

## Offset Extraction

The `offset` parser can be used to obtain the raw underlying offset of a parser, which can be used for calculations on the number of characters (not codepoints) a parser consumes, but should not be used to derive positions themselves.

# Context-Sensitive Parsing (`parsley.state`)

Normally, context-sensitive parsing can be done with monadic `flatMap`. However, in `parsley`, `flatMap` is a very expensive operation and is best avoided. Instead, `parsley` supports a form of arbitrary state threading called *references*. These can be used to perform context-sensitive parsing in a more performant way at a cost to expressive power.

> *The Scaladoc for this page can be found at* `parsley.state`.

## References

A reference is a single piece of mutable state threaded through a parser. They can be made in three different ways:

1. Importing `parsley.state.RefMaker` allows the use of the `makeRef` method on any type `A`:

   ```
   def makeRef[B](body: Ref[A] => Parsley[B]): Parsley[B]
   ```

   This will construct a new reference filled with the receiver value, and this can be used within the scope of the given continuation. Everytime this is executed, it will be uniquely scoped (when the parser is recursive).

2. Similarly, importing `parsley.state.StateCombinators` allows the use of the `fillRef` combinator on `Parsley[A]`, which has the same signature as `makeRef`.

   It behaves similarly to `makeRef`, but sources its result from a parser. Loosely: `p.fillRef(body)` is the same as `p.flatMap(_.makeRef(body))`, but is much more efficient.

3. The `Ref.make[A]: Ref[A]` method allows for the creation of a reference *outside* of the parsing context. This is not recommended, as it will not guarantee uniqueness of scoping. The state will **not** be initialised.

   > Using the same globally constructed reference in two places is undefined behaviour and should *not* be done.

References themselves have two core operations, with several more built on top:

```
class Ref[A] {
    def get: Parsley[A]
    def set(p: Parsley[A]): Parsley[Unit]
}
```

The `get` method will read the reference at parse-time and return the value contained within. The `set` method takes the result of a parser and stores that into the reference. As examples:

```
import parsley.character.item
import parsley.state._

List.empty[Char].makeRef { r1 =>
    item.fillRef { r2 =>
        r1.set(r2.get <::> (r2.get <::> r1.get))
    } *> r1.get
}.parse("a")
// res0: parsley.Result[String, List[Char]] = Success(List(a, a))
```

The above example fills a reference `r1`, with the empty list, then fills a second reference `r2` with the result of parsing any character. The value in `r1` is updated with the list obtained by prepending the parsed character onto the empty list stored in `r1` twice. After `r2` goes out of scope, the current value of `r1` is returned.

## Persistence

In the above example, the reference `r2` is only used for `get`, but is used multiple times. Normally, using the value of a parser more than once requires a `flatMap`; this is not the case in `parsley` when using references. To make this application more ergonomic, `parsley.state.StateCombinators` also exposes the `persist` combinator:

```
List.empty[Char].makeRef { r1 =>
    item.persist { c =>
        r1.set(c <::> (c <::> r1.get))
    } *> r1.get
}.parse("a")
// res1: parsley.Result[String, List[Char]] = Success(List(a, a))
```

Persist can be thought of as a composition of `fillRef` and `get`, or alternatively as a composition of `flatMap` and `pure`.

### Using Persistence

One use of `persist` is to otherwise reduce the scope of an expensive `flatMap`: the `flatMap` combinator is expensive because it has to process the body of the function in full everytime it is executed, if the size of the

body is reduced, that will keep the parser faster. Currently, there is no primitive functionality for parsing with respect to values inside references, like so:

```
def string(r: Ref[String]): Parsley[String]
 = r.get.flatMap(parsley.character.string(_))
```

The scope of the `flatMap` in that combinator is small, however, so is likely much more efficient than one that didn't use persistence. With this, the context-sensitive parsing of XML tags can be done:

```
import parsley.Parsley.{atomic, notFollowedBy}
import parsley.character.{stringOfSome, letter}
import parsley.combinator.optional
import parsley.syntax.character.{charLift, stringLift}

val openTag = atomic('<' <~ notFollowedBy('/'))
// openTag: Parsley[Char] = parsley.Parsley@3499ebb7
val tagName = stringOfSome(letter)
// tagName: Parsley[String] = parsley.Parsley@3a7d11b4

lazy val content: Parsley[Unit] = optional(tag)
lazy val tag: Parsley[Unit] = (openTag ~> tagName <~ '>').fillRef { name =>
    content <~ ("</" ~> string(name) <~ ">")
}

tag.parse("<hello></hello>")
// res2: parsley.Result[String, Unit] = Success(())
tag.parse("<hello></hi>")
// res3: parsley.Result[String, Unit] = Failure((line 1, column 10):
//   unexpected "hi>"
//   expected "hello"
//   ><hello></hi>
//             ^^^)
tag.parse("<a><b></b></c>")
// res4: parsley.Result[String, Unit] = Failure((line 1, column 13):
//   unexpected "c"
//   expected "a"
//   ><a><b></b></c>
//                ^)
```

## Long-Term State

Persistence is an example of read-only state used to preserve a value for later. Writable state can also be used for context-sensitive tasks, by tracking a value over time. Examples include whitespace-sensitivity or tracking matching brackets.

### Matching Brackets

The `setDuring` and `updateDuring` combinators can be used to reset state after a specific context is executed. They can be built primitively out of `get`, `set`, and `persist`. The following parser reports the position of the last unclosed bracket that is well-interleaved with the other kinds of brackets.

```
import parsley.Parsley.{eof, many}
import parsley.character.char
import parsley.errors.patterns.VerifiedErrors
import parsley.position.pos

case class Brackets(open: Char, position: (Int, Int)) {
    def enter(c: Char, p: (Int, Int)) = {
        val (line, col) = p
        // adjust the column, because it was parsed post-bracket
        this.copy(open = c, position = (line, col-1))
    }
    def missingClose = s"unclosed $open originating at $position"
}
object Brackets {
    def empty = Brackets(0, null) // this will never be matched on
    def toOpen(c: Char) = c match {
        case ')' => '('
        case ']' => '['
        case '}' => '{'
    }
}

def brackets = Brackets.empty.makeRef { bs =>
    def open(c: Char): Parsley[Unit] =
        char(c) ~> bs.update(pos.map[Brackets => Brackets](p => _.enter(c, p)))
    def close(c: Char): Parsley[Unit] =
        char(c).void | bs.get.verifiedExplain(_.missingClose)

    // ensure it is reset
    def scope[A](p: Parsley[A]): Parsley[A]
 = bs.updateDuring(identity[Brackets])(p)

    lazy val matching: Parsley[Unit] = scope {
        many {
              open('(') ~> matching <~ close(')')
            | open('[') ~> matching <~ close(']')
            | open('{') ~> matching <~ close('}')
        }.void
    }
    matching <~ eof
}
```

The above parser is designed to report where the last unclosed bracket was. It creates a reference `bs` that stores a `Brackets`, which tracks the last open character and its position. Then, whenever a bracket

is entered, `matching` will save the existing information using the `updateDuring` combinator: giving it the `identity` function will mean it will simply restore the existing state after it returns. Whenever an open bracket is parsed, it will write its position into the state (lagging by one character), and then if the corresponding closing bracket cannot be parsed, it will use an unconditional ***Verified Error*** to report a message based on the last opened bracket. The results are below:

```
val p = brackets
// p: Parsley[Unit] = parsley.Parsley@3502518c

p.parse("()()()")
// res5: parsley.Result[String, Unit] = Success(())
p.parse("[][][]")
// res6: parsley.Result[String, Unit] = Success(())
p.parse("{}[]()")
// res7: parsley.Result[String, Unit] = Success(())
p.parse("[[[[[]]]]]")
// res8: parsley.Result[String, Unit] = Success(())

p.parse("([)]")
// res9: parsley.Result[String, Unit] = Failure((line 1, column 3):
//   unexpected ")"
//   expected "(", "[", "]", or "{"
//   unclosed [ originating at (1,2)
//   >([)]
//      ^)
p.parse("({(")
// res10: parsley.Result[String, Unit] = Failure((line 1, column 4):
//   unexpected end of input
//   expected "(", ")", "[", or "{"
//   unclosed ( originating at (1,3)
//   >({(
//       ^)
p.parse("()[]{[(){}}")
// res11: parsley.Result[String, Unit] = Failure((line 1, column 11):
//   unexpected "}"
//   expected "(", "[", "]", or "{"
//   unclosed [ originating at (1,6)
//   >()[]{[(){}}
//              ^)
```

Given the relatively simple construction, it works quite well, and efficiently too: no `flatMap` necessary!

### Tail-Recursive Combinators

When combinators can be implemented tail recursively instead of recursively, they can be more efficient. In the context of `parsley`, tail-recursive combinators are ones which only return the result of the last recursive call they make:

```
lazy val tailRec: Parsley[Unit] = 'a' ~> tailRec | unit
```

The above is tail recursive, for instance. Combinators like `skipMany` are implemented tail recursively, with additional optimisations to make them more efficient: implementing new combinators in terms of `skipMany` with references to carry state is likely to be efficient. For example:

```
def setOf[A](p: Parsley[A]): Parsley[Set[A]] = {
    Set.empty[A].makeRef { set =>
        many(set.update(p.map[Set[A] => Set[A]](x => _ + x))) ~> set.get
    }
}
```

In the above code, a set is carried around in a reference, and a new element is added into this set every iteration. When the loop completes (successfully), the set in the reference is returned. A more efficient implementation, however, would use `persist` and a mutable set (along with `impure` and `fresh`): this, of course, still uses a reference.

### Whitespace-Sensitive Languages

Another application of long-term state is to track indentation levels in a whitespace-sensitive language: here the start of a new cause, statement or block will record the current column number, and further statements must verify that the column number is correct. Leaving a block will restore the identation level back to how it was before. This is similar to how matching brackets worked.

## Stateful Combinators

For some applications, a more structured strategy for tracking state can be useful. The `forP` combinators allow for looping where the a stateful variable is used to control the control flow. For instance, the classic context-sensitive grammar of `a^n b^n c^n` can be matched effectively using a reference and a `forP`:

```
import parsley.Parsley.pure
import parsley.state.forP
val abcs = 0.makeRef { i =>
    many('a' ~> i.update(_ + 1)) ~>
    forP[Int](i.get, pure(_ > 0), pure(_ - 1)) {
        'b'
    } ~>
    forP[Int](i.get, pure(_ > 0), pure(_ - 1)) {
        'c'
    } ~>
    i.get <~ eof
}
// abcs: Parsley[Int] = parsley.Parsley@1fed9ed8

abcs.parse("aabbcc")
```

```
// res12: parsley.Result[String, Int] = Success(2)
abcs.parse("aaaaabbbbbccccc")
// res13: parsley.Result[String, Int] = Success(5)
abcs.parse("aaaabbbbbbcccc")
// res14: parsley.Result[String, Int] = Failure((line 1, column 9):
//   unexpected "b"
//   expected "c"
//   >aaaabbbbbbcccc
//            ^)
```

First, as many `a`s as possible are read and each one will increment the counter `i`. Then, run the equivalent of a C-style: `for (int j = i, j > 0, j -= 1)` reading `b`s and `c`s. Internally, `forP` will use a reference to track its state.

# Basic Debug Combinators (`parsley.debug`)

Parsley has a collection of basic debugging utilities found within **`parsley.debug`**. These can help debug errant parsers, understand how error messages have been generated, and provide a rough sense of what parsers take the most time to execute.

The combinators themselves are all contained within **`parsley.debug.DebugCombinators`**.

> *The Scaladoc for this page can be found at* `parsley.debug` *and* `parsley.debug.DebugCombinators`

## Debugging Problematic Parsers (`debug`)

The most common quick debugging combinator is **`debug`**, which at its simplest prints some information on entering and exiting a combinator:

```
import parsley.Parsley.atomic
import parsley.character.string
import parsley.debug, debug._

val hello = ( atomic(string("hello").debug("hello")).debug("atomic1")
            | string("hey").debug("hey")
            | string("hi").debug("hi")
            )
// hello: parsley.Parsley[String] = parsley.Parsley@57bf6b19

debug.disableColourRendering()

hello.parse("hey")
// >atomic1> (1, 1): hey•
//                   ^
//   >hello> (1, 1): hey•
//                   ^
//   <hello< (1, 3): hey• Fail
//                     ^
// <atomic1< (1, 1): hey• Fail
//                   ^
// >hey> (1, 1): hey•
//               ^
// <hey< (1, 4): hey• Good
//                  ^
// res1: parsley.Result[String, String] = Success(hey)
```

```
hello.parse("hi")
// >atomic1> (1, 1): hi•
//                   ^
//   >hello> (1, 1): hi•
//                   ^
//   <hello< (1, 2): hi• Fail
//                    ^
// <atomic1< (1, 1): hi• Fail
//                   ^
// >hey> (1, 1): hi•
//               ^
// <hey< (1, 2): hi• Fail
//                ^
// res2: parsley.Result[String, String] = Failure((line 1, column 1):
//   unexpected "hi"
//   expected "hello" or "hey"
//   >hi
//    ^^)
```

In the above example, an unexpected failure to parse the input `"hi"` is being debugged using `debug`. Each of the `string` combinators are annotated, as well as the `atomic`. This allows us to see the control flow of the parser as it executes, as well as where the input was read up to. In this case, we can see that `atomic` has undone input consumption, but that doesn't apply to `hey`. In other words, there is another `atomic` missing! We could have added `debug` to the `|` combinator as well to make it even clearer, of course. Though not visible above, the output is usually coloured. If this causes problems, `debug.disableColourRendering()` will disable it, or the `coloured` parameter can be set to `false` on the combinator.

## Breakpoints

The `Breakpoint` type has values `EntryBreak`, `ExitBreak`, `FullBreak`, and `NoBreak`, which is the default. `FullBreak` has the effect of both `EntryBreak` and `ExitBreak` combined: `EntryBreak` will pause execution on the entry to the combinator, requiring input on the console to proceed, and `ExitBreak` will do the same during the exit.

## Watching References

The `debug` combinator takes a variadic number of reference/name pairs as its last argument. These allow you to watch the values stored in references as well during the debugging process. For instance:

```
import parsley.Parsley.atomic
import parsley.state._
import parsley.character.string
import parsley.debug._

val p = 0.makeRef { r1 =>
```

```
    false.makeRef { r2 =>
        val p = (  string("hello")
                ~> r1.update(_ + 5)
                ~> ( string("!")
                   | r2.set(true) ~> string("?")
                   ).debug("punctuation", r2 -> "r2")
                )
        r1.rollback(atomic(p).debug("hello!", r1 -> "r1", r2 -> "r2"))
          .debug("rollback", r1 -> "r1")
    }
}
// p: parsley.Parsley[String] = parsley.Parsley@558063b4

p.parse("hello world")
// >rollback> (1, 1): hello·
//                    ^
// watched registers:
//     r1 = 0
//
//   >hello!> (1, 1): hello·
//                    ^
//   watched registers:
//       r1 = 0
//       r2 = false
//
//     >punctuation> (1, 6): hello·world·
//                                 ^
//     watched registers:
//         r2 = false
//
//     <punctuation< (1, 6): hello·world· Fail
//                                 ^
//     watched registers:
//         r2 = true
//
//   <hello!< (1, 1): hello· Fail
//                    ^
//   watched registers:
//       r1 = 5
//       r2 = true
//
// <rollback< (1, 1): hello· Fail
//                    ^
// watched registers:
//     r1 = 0
//
// res3: parsley.Result[String, String] = Failure((line 1, column 6):
//   unexpected space
//   expected "!" or "?"
//   >hello world
//         ^)
```

## Debugging Error Messages (`debugError`)

The `debugError` is a slightly more experimental combinator that aims to provide some (lower-level) insight into how an error message came to be. For instance:

```
import parsley.character.{letter, digit, char}
import parsley.Parsley.many
import parsley.debug._

val q = (many( ( digit.debugError("digit")
               | letter.debugError("letter")
               ).debugError("letter or digit")
             ).debugError("many letterOrDigit")
      ~> many(char('@').debugError("@")).debugError("many @")
      ~> char('#').debugError("#")) | char('!').debugError("!")
// q: parsley.Parsley[Char] = parsley.Parsley@284e9cc8

q.parse("$")
// >many letterOrDigit> (offset 0, line 1, col 1): current hints are Set()
 (valid at offset 0)
//   >letter or digit> (offset 0, line 1, col 1): current hints are Set()
 (valid at offset 0)
//     >digit> (offset 0, line 1, col 1): current hints are Set() (valid at
 offset 0)
//     <digit< (offset 0, line 1, col 1): Fail
//     generated vanilla error (offset 0, line 1, col 1) {
//       unexpected item = "$"
//       expected item(s) = Set(digit)
//       reasons = no reasons given
//     }
//     >letter> (offset 0, line 1, col 1): current hints are Set() (valid at
 offset 0)
//     <letter< (offset 0, line 1, col 1): Fail
//     generated vanilla error (offset 0, line 1, col 1) {
//       unexpected item = "$"
//       expected item(s) = Set(letter)
//       reasons = no reasons given
//     }
//   <letter or digit< (offset 0, line 1, col 1): Fail
//   generated vanilla error (offset 0, line 1, col 1) {
//     unexpected item = "$"
//     expected item(s) = Set(letter, digit)
//     reasons = no reasons given
//   }
// <many letterOrDigit< (offset 0, line 1, col 1): Good, current hints are
 Set(letter, digit) with all added since entry to debug (valid at offset 0)
// >many @> (offset 0, line 1, col 1): current hints are Set(letter, digit)
 (valid at offset 0)
//   >@> (offset 0, line 1, col 1): current hints are Set(letter, digit) (valid
 at offset 0)
//   <@< (offset 0, line 1, col 1): Fail
//   generated vanilla error (offset 0, line 1, col 1) {
```

```
//     unexpected item = "$"
//     expected item(s) = Set("@", letter, digit)
//     reasons = no reasons given
//   }
// <many @< (offset 0, line 1, col 1): Good, current hints are Set("@", letter,
 digit) with Set("@") added since entry to debug (valid at offset 0)
// >#> (offset 0, line 1, col 1): current hints are Set("@", letter, digit)
 (valid at offset 0)
// <#< (offset 0, line 1, col 1): Fail
// generated vanilla error (offset 0, line 1, col 1) {
//   unexpected item = "$"
//   expected item(s) = Set("@", letter, digit, "#")
//   reasons = no reasons given
// }
// >!> (offset 0, line 1, col 1): current hints are Set() (valid at offset 0)
// <!< (offset 0, line 1, col 1): Fail
// generated vanilla error (offset 0, line 1, col 1) {
//   unexpected item = "$"
//   expected item(s) = Set("!")
//   reasons = no reasons given
// }
// res4: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected "$"
//   expected "!", "#", "@", digit, or letter
//   >$
//    ^)
```

In the above example, you can see how each individual error is raised, as well as evidence of merging, and how errors can be turned into "hints" if the error is successfully recovered from: this means that the label may be re-incorporated into the error again later if they are at the valid offset, as seen in the errors for `char('#')`.

## Profiling Parser (`Profiler`)

The `Profiler` class, and the accompanying `profile` combinator, provide a *rough* guideline of how much of the runtime a parser might be taking up. The execution of each combinator is measured with a resolution of 100ns. First, a `Profiler` object must be set up and implicitly available in scope: its role is to collect the profiling samples. Then, a parser annotated with `profile` combinators is ran, and the results can be displayed with `profiler.summary()`.

There is a disclaimer that the profiler "just provides data", no guarantee about its statistical significance is given. Multiple runs can be performed and these will be aggregated, the `profiler` can be cleared using `clear()`.

```
import parsley.Parsley
import parsley.character.{string, char}
import parsley.combinator.traverse
import parsley.debug._

def classicString(s: String): Parsley[String] =
    traverse(char(_), s.toList: _*).map(_.mkString)

implicit val profiler: Profiler = new Profiler
// profiler: Profiler = ...
val strings = many(classicString("...").profile("classic string")
               <~> string("!!!").profile("optimised string"))
// strings: Parsley[List[(String, String)]] = ...
val stringsVoid = many(classicString("...").profile("voided classic string")
                   <~> string("!!!").profile("voided optimised string")).void
// stringsVoid: Parsley[Unit] = ...

strings.parse("...!!!" * 10000)
// res5: parsley.Result[String, List[(String, String)]] = ...
stringsVoid.parse("...!!!" * 10000)
// res6: parsley.Result[String, Unit] = ...

profiler.summary()
// name                    self time   num calls   average self time
// ------------------------------------------------------------------
// classic string          24087.5µs       10001             2.408µs
// voided optimised string   822.3µs       10000             0.082µs
// optimised string         7072.3µs       10000             0.707µs
// voided classic string    2492.5µs       10001             0.249µs
// ------------------------------------------------------------------
```

The above example shows that the **`string`** combinator is much faster than the "classic" definition in terms of **`traverse`** and **`char`** (not even accounting for its improved error messages!). However, it also shows that when the results are not required (as indicated by the **`void`** combinator, which *aggressively* suppresses result generation underneath it), the combinators perform much more similarly. You will, however, notice variance depending on when you visit this page: these results are generated each publish, and sometimes the non-voided **`string`** can outperform the voided one by pure chance!

The **`profile`** combinator will account for other profiled combinators underneath it, accounting for their "self time" only. This helps to measure the impact of a specific sub-parser more accurately.

# Chain Combinators

The `parsley.expr.chain` module contains a variety of combinators for abstracting, most commonly, the application of operators to values in expressions. This allows `parsley` to handle *left recursion* idiomatically. To distinguish between these chains and the functionality found in `parsley.expr.infix`, it is recommended to always import this module qualified as `import parsley.expr.chain` -- except for `postfix` and `prefix`.

*The Scaladoc for this page can be found at* `parsley.expr.chain`.

## Binary Chains

The first kind of chains found in `chain` are the binary chains, which handle infix application of a binary operator to values in either a left- or right-associative way. These are called `chain.left1` or `chain.right1`. The `1` here means that there must be at least one value present, though there may be no operators. As an example:

```
p op p op p op p op p
```

The above can be parsed using `chain.left1(p, op)` to have the effect of parsing like:

```
(((p op p) op p) op p) op p
```

It can also be parded using `chain.right1(p, op)` to have the effect of parsing like:

```
p op (p op (p op (p op p)))
```

Both of these combinators share the same type, where the parser `p: Parsley[A]`, and the parser `op: Parsley[(A, A) => A]`. This means that the two combinators can be freely swapped between in an implementation. This is useful when the grammar being encoded for is fully-associative and the associativity within the parser is an implementation detail.

However, if more type-safety is desired, the `infix.left1` and `infix.right1` combinators may be more appropriate.

## Unary Chains

The other kind of chains found in `chain` are unary chains, which handle repeated prefix or postfix application of an operator to a single value. These are called `chain.prefix` and `chain.postfix`. There also `1` variants of these combinators, which will be discussed later.

Given input of shape:

```
p op op op op
```

The combinator `postfix p op` will parse the input and apply results such that it would look like:

```
(((p op) op) op) op
```

Similarly, given input of shape:

```
op op op op p
```

The combinator `prefix op p` will parse the input and apply results such that it would look like:

```
op (op (op (op p)))
```

Unlike `chain.left1` and `chain.right1`, there is no `infix` equivalent for `prefix` and `postfix`. This is because a refined type will not add much to the way the combinators operate.

### Ensuring an operator exists

Unlike the difference between `chain.left` and `chain.left1`, which allows for an absence of terminal value; `prefix` vs `prefix1` describe whether or not an operator is required or not. This is enforced by the type signature of the operations themselves:

```
def prefix1[A, B <: A](op: Parsley[A => B], p: Parsley[A]): Parsley[B]
```

Given that `B` is a subtype of `A`, it is not possible for the `p`'s result of type `A` to be the final return value of type `B`. As such, an operator must be parsed which wraps the `A` into a `B`. The subtying relation then allows a nested application of an operator to be upcast into an `A` so it can be fed into another layer of operator.

# Infix Chain Combinators

The `parsley.expr.infix` module contains combinators for the strongly-typed abstraction of operators to values in expressions. This allows `parsley` to handle left recursion idiomatically. To distinguish between these chains and those found in **parsley.expr.chain**, it is recommended to always import this module qualified as `import parsley.expr.infix`.

> *The Scaladoc for this page can be found at* `parsley.expr.infix`.

## Binary Chains

The stronger types implied by `infix` means that only binary chains are useful to define. Both `infix.left1` and `infix.right1` behave the same as `chain.left1` and `chain.right1`. The difference is in the types:

```
def left1[A, B](p: Parsley[A], op: Parsley[(B, A) => B])(implicit wrap: A
 => B): Parsley[B]
def right1[A, B](p: Parsley[A], op: Parsley[(A, B) => B])(implicit wrap: A
 => B): Parsley[B]
```

As both combinators return type `B`, the `B` denotes where recursion can appear. The `wrap` function is a way of converting the final `p` value in either bracketing to be a value of type `B` to fit into the operator. When `A <: B`, the `A <:< B` instance, which is of type `A => B` will be summoned, meaning the terminal `p` is upcast into a `B`. When `A` and `B` are the same type, `A =:= B` is summoned, which acts as the `identity` function, and they behave the same as `chain.left1` or `chain.right1`.

Ultimately, these chains are useful when the parser writer wants further guarantees that the parser adheres to the grammar precisely: when strong types are employed, `infix.left1` and `infix.right1` cannot be substituted for each other.

As an example:

```
import parsley.character.digit
import parsley.expr.infix
import parsley.syntax.character.stringLift

sealed trait Expr
case class Add(x: Expr, y: Num) extends Expr
case class Num(n: Int) extends Expr

lazy val expr = infix.left1(num, "+".as(Add(_, _)))
lazy val num = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit).map(Num(_))
expr.parse("56+43+123")
// res0: parsley.Result[String, Expr] =
 Success(Add(Add(Num(56),Num(43)),Num(123)))
```

In the above example, the `Add` constructor is recursive on the left, but `Num` must appear on the right. As `Num` and `Add` share a common supertype `Expr`, this is what the chains will return. To illustrate what happens if `right1` was used instead:

```
lazy val badExpr = infix.right1(num, "+".as(Add(_, _)))
// error: inferred type arguments [Num,Add,Num] do not conform to method
 right1's type parameter bounds [A,B,C >: B]
// lazy val badExpr = infix.right1(num, "+".as(Add(_, _)))
//                    ^^^^^^^^^^^^
// error: type mismatch;
//  found   : parsley.Parsley[Num]
//  required: parsley.Parsley[A]
// lazy val badExpr = infix.right1(num, "+".as(Add(_, _)))
//                                 ^^^
// error: type mismatch;
//  found   : parsley.Parsley[(Expr, Num) => Add]
//  required: parsley.Parsley[(A, C) => B]
// lazy val badExpr = infix.right1(num, "+".as(Add(_, _)))
//                                      ^^^^^^^^^^^^^^^^^
```

Notice that the error message here refers to the type `(A, C) => B`. In practice, to help reduce type ascriptions, the explicit `C >: B` is used as well -- in the working example, `C` is `Expr`, `A` is `Num`, and `B` is `Add`. The wrap is actually `A => C`, which in this case is provided by `Num <:< Expr`.

# Precedence Combinators

The chain combinators in `parsley.expr.{chain, infix}` are useful for handling isolated instances of expression-like behaviours in a grammar. However, they can be cumbersome to use when there are multiple interacting layers of expression operators. In these circumstances, a precedence table is the more appropriate choice. In `parsley`, the precedence machinery is very general, which can make the documentation intimidating. This page aims to break down how this works, with examples of different uses. The relevant combinator for turning a precedence table into a parser is the `precedence` combinator.

## Formulating Precedence

In general, a precedence table is formed up of a collection of base "atoms", and then one or more levels of operators. The precise shape these components take depends on how complex the types of the results are.

The simplest shape of `precedence` directly takes a variadic list of atoms and then a variadic list of levels as arguments: the levels lexically closest to the atoms will be the tightest-binding operators. This shape is discussed in **Homogeneous Precedence**, below.

More generally, however, the `precedence` combinator takes a *heterogeneous list* of levels of operators, where the base case of this structure contains the atoms. This can be used to handle more generic shapes of results, and is discussed in **Heterogeneous Precedence**.

However, both shapes make use of the `Ops` type, which requires a `Fixity` to be provided to it. First, it is good to understand the basics of how that works in a simplified presentation: for now, trust that the types work out. There are five different `Fixity` values: `InfixL`, `InfixR`, `Prefix`, `Postfix`, and `InfixN`. These each denote either binary or unary operators, and where recursion may appear for each of them. `InfixN` is a non-recursive operator, like < in some languages. At its simplest, `Ops` first takes a `Fixity` and then takes a variadic number of operators that should have that fixity and all have the same precedence. This will be explored in more detail in **`Ops` and `Fixity`**.

Throughout this section, the following imports and parsers are assumed:

```
import parsley.Parsley
import parsley.character.{letter, digit, stringOfSome}
import parsley.syntax.character.stringLift

val int: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)
val ident: Parsley[String] = stringOfSome(letter)
```

### Homogeneous Precedence

When the result type of the precedence table is the same throughout the table -- right through to the atoms, this is a *homogeneous* precedence table. This can be captured by the `Ops` smart-constructor, and make use of the simple flat structure. As an example:

```
import parsley.expr.{precedence, Ops, InfixL}

sealed trait Expr
case class Add(x: Expr, y: Expr) extends Expr
case class Sub(x: Expr, y: Expr) extends Expr
case class Mul(x: Expr, y: Expr) extends Expr
case class Num(n: Int) extends Expr
case class Var(v: String) extends Expr

val expr: Parsley[Expr] =
    precedence(ident.map(Var), int.map(Num))(
        Ops(InfixL)("*" as Mul),
        Ops(InfixL)("+" as Add, "-" as Sub)
    )
// expr: Parsley[Expr] = parsley.Parsley@77768dda

expr.parse("x+5*y")
// res0: parsley.Result[String, Expr] = Success(Add(Var(x),Mul(Num(5),Var(y))))
```

In the above example, notice that `Add` and `Sub` are within the same `Ops`, and therefore are the same precedence; `Mul` is closer to the atoms lexically than `Add` and `Sub` are, so it binds tighter. All levels have been marked as left associative with `InfixL`. When using `Prefix` or `Postfix`, the type of the parsers in the `Ops` will have type `Parsley[A => A]` instead of the above `Parsley[(A, A) => A]`.

### `Ops` and `Fixity`

The above example of `precedence` did not elaborate on how the types work to allow the operators to vary in type when the fixity does. Here is the type of the `Ops` constructor from above:

```
object Ops {
    def apply[A](fixity: Fixity)(ops: Parsley[fixity.Op[A, A]]*): Ops[A, A]
}
```

There are a few things to note here. First is that the type returned is `Ops[A, A]`: this means that the input type to this precedence layer matches the output type (this is because it's homogeneous). Second is that the type that the operators must return is `fixity.Op[A, A]`: this is known as a *path-dependent type*. Each object of type `Fixity` will describe the shape of operators that match that fixity, the two type parameters

are the input and output of the operator (again, for homogeneous precedence, these will coincide). This is formulated as follows:

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

These types align with those seen in `parsley.expr.infix`, as well as `Prefix` and `Postfix` matching the homogeneous shape in `parsley.expr.chain.{prefix, postfix}`. As such, when a specific fixity is provided to `Ops`, it also fixes the type of the operators in the next set of parentheses.

When considering heterogeneous combinators, the types `A` and `B` can indeed vary, and the more general `SOps` and `GOps` will be explored in the next section.

## Heterogeneous Precedence

When the results of the precedence table is not the same throughout, it is necessary to generalise the machinery to make use of either `SOps` or `GOps`:

```
object SOps {
    def apply[B, A <: B](fixity: Fixity)
(ops: Parsley[fixity.Op[A, B]]*): Ops[A, B]
}

object GOps {
    def apply[A, B](fixity: Fixity)(ops: Parsley[fixity.Op[A, B]]*)
                   (implicit wrap: A => B): Ops[A, B]
}
```

The `SOps` object allows the input and output to the layer to vary so long as they are in a sub-type relation: this is the most common form of heterogeneous hierachy that leverages Scala's strengths. Otherwise, `GOps` handles any arbitrary relationship between the types, so long as there is a known implicit `A => B` conversion.

Since the types between layers differ, a variadic argument list cannot be used to collect them together. Instead, the `Ops` are stitched together into a `Prec[A]` structure:

```
sealed trait Prec[A] {
    def +:[B](ops: Ops[A, B]): Prec[B]
    def :+[B](ops: Ops[A, B]): Prec[B]
}
case class Atoms[A](atoms: Parsley[A]*) extends Prec[A]
```

The atoms are placed into `Atoms`, which is the base case of the list. The `+:` and `:+` operators attach an additional layer onto the table, adapting the existing tables input type into a new output type via the given operators. Again, levels closer to the `Atoms` will bind tighter. As an example:

```
import parsley.expr.{precedence, SOps, Atoms, InfixL}

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

val expr: Parsley[Expr] =
    precedence {
        Atoms(ident.map(Var), int.map(Num)) :+
        SOps(InfixL)("*" as Mul) :+
        SOps(InfixL)("+"as Add, "-" as Sub)
    }
// expr: Parsley[Expr] = parsley.Parsley@5ea2e3d4

expr.parse("x+5*y")
// res1: parsley.Result[String, Expr] = Success(Add(Var(x),Mul(Num(5),Var(y))))
```

Here, the types within the syntax tree are very specific about the fact that all three operators are left associative. To be able to use this type with precedence `SOps` is required, since the outer layer takes in `Term`s and produces `Expr`s, the middle layer takes `Atoms` and produces `Term`s, and the inner-most atoms produce `Atoms`. Notice that, if the layers are incorrectly stitched together, the table does not typecheck:

```
import parsley.expr.InfixR

val expr: Parsley[Expr] =
    precedence {
        Atoms(ident.map(Var), int.map(Num)) :+
        SOps(InfixR)("*" as Mul) :+
        SOps(InfixL)("+" as Add, "-" as Sub)
    }
// error: type mismatch;
//  found   : parsley.Parsley[Mul.type]
//  required: parsley.Parsley[parsley.expr.InfixR.Op[Term,Term]]
```

```
//     (which expands to)  parsley.Parsley[(Term, Term) => Term]
//         SOps(InfixR)("*" as Mul) :+
//                      ^^^^^^^^^^
```

```
val expr: Parsley[Expr] =
    precedence {
        Atoms(ident.map(Var), int.map(Num)) :+
        SOps(InfixL)("+" as Add, "-" as Sub) :+
        SOps(InfixL)("*" as Mul)
    }
// error: inferred type arguments [Atom,Term] do not conform to method :+'s
 type parameter bounds [A' >: Expr,B]
//         SOps(InfixL)("+" as Add, "-" as Sub) :+
//                                              ^^
// error: type mismatch;
//  found   : parsley.expr.Ops[Atom,Term]
//  required: parsley.expr.Ops[A',B]
//         SOps(InfixL)("*" as Mul)
//         ^^^^^^^^^^^^
```

# Lexer (`parsley.token.Lexer`)

The `Lexer` class is the main-entry point to the combinator-based functionality of the `parsley.token` package. It is given configuration in the form of a `LexicalDesc` and an optional `ErrorConfig`. The internal structure is then a collection of objects that contain various forms of functionality: these are explored in more detail in this page.

It is worth noting the highest-level structure:

- `lexeme` and `nonlexeme` are the top level categorisation of functionality, accounting for whitespace
- `fully` is a combinator designed to be used around the **outer-most** parser, ran **at most once** during a parse, to consume leading whitespace and ensure all input is consumed.
- `space` is an object that allows for explicit interaction with whitespace parsing: this is really only important for whitespace-sensitive languages, and `lexeme` should be used for almost all other applications.

*The Scaladoc for this page can be found at* `parsley.token.Lexer`.

## Distinguishing Between "Lexeme" and "Non-Lexeme"

Broadly, the `Lexer` duplicates the vast majority of its functionality between two different objects: `lexeme` and `nonlexeme`. Broadly speaking, everything within `nonlexeme` can be found inside `lexeme`, but not the other way around. The name "lexeme" is not an amazing one terminology wise, but there is a historical precedent set by `parsec`.

Non-lexeme things A non-lexeme thing does not care about whitespace: these are raw tokens. It is highly likely that you wouldn't want to use these in a regular parser, but they may be handy for **custom error handling** or **building composite tokens**.

Lexeme things These do account for whitespace that occurs *after* a token, consuming everything up until the next token. This means there are some extra pieces of functionality available that don't make much sense for non-lexeme handling. The `lexeme` object can also be used as a function via its `apply` method, allowing it to make any parser into one that handles whitespace: this should be done for any composite tokens made with `nonlexeme`.

Whitespace handling should ideally be handled *uniformly* by `lexeme`: it establishes a convention of only consuming **trailing** whitespace, which is **important** for avoiding ambiguity in a parser. If you cannot use `lexeme.apply`, you *must* adhere to this same convention.

For handling initial whitespace in the parser (before the very first token), you should use `Lexer.fully`.

## Names and Symbols

These two categories of parser are closely linked, as described below.

### `Lexer.{lexeme, nonlexeme}.names`

This **object** contains the definitions of several different parsers for dealing with values that represent names in a language, specifically identifiers and operators. These are configured directly by `LexicalDesc.nameDesc`, however valid names are affected by the keywords and reserved operators as given in `LexicalDesc.symbolDesc`. Both are defined by an initial letter, and then any subsequent letters.

Note that **both** the start *and* end letters must be defined for an `identifier`/`userDefinedOperator` to work properly. It is not the case that, say, if `identifierStart` is ommited, that `identifierLetter` is used in its place.

In some cases, languages may have special descriptions of identifiers or operators that work in specific scenarios or with specific properties. For instance: Haskell's distinction between constructors, which start uppercase, and variables, which start lowercase; and Scala's special treatment of operators that end in `:`. In these cases, the `identifier` and `userDefinedOperator` parsers provided by `Names` allow you to refine the start letter (and optionally the end letter for operators) to restrict them to a smaller subset. This allows for these special cases to be handled directly.

### `Lexer.{lexeme, nonlexeme}.symbol`

Compared with `names`, which deals with user-defined identifiers and operators, `symbol` is responsible for hard-coded or reserved elements of a language. This includes keywords and built-in operators, as well as specific symbols like `{`, or `;`. The description for symbols, found in `LexicalDesc.symbolDesc`, describes what the "hard" keywords and operators are for the language: these are *always* regarded as reserved, and identifiers and user-defined operators may not take these names. However, the `symbol` object also defines

the `softKeyword` and `softOperator` combinators: these are for keywords that are *contextually* reserved. For example, in Scala 3, the soft keyword `opaque` is only considered a keyword if it appears before `type`; this means it is possible to define a variable `val opaque = 4` without issue. In `parsley`, this could be performed by writing `atomic(symbol.softKeyword("opaque") ~> symbol("type"))`. Keywords and reserved operators are only legal if they are not followed by something that would turn them into part of a wider identifier or user-defined operator: even if `if` is a keyword, `iffy` should not be parsed as `if` then `fy`!

Both soft and hard keywords cannot form part of a wider identifer. However, for this to work it is important that `NameDesc.identifierLetter` (and/or `NameDesc.operatorLetter`) is defined. If not, then `parsley` will not know what constitutes an illegal continutation of the symbol!

To make things easier, `symbol.apply(String)` can be used to take any literal symbol and handle it properly with respect to the configuration (except soft keywords need to go via `softKeyword`). If `if` is part of the `hardKeywords` set, then `symbol("if")` will properly parse it, disallowing `iffy`, and so on. If the provided string is not reserved in any way, it will be parsed literally, as if `string` had been used.

The `symbol` object also defines a bunch of pre-made helper parsers for some common symbols like `;`, `,`, and so on. They are just defined in terms of `symbol.apply(String)` or `symbol.apply(Char)`.

Implicits `symbol.implicits` contains the function `implicitSymbol`, which does the same job as `symbol.apply`, but is defined as an implicit conversion. By importing this, string literals can themselves serve as parsers of type `Parsley[Unit]`, and parse symbols correctly. With this, it instead of `symbol("if")` you can simply write `"if"`.

## `Lexer.{lexeme, nonlexeme}` Numeric Parsers

This **object** contains the definitions of several different parsers for handling *numeric* data: this includes both integers and floating point numbers. The configuration for all of these parsers is managed by `LexicalDesc.numericDesc`. The members are split into three kinds:

- `parsley.token.numeric.Integer`: values with this type deal with whole numbers, and this interface in particular has support for different bases of number as well as various bit-widths of parsed data. When the bit-width of the parser is restricted, the generated result can be any numeric type that is wide enough to accommodate those values. If the parsed int does not fit in the required bounds, a parse error will be generated. If no bit-width is specified, an arbitrary `BigInt` is used.

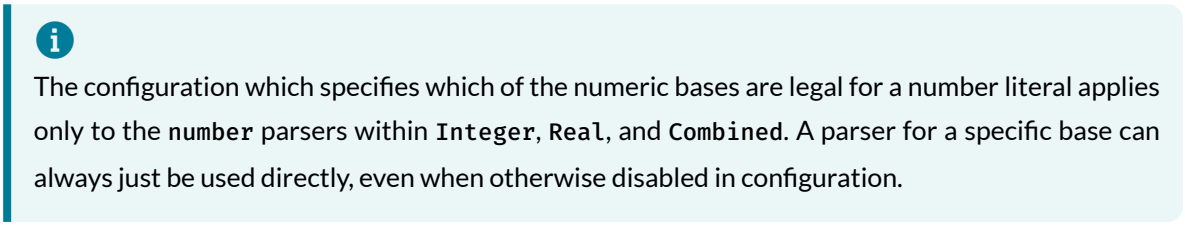
The supported bit-widths within `parsley` are 8, 16, 32, and 64. When one of these widths is chosen, the Scala compiler will automatically pick a result type that matches the same width (so `Int` for 32). If the generic parameter is filled manually, the given type will be used instead as long as it is wide enough.

Currently, there is no way of adding new bit-widths or defining custom numeric container types.

- `parsley.token.numeric.Real`: values with this type deal with floating-point numbers **only**: values without a point or an exponent (if allowed) will not be parsed by these parsers. Like `Integer`, different bases can be specified: in this case the meaning of exponents can be controlled within the configuration, for instance, a hexadecimal floating-point literal like `0xAp4` classically would represent `10 * 2^4`, or `160`, because `p` represents an exponent delimiter and hexadecimal exponents are normally base 2 (but this is fully configurable in `parsley`).

  Compared to `Integer`, different precisions can be chosen for `Real`, allowing for arbitrary-precision floats, `Float`, and `Double` results. For the stricter representations, there is a `doubleRounded`/`floatRounded` that just gives the nearest valid value (with no parse errors), and a `double`/`float` which demands that the parsed literal must at least be between the smallest and largest numbers of the type.

- `parsley.token.numeric.Combined`: values with this type can deal with both integers and floating-point numbers. This is done by returning one or the other as part of an `Either`. A *slightly* limited selection of bit-widths and precisions are available for both parts. The draw of these combinators is that they may remove the ambiguity between the two kinds of literal so that no backtracking is required within the parser.



The configuration which specifies which of the numeric bases are legal for a number literal applies only to the `number` parsers within `Integer`, `Real`, and `Combined`. A parser for a specific base can always just be used directly, even when otherwise disabled in configuration.

### Examples of Configuration and Valid Literals

The `plain` definition of `NumericDesc` provides a variety of different configurations for the numeric literals depending on the literal base, so it mostly suffices to look at the effects of these on the different bases to get a sense of what does what.

```
val lexer = new Lexer(LexicalDesc.plain)
```

The basic configuration allows `number` to work with hexadecimal and octal literals, as well as decimal. These have their standard prefixes of `0x` and `0o`, respectively (or uppercase variants). This means that `unsigned.number` will allow literals like `0`, `0xff`, `0o45` and `345`. For `signed`, each of these may be preceded by a `+` sign, but this is not *required*; if `positiveSign` is set to `PlusSignPresence.Compulsory`, positive literals would always require a `+`; and if it is set to `PlusSignPrecense.Illegal`, the `+` prefix can never be used (but `-` is fine regardless). By default, `023` is legal, but this can be disabled by setting `leadingZerosAllowed` to `false`.

```
val num = lexer.lexeme.signed.number
// num: parsley.Parsley[BigInt] = parsley.Parsley@4dd0fe2d
num.parse("0")
// res0: parsley.Result[String, BigInt] = Success(x = 0)
num.parse("0xff")
// res1: parsley.Result[String, BigInt] = Success(x = 255)
num.parse("+0o45")
// res2: parsley.Result[String, BigInt] = Success(x = 37)
num.parse("-345")
// res3: parsley.Result[String, BigInt] = Success(x = -345)
```

In the basic configuration, break characters are not supported. However, by setting `literalBreakChar` to `BreakCharDesc.Supported('_', allowedAfterNonDecimalPrefix = true)`, say, will allow for `1_000` or `0x_400`. Setting the second parameter to `false` will forbid the latter example, as the break characters may then only appear between digits.

```
val lexerWithBreak = new Lexer(LexicalDesc.plain.copy(
        numericDesc = NumericDesc.plain.copy(
            literalBreakChar
 = BreakCharDesc.Supported('_', allowedAfterNonDecimalPrefix = true))
    ))
// lexerWithBreak: Lexer = parsley.token.Lexer@4731c284
val withBreak = lexerWithBreak.lexeme.signed.number
// withBreak: parsley.Parsley[BigInt] = parsley.Parsley@7248ea83
withBreak.parse("1_000")
// res4: parsley.Result[String, BigInt] = Success(x = 1000)
withBreak.parse("1_")
// res5: parsley.Result[String, BigInt] = Failure(
// ...
withBreak.parse("2__0") // no double break
// res6: parsley.Result[String, BigInt] = Failure(
// ...
```

Real numbers in the default configuration do not support literals like `.0` or `1.`, this behaviour must be explicitly enabled with `trailingDotAllowed` and `leadingDotAllowed`: note that `.` is not a valid literal, even with both flags enabled! By default, all four different bases support exponents on their literals for

floating-point numbers. This could be turned off for each by using `ExponentDesc.NoExponents`. However, with exponents enabled, it is configured that the non-decimal bases all *require* exponents for valid literals. Whilst `3.142` is valid decimal literal, `0x3.142` is not a legal hexadecimal literal: to make it work, the exponent must be added, i.e. `0x3.142p0`, where `p0` is performing `* 2^0`. For each of the non-decimal literals, the base of the exponent is configured to be `2`, hence `2^0` in the previous example; for decimal it is set to the usual `10`, so that `2e3` is `2*10^3`, or `2000`. Notice that literals do not *require* a point, so long as they do have an exponent.

```
val real = lexer.lexeme.real
// real: parsley.token.numeric.Real = parsley.token.numeric.LexemeReal@147e4580
real.hexadecimalDouble.parse("0x3.142")
// res7: parsley.Result[String, Double] = Failure(
// ...
real.hexadecimalDouble.parse("0x3.142p0")
// res8: parsley.Result[String, Double] = Success(x = 3.07861328125)
real.binary.parse("0b0.1011p0")
// res9: parsley.Result[String, BigDecimal] = Success(x = 0.6875)
real.decimal.parse("3.142")
// res10: parsley.Result[String, BigDecimal] = Success(x = 3.142)
real.decimal.parse("4")
// res11: parsley.Result[String, BigDecimal] = Failure(
// ...
real.decimal.parse("2e3")
// res12: parsley.Result[String, BigDecimal] = Success(x = 2000)
```

When a floating point literal is parsed in a non-decimal base, the meaning of each digit past the point is to be a fraction of that base. The example `0x3.142p0`, for instance is not equal to the decimal `3.142`. Instead, it is equal to `(3 + 1/16 + 4/(16^2) + 2/(16^3)) * 2^0 = 3.07861328125`. Handily, hexadecimal floats are still equal to the 4-bit bunching up of binary floats: `0x0.Bp0` is the same as `0b0.1011p0`, both of which are `0.6875` in decimal.

## `Lexer.{lexeme, nonlexeme}` Text Parsers

This **object** deals with the parsing of both string literals and character literals, configured broadby by `LexicalDesc.textDesc`:

- `parsley.token.text.String`: values with this type deal with multi-character/codepoint strings. Specifically, the interface provides ways of dealing with different levels of character encodings.

  In practice, there are four values with this type: `text.string`, `text.rawString`, `text.multiString`, and `text.rawMultiString` covering three different kinds of string:

- `text.string`: used to handle regular string literals where escape characters are present. They are single-line.
- `text.multiString`/`text.rawMultiString`: handles string literals using the `multiStringEnds` configuration within `LexicalDesc.textDesc`. These string literals can span multiple lines.
- `text.rawString`/`text.rawMultiString`: handles string literals that do not have escape characters.

> Note that currently, whatever a string literal is started with, it must end with the exact same sequence.

- `parsley.token.text.Character`: like `String`, the `text.character` object can handle different kinds of text encoding. However, unlike `String`, there is only one kind of character available, which has escape codes and can only contain a single graphic character (or, if applicable, single unicode codepoint).

> A single codepoint here refers to having at most two 16-bit UTF-16 characters in a surrogate pair, allowing for any character in the range `0x00000` to `0x10ffff`. Some unicode characters are composed of multiple codepoints. As an example, national flags are composed of two "regional indicator characters", for instance # and #, making ##. Such emoji cannot appear within a parsed character in `parsley`, and can instead only be written in a string.

### Examples of Configuration and Valid Literals

The majority of configuration for strings and characters is focused around the escape sequences. Outside of that, it is mostly just what the valid start and end sequences are valid for different flavours of literal. However, the `graphicCharacter` predicate is used to denote what the valid characters are that can appear in a string verbatim. This can be restricted to a smaller set than might otherwise have been checked by `ascii` or `latin1` parsers. In these instances, a different error message would be generated:

```
val aboveSpace = predicate.Unicode(_ >= 0x20)
// aboveSpace: predicate.Unicode = Unicode(<function1>)
def stringParsers(graphicChar: CharPredicate = aboveSpace,
                  escapeDesc: EscapeDesc = EscapeDesc.plain) =
    new Lexer(LexicalDesc.plain.copy(
        textDesc = TextDesc.plain.copy(
            escapeSequences = escapeDesc,
            graphicCharacter = graphicChar
        )
```

```
    )).nonlexeme.string

val fullUnicode = stringParsers(aboveSpace)
// fullUnicode: parsley.token.text.String =
 parsley.token.text.ConcreteString@50383c91
val latin1Limited = stringParsers(predicate.Basic(c => c >= 0x20 && c <= 0xcf))
// latin1Limited: parsley.token.text.String =
 parsley.token.text.ConcreteString@c2a7678

fullUnicode.latin1.parse("\"hello α\"")
// res13: parsley.Result[String, String] = Failure((line 1, column 2):
//   non-latin1 characters in string literal, this is not allowed
//   >"hello α"
//     ^^^^^^^)
latin1Limited.fullUtf16.parse("\"hello α\"")
// res14: parsley.Result[String, String] = Failure((line 1, column 8):
//   unexpected "α"
//   expected """ or string character
//   >"hello α"
//           ^)
```

When it comes to escape characters, the **configuration** distinguishes between four kinds of escape sequence, which are further sub-divided:

Denotative escapes These are a family of escape sequences that are names or symbols for the escape characters they represent. Parsley supports three different kinds of denotative escape characters:

- `EscapeDesc.literals`: these are a set of characters that plainly represent themselves, but for whatever reason must be escaped to appear within the string. Some of the most common examples would be `\"` or `\\`, which are an escaped double quote and backslash, respectively. The literal set for these would be `Set('"', '\\')`
- `EscapeDesc.singleMap`: these are a mapping of specific single characters to the underlying characters they represent. Commonly, this might be something like `\n`, which would be represented in the map by an entry `'n' -> 0xa`.
- `EscapeDesc.multiMap`: these generalise the single map by allowing a multiple character sequence to represent a specific escape character. These are less common in "the wild", but a good example is Haskell, where `'\NULL'` is a valid character, represented by an entry `"NULL" -> 0x0` in the `multiMap`.

Of course, all denotative escape sequences can be represented by the `multiMap` on its own, and all the above examples could be represented by `Map("\"" -> '"', "\\" -> '\\', "n" -> 0xa, "NULL" -> 0x0)`. For `literals` in particular, the `Set` is more ergonomic than the `Map`.

Note that the `literals` set, along with the keys of `singleMap` and `multiMap`, must all be distinct from each other. Furthermore, no empty sequences may be placed in `multiMap`. Violating any of these requirements will result in an error.

Numeric escapes These are escapes that represent the numeric code of a specific character. There are four different bases for numeric escapes: binary, octal, hexadecimal, and decimal. Each of these can have their own unique prefix (or lack there of), maximum allowed value, and specific number of digits:

- `NumberOfDigits.Unbounded`: simply, this allows the numeric escape to have any number of digits, so long as the end result is within the specified maximum value of the escape.
- `NumberOfDigits.AtMost(n: Int)`: this denotes that there is an upper-limit to the number of digits allowed for the escape sequence, but it can take any number of digits below this limit. Again, the end result must still be within the specified maximum value of the escape.
- `NumerOfDigits.Exactly(ns: Int*)`: this denotes that the number of digits can be one of the specified totals in `ns` (there must be at least one provided number). For example `Exactly(1, 2, 4, 6)` would allow escapes like `\0`, `\20`, `\0400`, `\10fffe` are legal, but `\400` would not be.

String gaps Supported for string literals only, string gaps allow for prunable whitespace within a string literal. These take the form of a backslash, followed by whitespace, terminated by another backslash (this can include newlines, even in otherwise single-line strings). As an example:

```
val withGaps = stringParsers(escapeDesc = EscapeDesc.plain.copy(gapsSupported
 = true))
// withGaps: parsley.token.text.String =
 parsley.token.text.ConcreteString@6474aa8c
withGaps.ascii.parse("""Hello \

      \World!" """)
// res15: parsley.Result[String, String] = Success(Hello World!)
```

Empty escapes These are also only supported by string literals. These characters have no effect on the string literal, but otherwise allow for disambiguation with multi-character escape sequences. For example, if `EscapeDesc.emptyEscape` is set to `Some('&')`, then `"\x20\&7"` would be interpreted as the string `" 7"`, however, without the `\&` character, it would try and render character `0x207`.

## `Lexer.lexeme.{enclosing, separators}`

These two objects just contain various shortcuts for doing things such as semi-colon separated things, or braces enclosed things, etc. There is nothing special about them: with

`lexer.lexeme.symbol.implicits.implicitSymbol` imported, `"(" ~> p <~ ")"` is the same as `lexer.lexeme.enclosing.parens(p)`. The choice of one style over the other is purely up to taste.

## Whitespace-Sensitive Languages and `Lexer.space`

Normally, the whitespace definitions used by `lexeme` are fixed as described by the `LexicalDesc.spaceDesc`; accounting for the comments and spaces themselves. However, some languages, like Python and Haskell do not have constant definitions of whitespace: for instance, inside a pair of parentheses, newline characters are no longer considered for the current indentation. To support this, `parsley` allows for the space definition to be locally altered during parsing if `LexicalDesc.spaceDesc.whitespaceIsContextDependent` is set to `true`: this *may* impact the performance of the parser.

> ⊗
> If the `LexicalDesc.spaceDesc.whitespaceIsContextDependent` flag is turned on it is **crucial** that either the `Lexer.fully` combinator is used, *or* `Lexer.space.init` is ran as the very first thing the top-level parser does. Without this, the context-dependent whitespace will not be set-up correctly!

In this mode, it is possible to use the `lexer.space.alter` combinator to *temporarily* change the definition of whitespace (but not comments) within the scope of a given parser. As an example:

```
val withNewline = predicate.Basic(_.isSpace)
val expr = ... | "(" ~> lexer.space.alter(withNewline)(expr) <~ ")"
```

For the duration of that nested `expr` call, newlines are considered regular whitespace. This, of course, is assuming that newlines were *not* considered whitespace under normal conditions.

# Configuring the Lexer (`parsley.token.descriptions`)

The `Lexer` is configured primarily by providing it a `LexicalDesc`. This is a structure built up of many substructures that each configure a specific part of the overall functionality available. In general, many parts of this hierarchy have "sensible defaults" in the form of their `plain` value within their companion objects; these document what choices were made in each individual case. There may also be some values crafted to adhere to some specific language specification; for instance, `EscapeDesc.haskell` describes escape characters that adhere to the Haskell Report.

This page does not aim to document everything that is configurable within `LexicalDesc`, but it will outline the general design and how things slot together.

## Diagram of Dependencies

The hierarchy of types involved with lexical configuration can be daunting. The following diagram illustrates both the "has-a" and "is-a" relationships between the types. For instance, `TextDesc` contains an `EscapeDesc`, and `NumericEscape` may be implemented by either `NumericEscape.Illegal` or `NumericEscape.Supported`.

*In the above diagram, an _ represents a* `.`

The types in the diagram that have alternative implements are as follows:

- `BreakDescChar`: used to describe whether or not numeric literals can contain meaningless "break characters", like `_`. It can either be `NoBreakChar`, which disallows them; or `Supported`, which will specify the character and whether it is legal to appear after a non-decimal prefix like hexadecimal `0x`.
- `PlusSignPresence`: used to describe whether or not a `+` is allowed in numeric literals, which appears for the start of numeric literals and floating-point exponents. It can either be `Required`, which means either a `+` or `-` must always be written; `Optional`, which means a `+` can be written; or `Illegal`, which means only a `-` can appear.
- `ExponentDesc`: used to describe how an exponent is formed for different bases of floating point literals. It can either be `Supported`, in which case it will indicate whether it is compulsory, what characters can start it, what the numeric base of the exponent number itself is, and then what the `PlusSignPresence` is, as above; otherwise, it is `NoExponents`, which means that the exponent notation is not supported for a specific numeric base.

- `NumericEscape`: used to describe whether or not numeric escape sequences are allowed in string and character literals. It either be `Illegal`, which means there are no numeric escapes; or `Supported`, which means that the prefix, `NumberOfDigits`, and the maximum value of the escape must all be specified.
- `NumberOfDigits`: used by the above `NumericEscape` to determine how many digits can appear within a numeric escape literal. These can be one of: `Unbounded`, which means there can be any well-formed number as the escape; `AtMost`, which puts an upper limit on the number of digits that can appear; or `Exactly`, which details one or more exact numbers of digits that could appear, for instance, some languages allow for numeric escapes with exactly 2, 4, or 6 digits in them only.

# Configuring Errors (`parsley.token.errors`)

The default error messages generated by the parsers produced by the `Lexer` are *ok*, but can be much improved.

## `errors.ErrorConfig`

The `ErrorConfig` class is where all the configuration for error messages generated by the `Lexer` resides. Everything in this class will have a default implementation (nothing is abstract); this ensures easy backwards compatibility. Each of the configurations inside takes one of the following forms:

- A plain `String` argument, usually indicating a name of a compulsory label.
- A `LabelConfig`, which can either be unconfigured, hidden, or a regular label name.
- A `LabelWithExplainConfig`, which augments the previous configuration to also allow for a reason to be added, if desired.
- A `FilterConfig`, or one of its specific subtypes, which can be used to handle the messages for ill-conforming data.
- A **special configuration**, which is used for very specific error messages, usually arising from one of the more advanced error patterns (see **Advanced Error Messages**)

## Configuring Labels and Explains

Labels are one of the most common additional error configurations that can be applied throughout the pre-made lexer parsers. Some, but not all, of these labels can be configured to also produce a *reason* if the configuree cannot be parsed (either for why it should be there or what it requires). The hierarchy of components is visualised by the following UML diagram:

Broadly, a component may either be marked as a `LabelWithExplainConfig`, which means it can contain either labels, reasons, or both; `LabelConfig` if a reason wouldn't make sense; and a `ReasonConfig` if it does not make sense to name.

### Configuring Labels

Adding a label can be one of the following:

- `Label`: this labels the corresponding parser with one or more labels -- this also applies for `LabelAndReason`.

- `Hidden`: this suppresses any error messages arising from the corresponding parser.
- `NotConfigured`: this doesn't alter the error messages from the corresponding parser.

### Adding Explanations

Adding an explanation can be one of the following:

- `Reason`: this adds a reason for the corresponding parser though doesn't change the labelling -- unless `LabelAndReason` is used instead.
- `NotConfigured`: this doesn't alter the error messages from the corresponding parser.

## Configuring Filtering

Some parsers perform filtering on their results, for instance checking if a numeric literal is within a certain bit-width. The messages generated when these filters fail is controled by the `FilterConfig[A]`, where `A` is the type of value being filtered. The below diagram shows how the various sub-configurations are laid out.

Some filters within the `Lexer` are best left as a *specialised* or *vanilla* error, which is why the hierarchy is constrained. Other than that, the various leaf classes allow for various combinations of adding reasons, altering the unexpected message, or bespoke error messages. The `BasicFilter` here does not attach any special error messages to the filtering, having the effect of just using the basic `filter` combinator internally.

## Special Configuration

Some parts of the error configuration for the `Lexer` are special. In particular, these are `preventRealDoubleDroppedZero` and the two `verifiedXBadCharsUsedInLiteral` (for both `Char` and `String`). These provide very hand-crafted error messages for specific scenarios, based on the ideas of the *Preventative* and *Verified Errors* patterns.

### Preventing Double-Dropped Zero

When writing floating point literals, it is, depending on the configuration, possible to write `.0`, say, or `0.`. However, it should not be possible to have the literal `.` on its own! Overriding `preventRealDoubleDroppedZero` is the way to prevent this, and provide a good error message in the process. There are a few options:

- `UnexpectedZeroDot`: sets an unexpected message when just `.` is seen to be the given string.
- `ZeroDotReason`: does not set an unexpected message, but adds a reason explaining why `.` is illegal.

- `UnexpectedZeroDotWithReason`: combines both above behaviours.
- `ZeroDotFail`: throws an error with the given bespoke error messages.

### Preventing Bad Characters in Literals

When writing string and character literals, some characters may be considered illegal. For instance, a langauge may not allow `"` to appear unescaped within a character literal. To help make it clear why a character was rejected by the parser, `verifiedCharBadCharsUsedInLiteral` and `verifiedStringBadCharsUsedInLiteral` allow for fine-grained error messages to be generated when an illegal character occurs. There are a few options:

- `BadCharsFail` takes a `Map[Int, Seq[String]]` from unicode characters to the messages to generate if one of the keys was found in the string.
- `BadCharsReason` takes a `Map[Int, String]` from unicode characters to the reason they generate if they are found in the string.
- `Unverified` does no additional checks for bad characters.

# Error Message Combinators

Aside from the failures generated by character consumption, `parsley` has many combinators for both generating failures unconditionally, as well as augenting existing errors with more information. These are found within the `parsley.errors.combinator` module.

> *The Scaladoc for this page can be found at* `parsley.errors.combinator`.

## Failure Combinators

Normally, failures can be generated by `empty`, `satisfy`, `string`, and `notFollowedBy`; as well as their derivatives. However, those do not capture the full variety of "unexpected" parts of error messages. In the below table, `empty` corresponds to `empty(0)` (these are both found in `parsley.Parsley`). The *named* items are produced by `unexpected` combinators, and wider carets of *empty* items can be obtained by passing wider values to `empty`. This is summarised in the table below.

| Caret | *empty* | *raw/eof* | *named* |
|---|---|---|---|
| `0` | `empty(0)` | n/a | `unexpected(0, _)` |
| `1` | `empty(1)` | `satisfy` | `unexpected(1, _)` |
| `n` | `empty(n)` | `string` | `unexpected(n, _)` |

### The `unexpected` Combinator

The `unexpected` combinator fails immediately, but produces a given name as the unexpected component of the error message with a caret as wide as the given integer. For instance:

```
import parsley.character.char
import parsley.errors.combinator.unexpected

unexpected(3, "foo").parse("abcd")
// res0: parsley.Result[String, Nothing] = Failure((line 1, column 1):
//   unexpected foo
//   >abcd
//    ^^^)
(char('a') | unexpected("not an a")).parse("baa")
// res1: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected not an a
```

```
//   expected "a"
//   >baa
//    ^)
```

There are a few things to note about the above examples:

- Just using `unexpected` alone does not introduce any other components, like expected items, to the error
- When the caret width is unspecified, it will adapt to whatever the caret would have been for the error message
- The *named* items resulting from the combinator *dominate* other kinds of item, so that `char('a')`'s natural "unexpected 'a'" disappears

## The `fail` Combinator

In contrast to the `unexpected` combinator, which produces *vanilla* errors, the `fail` combinator produces *specialised* errors, which suppress all other components of an error in favour of some specific messages.

```
import parsley.character.string
import parsley.errors.combinator.fail

fail(2, "msg1", "msg2", "msg3").parse("abc")
// res2: parsley.Result[String, Nothing] = Failure((line 1, column 1):
//   msg1
//   msg2
//   msg3
//   >abc
//    ^^)
(fail(1, "msg1") | fail(2, "msg2") | fail("msg3")).parse("abc")
// res3: parsley.Result[String, Nothing] = Failure((line 1, column 1):
//   msg1
//   msg2
//   msg3
//   >abc
//    ^^)
(fail("msg") | string("abc")).parse("xyz")
// res4: parsley.Result[String, String] = Failure((line 1, column 1):
//   msg
//   >xyz
//    ^^^)
(fail(1, "msg") | string("abc")).parse("xyz")
// res5: parsley.Result[String, String] = Failure((line 1, column 1):
//   msg
//   >xyz
//    ^)
```

Notice that if a caret width is specified, it will override any other carets from other combinators, like `string`. Not specifying a caret is adaptive. The `fail` combinator also suppressed other error messages, and merges within itself as if all the messages were generated by one `fail`.

## Error Enrichment

Other than the freestanding combinators, some combinators are enabled by importing `parsley.errors.combinator.ErrorMethods`. Some of these are involved with augmenting error messages with additional information. These are discussed below.

None of the combinators in this section have any effect on `fail` or its derivatives.

### The `label` Combinator

When combinators that read characters fail, they produce "expected" components in error messages:

```
import parsley.character.{char, string, satisfy}

char('a').parse("b")
// res6: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected "b"
//   expected "a"
//   >b
//    ^)
string("abc").parse("xyz")
// res7: parsley.Result[String, String] = Failure((line 1, column 1):
//   unexpected "xyz"
//   expected "abc"
//   >xyz
//    ^^^)
satisfy(_.isDigit).parse("a")
// res8: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected "a"
//   >a
//    ^)
```

Notice that the `satisfy` combinator cannot produce an expected item because nothing is known about the function passed in. The other two produce *raw* expected items. The `label` combinator can be used to replace these and generate *named* items. This is employed by `parsley.character` for its more specific parsers:

```
import parsley.errors.combinator.ErrorMethods

val digit = satisfy(_.isDigit).label("digit")
// digit: parsley.Parsley[Char] = parsley.Parsley@128f792a
digit.parse("a")
// res9: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected "a"
//   expected digit
//   >a
//    ^)
```

The `label` combinator above has added the label `digit` to the parser. If there was an existing label there, it would have been replaced.

A `label` combinator cannot be provided with `""`. In other libraries, this may represent hiding, however in `parsley`, the `hide` combinator is distinct.

A `label` combinator, along with other combinators, only applies if the error message properly lines up with the point the input was at when it entered the combinator - otherwise, the label may be inaccurate. For example:

```
val twoDigits = (digit *> digit).label("two digits")
// twoDigits: parsley.Parsley[Char] = parsley.Parsley@257c37c1
twoDigits.parse("a")
// res10: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected "a"
//   expected two digits
//   >a
//    ^)
twoDigits.parse("1a")
// res11: parsley.Result[String, Char] = Failure((line 1, column 2):
//   unexpected "a"
//   expected digit
//   >1a
//     ^)
```

### The `explain` Combinator

The `explain` combinator allows for the addition of further lines of error message, providing more high-level reasons for the error or explanations about a syntactic construct. It behaves similarly to `label` in that it will only apply when the position of the error message matches the offset that the combinator entered at.

```
import parsley.errors.combinator.ErrorMethods

digit.explain("a digit is needed, for some reason").parse("a")
// res12: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected "a"
//   expected digit
//   a digit is needed, for some reason
//   >a
//    ^)
```

A `explain` combinator cannot be provided with `""`.

### The `hide` Combinator

Sometimes, a parser should not appear in an error message. A good example is whitespace, which is *almost* never the solution to any parsing problem, and would otherwise distract from rest of the error content. The `hide` combinator can be used to suppress a parser from appearing in the rest of a message:

```
import parsley.errors.combinator.ErrorMethods

(char('a') | digit.hide).parse("b")
// res13: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected "b"
//   expected "a"
//   >b
//    ^)
```

## Error Adjustment Combinators

The previous combinators in this page have been geared at adding additional richer information to the parse errors. However, these combinators are used to adjust the existing information, mostly relating to position, to ensure the error remains specific.

### The `amend` Combinator

The `amend` combinator can adjust the position of an error message so that it occurs at an earlier position. This means that it can be affected by other combinators like `label` and `explain`. This is a precision tool, designed for fine-tuning error messages.

```
import parsley.errors.combinator.amend

amend(digit *> char('a')).parse("9b")
// res14: parsley.Result[String, Char] = Failure((line 1, column 1):
//   unexpected "9"
//   expected "a"
//   >9b
//    ^)
```

Notice that the above error makes no sense. This is why `amend` is a precision tool: it should ideally be used in conjunction with other combinators. For instance:

```
import parsley.syntax.character.charLift
import parsley.combinator.choice
import parsley.character.{noneOf, stringOfMany}

val escapeChar = choice('n'.as('\n'), 't'.as('\t'), '\"', '\\')
val strLetter =
    noneOf('\"', '\\').label("string char") | ('\\'
 ~> escapeChar).label("escape char")
val strLit = '\"' ~> stringOfMany(strLetter) <~ '\"'
```

```
strLit.parse("\"\\b\"")
// res15: parsley.Result[String, String] = Failure((line 1, column 3):
//   unexpected "b"
//   expected """, "\", "n", or "t"
//   >"\b"
//      ^)
```

In the above error, it is not *entirely* clear why the presented characters are expected. Perhaps it would be better to highlight a correct escape character instead? The `amend` combinator can be used in this case to pull the error back and rectify it:

```
val strLetter = noneOf('\"', '\\').label("string char") |
                amend('\\' ~> escapeChar).label("escape char")
```

```
strLit.parse("\"\\b\"")
// res16: parsley.Result[String, String] = Failure((line 1, column 2):
//   unexpected "\"
//   expected escape char or string char
//   >"\b"
//     ^)
```

While the `amend` has pulled the error back, and thanks to the `label` the error is still sensible, it could be improved by widening the caret and providing an explanation:

```
import parsley.Parsley.empty
val escapeChar = choice('n'.as('\n'), 't'.as('\t'), '\"', '\\') | empty(2)
val strLetter = noneOf('\"', '\\').label("string char") |
                amend('\\' ~> escapeChar)
                  .label("escape char")
                  .explain("escape characters are \\n, \\t, \\\", or \\\\")
```

```
strLit.parse("\"\\b\"")
// res17: parsley.Result[String, String] = Failure((line 1, column 2):
//   unexpected "\b"
//   expected escape char or string char
//   escape characters are \n, \t, \", or \\
//   >"\b"
//     ^^)
```

Note, an `unexpected` could also have been used instead of `empty` to good effect.

### The `entrench` and `dislodge` Combinators

The `amend` combinator will indiscriminately adjust error messages so thay they occur earlier. However, sometimes only errors from some parts of a parser should be repositioned. The `entrench` combinator protects errors from within its scope from being amended, and `dislodge` undoes that protection.

This can be useful if you want an error to be able to *dominate* another one, and then be amended afterwards, without affecting the original error. This normally has the following pattern:

```
val p = amendThenDislodge(1) {
    entrench(q) | r
}
```

In this example, we believe that `r` will produce errors deeper than `q`s, but after it discards `q`s message should be reset to an earier point. On the other hand, `q` is protected from the initial amendment, but then is free to be amended again after the `dislodge` has removed the protection.

### The `markAsToken` Combinator

The `markAsToken` combinator will assign the "lexical" property to any error messages that happen within its scope at a *deeper* position than the combinator began at. This is fed forward onto the `unexpectedToken` method of the `ErrorBuilder`: more about this in **lexical extraction**.

# Error Message Patterns

The combinators discussed in `parsley.errors.combinator` are useful for changing the contents of an error message, but they do not account for any contextual obligations a more carefully crafted error might have. Ensuring that error messages are valid in the context in which they occurred is the job of the *Verified Errors* and *Preventative Errors* parsing design patterns. These are both encoded in `parsley` as combinators within `parsley.errors.patterns`.

> *The Scaladoc for this page can be found at* `parsley.errors.patterns`.

## *Verified Errors*

When a hand-tailored reason for a syntax error relies on some future input being present, a *verified* error can be used to ensure these obligations are met. These errors, called *error widgets*, will have the following properties:

- have type `Parsley[Nothing]`, ensuring that it is guaranteed to fail
- produce errors that are rooted at the start of the widget
- generate a caret as wide as the problematic input
- if the problematic parser succeeds having consumed input, the widget consumes input
- if the problematic parser fails, it does not consume input or influence the error message of the surrounding parser in any way

The "problematic parser" referred to above is what will be parsed to try and verify that the requirements for the error messages are met. Any widget that meets these five properties is likely already a *verified error*, however, they can be satisfied by the combinators enabled by importing `parsley.errors.patterns.VerifiedErrors` -- this is the focus of this page.

Each of the `verifiedX` combinators try a parse something, and if it succeeds will generate an error based on the result. If it could not be parsed, they will generate an empty error. Each different combinator will generate an error with different content. As examples (in isolation of surrounding content):

```
import parsley.errors.patterns.VerifiedErrors
import parsley.character.char

// assume that a `lexer` is available
val float = lexer.nonlexeme.floating.decimal
// float: parsley.Parsley[BigDecimal] = parsley.Parsley@66f6199
val _noFloat =
```

```
    float.verifiedExplain("floating-point values may not be used as array
 indices")
// _noFloat: parsley.Parsley[Nothing] = parsley.Parsley@2b7aaedb

_noFloat.parse("hello")
// res0: parsley.Result[String, Nothing] = Failure((line 1, column 1):
//   unknown parse error
//   >hello
//    )
_noFloat.parse("3.142")
// res1: parsley.Result[String, Nothing] = Failure((line 1, column 1):
//   unexpected "3.142"
//   floating-point values may not be used as array indices
//   >3.142
//    ^^^^^)

val int = lexer.nonlexeme.unsigned.decimal
// int: parsley.Parsley[BigInt] = parsley.Parsley@515ec88e
val _noPlus = (char('+') ~> int).verifiedFail { n =>
    Seq(s"the number $n may not be preceeded by \"+\"")
}
// _noPlus: parsley.Parsley[Nothing] = parsley.Parsley@b9381b1
_noPlus.parse("+10")
// res2: parsley.Result[String, Nothing] = Failure((line 1, column 1):
//   the number 10 may not be preceeded by "+"
//   >+10
//    ^^^)
```

Occasionally, there may need to be more fine-grained control over the errors. In these cases, the most generic version of the combinator is `verifiedWith`, which takes an `ErrorGen` object. For instance, perhaps the focus of the last error above should only be the `+`. In which case, the caret must be adjusted by an `ErrorGen` object -- this can either be `VanillaGen` or `SpecialisedGen`.

```
import parsley.errors.SpecialisedGen
val _noPlus = (char('+') ~> int).verifiedWith {
    new SpecialisedGen[BigInt] {
        def messages(n: BigInt): Seq[String] =
            Seq("a number may not be preceeded by \"+\"")
        override def adjustWidth(x: BigInt, width: Int) = 1
    }
}
// _noPlus: parsley.Parsley[Nothing] = parsley.Parsley@6bc1d375
_noPlus.parse("+10")
// res3: parsley.Result[String, Nothing] = Failure((line 1, column 1):
//   a number may not be preceeded by "+"
//   >+10
//    ^)
```

In the above, the `width` parameter would have been the original determined size, which would have been 3 based on the other error message. A `VanillaGen` would also have the ability to change the unexpected message:

```
import parsley.errors.VanillaGen
val _noFloat = float.verifiedWith {
    new VanillaGen[BigDecimal] {
        override def reason(x: BigDecimal): Option[String] =
            Some("floats may not be array indices")
        override def unexpected(x: BigDecimal): VanillaGen.UnexpectedItem = {
            VanillaGen.NamedItem("floating-point number")
        }
    }
}
// _noFloat: parsley.Parsley[Nothing] = parsley.Parsley@54c2c2ec

_noFloat.parse("3.142")
// res4: parsley.Result[String, Nothing] = Failure((line 1, column 1):
//   unexpected floating-point number
//   floats may not be array indices
//   >3.142
//    ^^^^^)
```

## *Preventative Errors*

The *verified error* pattern can only be applied as part of a chain of alternatives. However, if the alternatives are spread far apart, this can make the pattern cumbersome to use. Instead, a *preventative error* seeks to rule out bad inputs not as a last resort, but eagerly as soon as it might become possible to do so. Widgets following this pattern have the following properties:

- succeed if the problematic parser does not succeed, returning `Unit`
- produce errors that are unconditionally rooted at the start of the widget
- generate a caret as wide as the problematic input
- if the problematic parser succeeds having consumed input, this widget must consume input
- if the problematic parser fails, it should not consume input nor influence any errors generated afterwards

Similarly to *verified errors*, *preventative errors* can have many forms but the most common are embodied by the combinators available by importing `parsley.errors.patterns.PreventativeErrors`.

Unlike, `verifiedX`, `preventativeX` will try and parse something, and succeed if that fails. This makes it *similar* to `notFollowedBy`, but that alone does not have all the desired properties. As an example:

```
import parsley.errors.patterns.PreventativeErrors
```

```
val ident = lexer.nonlexeme.names.identifier
// ident: parsley.Parsley[String] = parsley.Parsley@4b0602a8
val _noDot = (char('.') ~> ident).preventativeFail { v =>
    Seq(s"accessing field $v is not permitted here")
}
// _noDot: parsley.Parsley[Unit] = parsley.Parsley@71e977a7
_noDot.parse("hi")
// res5: parsley.Result[String, Unit] = Success(())
_noDot.parse(".foo")
// res6: parsley.Result[String, Unit] = Failure((line 1, column 1):
//   accessing field foo is not permitted here
//   >.foo
//    ^^^^)
```

There are also vanilla variants too (and one for `ErrorGen`). However, these also allow for additional optional labels to be provided to describe *valid* alternatives that would have been successful from this point. This is useful, since parsers that follow from this point will not be parsed and cannot contribute their own labels. It is for this reason, that the *verified error* pattern is more effective if it possible to use.

# Constructing Custom Errors

By default, `parsley` returns errors that consist of `String`-based content. However, it is possible to build error messages into a datatype or format that is user-defined. This is done with the `ErrorBuilder` typeclass.

The `ErrorBuilder` is pulled in implicitly by the `parse` method of the `Parsley` type:

```
class Parsley[A] {
    def parse[Err: ErrorBuilder](input: String): Result[Err, A]
}
```

This is equivalent to having an implicit parameter of type `ErrorBuilder[Err]`. As the `ErrorBuilder` companion object has an implicit value of type `ErrorBuilder[String]` only, the type `String` is chosen as the default instantiation of `Err` by Scala. Providing another `ErrorBuilder` implicit object in a tighter scope (or adding an explicit type ascription with another implicit object available), you are able to hook in your own type instead.

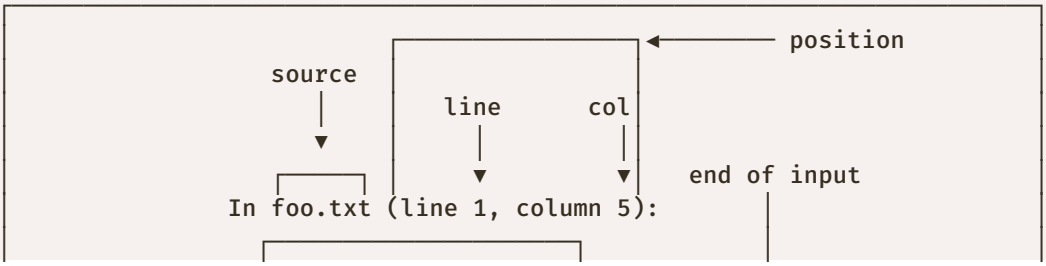
This page describes how the `ErrorBuilder` is structured, and gives an example of how to construct a lossy type suitable for unit testing generated error messages.

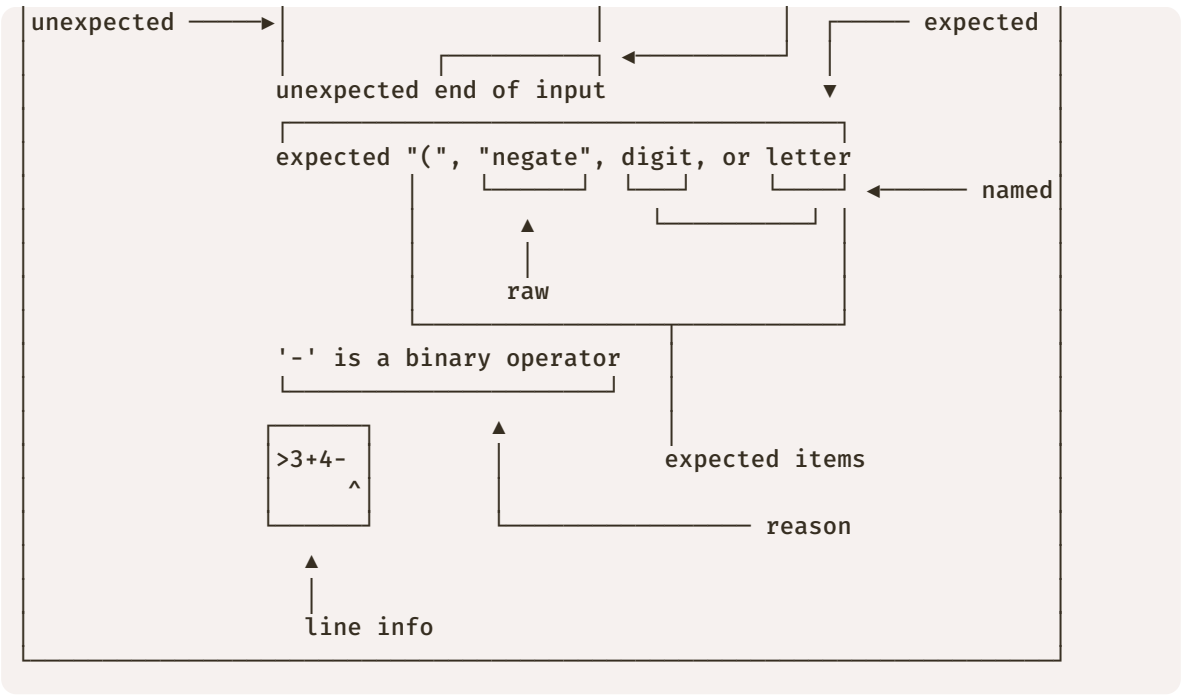
> *The Scaladoc for this page can be found at* `parsley.errors.ErrorBuilder`.

## Error Message Structure

Error messages within `parsley` take two different forms: *vanilla* or *specialised*. The error chosen depends on the combinators used to produce it: `empty`, `unexpected`, `char`, `string`, etc all produce vanilla errors; and `fail` and its derivatives produce specialised errors. An `ErrorBuilder` must describe how to format both kinds of error; their structure is explained below.

### Vanilla Errors

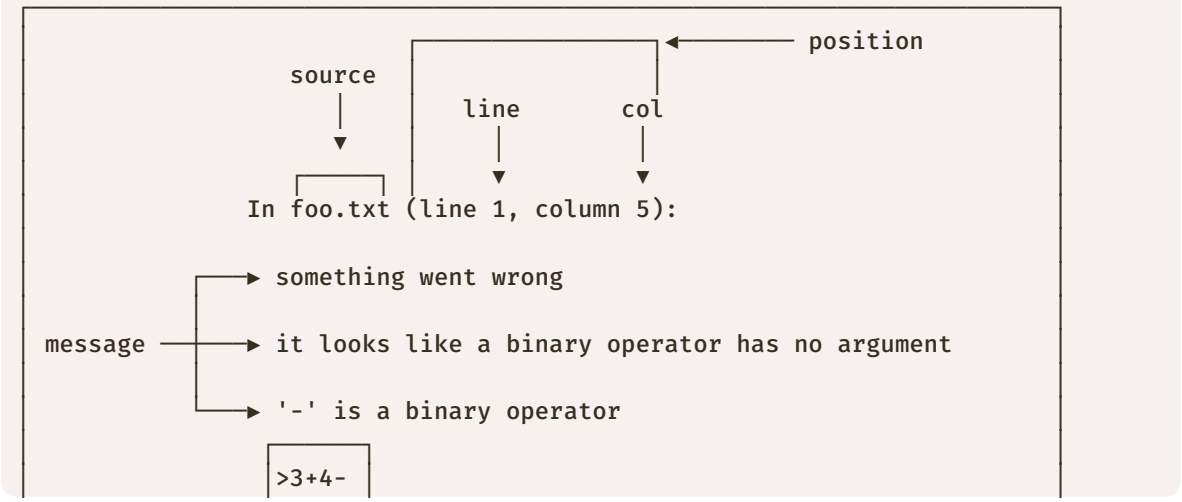





A vanilla error consists of three unique components: the `unexpected` component, which describes the problematic token; the `expected` component, which describes the possible parses that would have avoided the error; and the `reason` component, which gives additional context for an error. These are in addition to parts shared with *specialised* errors: the `source`, `position`, and the `lineInfo`. Any of the three unique components may be missing from the error, and the `ErrorBuilder` will need to be able to handle this.

Within both `unexpected` and `expected`, items can have one of three forms: `named`, which indicates they came from labels; `raw`, which means they came directly from the input itself; and `endOfInput`, which means no more input was available. All three of these states can be formatted independently.

## Specialised Errors



```
                  ^
                  ▲
                  |
                  line info
```

In contrast to the *vanilla* error, specialised errors have one unique component, `messages`, which is zero or more lines of bespoke error messages generated by `fail` combinators.

## The `ErrorBuilder` Typeclass

Within the `ErrorBuilder` trait, there is a number of undefined type aliases. Each of these must be implemented by an extender and provide an internal type to represent different components within the system. These are used to ensure maximal flexiblity of the user to pick how each component should be represented without exposing unnecessary information into the rest of the system.

After these types are specified, the methods of the typeclass can be implemented. These put together the primtive-most components and compose them into the larger whole. The documentation of the typeclass details the role of these well enough, however. For example's sake, however, these are the two shapes of call that will be made for the different types of error messages:

### Vanilla

```
(line 1, column 5):
  unexpected end of input
  expected "(", "negate", digit, or letter
  '-' is a binary operator
  >3+4-
       ^
```

```
val builder = implicitly[ErrorBuilder[String]]
builder.format (
    builder.pos(1, 5),
    builder.source(None),
    builder.vanillaError (
        builder.unexpected(Some(builder.endOfInput)),
        builder.expected (
            builder.combineExpectedItems(Set (
                builder.raw("("),
                builder.raw("negate"),
                builder.named("digit"),
                builder.named("letter")
            ))
        ),
        builder.combineMessages(List(
```

```
            builder.reason("'-' is a binary operator")
        )),
        builder.lineInfo("3+4-", Nil, Nil, 4, 4)
    )
)
```

One builder call not shown here, is a call to `builder.unexpectedToken`. This is a bigger discussion and is deferred to **Token Extraction in `ErrorBuilder`**

### Specialised

```
In file 'foo.txt' (line 2, column 6):
  first message
  second message
  >first line of input
  >second line
        ^^^
  >third line
  >fourth line
```

```
val builder = implicitly[ErrorBuilder[String]]
builder.format (
    builder.pos(2, 6),
    builder.source(Some("foo.txt")),
    builder.specialisedError (
        builder.combineMessages(List(
            builder.message("first message"),
            builder.message("second message"),
        )),
        builder.lineInfo("second line",
                         List("first line of input"),
                         List("third line", "fourth line"),
                         5,
                         3)
    )
)
```

## Constructing Test Errors

As an example of how to construct an `ErrorBuilder` for a type, consider the following representation of `TestError`:

```
case class TestError(pos: (Int, Int), lines: TestErrorLines)

sealed trait TestErrorLines
case class VanillaError(
    unexpected: Option[TestErrorItem],
```

```
    expecteds: Set[TestErrorItem],
    reasons: Set[String]) extends TestErrorLines
case class SpecialisedError(msgs: Set[String]) extends TestErrorLines

sealed trait TestErrorItem
case class TestRaw(item: String) extends TestErrorItem
case class TestNamed(item: String) extends TestErrorItem
case object TestEndOfInput extends TestErrorItem
```

This type, as will become evident from the formatter derived from it, is lossy and does not perfectly encode all the information available. Notice that `TestErrorItem` is a supertype of `TestRaw`, `TestNamed`, and `TestEndOfInput`: this is required, as the representation of each must all share a common supertype.

To construct an `ErrorBuilder[TestError]`, the type aliases must first be filled in:

```
class TestErrorBuilder extends ErrorBuilder[TestError] {
    type Position = (Int, Int)
    type Source = Unit
    type ErrorInfoLines = TestErrorLines
    type Item = TestErrorItem
    type Raw = TestRaw
    type Named = TestNamed
    type EndOfInput = TestEndOfInput.type
    type Message = String
    type Messages = Set[String]
    type ExpectedItems = Set[TestErrorItem]
    type ExpectedLine = Set[TestErrorItem]
    type UnexpectedLine = Option[TestErrorItem]
    type LineInfo = Unit
    //...
}
```

These types can be determined by examining the shape of `TestError`: for bits that it doesn't work, these are set to `Unit`. With these in place, the refined types of the typeclass methods make it very easy to fill in the gaps:

```
class TestErrorBuilder extends ErrorBuilder[TestError] {
    //...
    def format(pos: (Int, Int), source: Unit,
               lines: TestErrorLines): TestError = TestError(pos, lines)
    def vanillaError(
        unexpected: Option[TestErrorItem],
        expected: Set[TestErrorItem],
        reasons: Set[String],
        line: Unit
      ): TestErrorLines = VanillaError(unexpected, expected, reasons)
    def specialisedError(
        msgs: Set[String],
```

```
        line: Unit
      ): TestErrorLines = SpecialisedError(msgs)
    def pos(line: Int, col: Int): (Int, Int) = (line, col)
    def source(sourceName: Option[String]): Unit = ()
    def combineExpectedItems(alts: Set[TestErrorItem]): Set[TestErrorItem]
 = alts
    def combineMessages(alts: Seq[String]): Set[String] = alts.toSet
    def unexpected(item: Option[TestErrorItem]): Option[TestErrorItem] = item
    def expected(alts: Set[TestErrorItem]): Set[TestErrorItem] = alts
    def message(msg: String): String = msg
    def reason(msg: String): String = msg
    def raw(item: String): TestRaw = TestRaw(item)
    def named(item: String): TestNamed = TestNamed(item)
    val endOfInput: TestEndOfInput.type = TestEndOfInput

    val numLinesAfter: Int = 0
    val numLinesBefore: Int = 0
    def lineInfo(
        line: String,
        linesBefore: Seq[String],
        linesAfter: Seq[String],
        errorPointsAt: Int, errorWidth: Int
      ): Unit = ()

    // The implementation of this is usually provided by a mixed-in
    // token extractor, discussed in `tokenextractors`
    def unexpectedToken(
        cs: Iterable[Char],
        amountOfInputParserWanted: Int,
        lexicalError: Boolean
      ): Token = ???
}
```

Each of the methods above do the bare minimum work to satisfy the types. As noted in the comment, the implementation of `unexpectedToken` is usually done by mixing in a *token extractor*, which is explained **here**.

# Token Extraction in `ErrorBuilder`

When *vanilla* error messages are generated internally to `parsley`, the unexpected component is usually derived from the raw input, or a name explicitly given to an `unexpected` combinator. However, that does not necessarily provide the most informative or precise error messages.

Instead, the `ErrorBuilder` typeclass has an `unexpectedToken` method that can be used to determine how the token should be formulated in the event that it would have otherwise come raw from the input. Its signature is as follows:

```
def unexpectedToken(
    cs: Iterable[Char],
    amountOfInputParserWanted: Int,
    lexicalError: Boolean
  ): Token
```

The first argument, `cs`, is the input from the point that the bad input was found; the second is the amount of input the parser tried to read when it failed; and `lexicalError` denotes whether or not the failure happened whilst trying to parse a token from `Lexer`, or not. The return value, `Token`, is one of the following classes:

```
case class Named(name: String, span: TokenSpan) extends Token
case class Raw(tok: String) extends Token

sealed trait TokenSpan
case class Spanning(line: Int, col: Int) extends TokenSpan
case class Width(w: Int) extends TokenSpan
```

A `Raw` token indicates no further processing of the input could occur to get a better token, and some is returned verbatim. Otherwise, a `Named` token can replace a raw token with something derived from the input -- the `span` here denotes how wide that token had been determined to be.

> ⚠
>
> The `Spanning` class will be removed in `parsley:5.0.0`: in future, the width will be in `Named` directly.

The idea is that `unexpectedToken` should examine the provided arguments and determine if a more specific token can be extracted from the residual input, or, if not, produce a final `Raw` token of the desired width. In practice, while a user could implement the `unexpectedToken` method by hand, `parsley` provides a collection of *token extractors* that can be mixed-in to an `ErrorBuilder` to implement it instead.

## Basic Extractors

There are three basic extractors available in `parsley`: `SingleChar`, `MatchParserDemand`, and `TillNextWhitespace`. Each is discussed below. Each of them have special handling for whitespace characters and ones that are unprintable, which are given names.

### `SingleChar`

This extractor simply takes the first *codepoint* of the input stream `cs` and returns it. A *codepoint* is a single unicode character, which may consist of one or two bytes. As an example, the default formatting may be instantiated with this extractor by writing:

```
import parsley.errors.DefaultErrorBuilder
import parsley.errors.tokenextractors.SingleChar

val builder = new DefaultErrorBuilder with SingleChar
```

### `MatchParserDemand`

This extractor, as its name suggests, takes more than a single codepoint from the input, instead taking as many as the parser has requested via the `amountOfInputParserWanted` argument. As an example, the default formatting may be instantiated with this extractor by writing:

```
import parsley.errors.DefaultErrorBuilder
import parsley.errors.tokenextractors.MatchParserDemand

val builder = new DefaultErrorBuilder with MatchParserDemand
```

### `TillNextWhitespace`

Unlike the other extractors, this one has additional configuration. It generally aims to take as much input as necessary to find the next the next whitespace character, which can be changed by overriding the `isWhitespace` method. However, this can be capped as the minimum of the input the parser demanded or until the next whitespace. As an example, the default formatting may be instantiated with this extractor by writing:

```
import parsley.errors.DefaultErrorBuilder
import parsley.errors.tokenextractors.TillNextWhitespace

val builder = new DefaultErrorBuilder with TillNextWhitespace {
    def trimToParserDemand = true
}
```

This extractor, with `trimToParserDemand = true` is the default currented used by `parsley` for all error messages. By default, `isWhitespace` matches any character `c` for which `c.isWhitespace` is true.

## `Lexer`-backed Extraction

The default strategies outlined above all ignore the `lexicalError` flag passed to `unexpectedToken`. To provide a more language-directed token extraction, however, the `LexToken` extractor is also provided.

It has one compulsory configuration and two more that have defaults:

```
trait LexToken {
    def tokens: Seq[Parsley[String]]
    def extractItem(cs: Iterable[Char], amountOfInputParserWanted: Int): Token
 = {
        SingleChar.unexpectedToken(cs)
    }
    def selectToken(matchedToks: List[(String, (Int, Int))]): (String,
 (Int, Int)) = {
        matchedToks.maxBy(_._2)
    }
}
```

Here, the `tokens` are parsers for valid tokens within the language being parsed: each returns the name of that token as it would be displayed in the error message. The extractor will try to parse all of these tokens, and should at least one succeed the **non-empty** list of parsed tokens will be passed to `selectToken` for one to be picked to be used in the error: by default, the one which is the widest is chosen. If no tokens could be parsed, or the error occured *during* the parsing of a token/within the `markAsToken` combinator (as denoted by `lexicalError` normally), then `extractItem` is used instead. This usually should defer to another kind of token extractor, which, for convience, all expose their functionality in their companion objects.

> ⚠
>
> The intention of the `tokens` sequence is that they should *not* consume whitespace: were they to do so, this whitespace would form part of the generated token! When using `Lexer` to fill this sequence, be sure to use `lexer.nonlexeme` to source the tokens.

As an example, a language which already has an available `lexer` built with lexical description `desc` can implement a `LexToken` as follows:

```
import parsley.errors.DefaultErrorBuilder
import parsley.errors.tokenextractors.LexToken

val builder = new DefaultErrorBuilder with LexToken {
    def tokens = Seq(
        lexer.nonlexeme.integer.decimal.map(n => s"integer $n"),
        lexer.nonlexeme.names.identifier.map(v => s"identifier $v")
    ) ++ desc.symbolDesc.hardKeywords.map { k =>
        lexer.nonlexeme.symbol(k).as(s"keyword $k")
    }
}
```

Obviously, this may not be an exhaustive list of tokens, but is illustrative of how to set things up.

# Parser Combinator Tutorial

Parsley is a *parser combinator library*. In contrast to a parser generator library, like ANTLR, this allows the users to build their parsers as part of the host language: in this case Scala. This brings a huge amount of power to the fingertips of the programmer, but, admittedly, embarking on the journey to learning how they work is very intimidating! Personally, I think the pay-off is still great, and, really, it's quite hard to see why until after you've tried it. The most important part is not being afraid to play with things.

This series of wiki posts aims to help guide you on your journey to learning parser combinators. Unfortunately, while the fundamentals carry nicely over to other libraries, such as `parsec`, `megaparsec` and so on in Haskell (and to a lesser extent the other Scala libraries, which admittedly have a slightly different feel to them), learning a parser combinator library is still a specialism, and many of the helpful abstractions and patterns that Parsley provides may not be available or even possible in some other contexts.

Something I've learnt about this wiki is that people can take it fairly literally at each step instead of viewing each page as a larger whole. The consequence is that some of the neat techniques that are presented later in the series may have come too late, and users may have already implemented something the "long way". My advice is to **keep reading** until the end before embarking on any serious work using the library. Of course, you don't have to, but I've distilled *years* of knowledge into this wiki, and it would be a shame to miss out on it. To try and make this process easier, I've added a road-map of sorts below, to help you understand the whole story before you start it.

## The Roadmap

### Basics of Combinators

Our journey starts at the beginning: the very beginning. In many respects, this first post probably goes into too much detail. However, I think it's very important to see the lower level primitives to be able to understand the more powerful and convenient abstractions. In particular, the nuances of backtracking can have a big effect on the error messages of even correct parsers, so understanding *how* and *when* to backtrack is an important skill. That being said, the way that parsers are written in this post are **not** representative of how parsers really **should** be written! Instead it demonstrates some recurring *themes* and some of the most important ideas.

### Building Expression Parsers (*introduces* `expr.precedence`)

By the end of the first post, the basic combinators, recursion and the main principles of combinators have been demonstrated. However, the final example, which sees us write a parser for boolean expressions, leaves much to be desired. For the uninitiated reader, they'll see nothing wrong with it, but as the second page shows, there are much easier and more powerful tools at our disposal for working with expressions. Again, this page starts by showing off the fundamental building blocks of the expression parser before showing the more commonly used `precedence` combinator. The reason I chose to take this route is similar

to before, it's good to be able to have a sense of how the more powerful tools were built up, and there are often opportunities to use these combinators where a `precedence` (which takes the center stage) is just a bit more clunky.

Be warned that this page doesn't re-introduce any of the material of the previous page, so if a combinator is used without being explained, you can always check back to the first post (or the cheatsheet). Also you should be aware that the latter half of the post gets a bit more technical than most people will a) need to know and b) care about. The reason these sections were left in was to help the many people who don't like to blindly accept concepts presented to them without having a, however basic, understanding of how it works. If you're not one of those people, or you're otherwise not interested, then you can feel free to move on to the next page at the section *Path-Dependent Typing and* `Ops`/`Gops`.

## Effective Whitespace Parsing

By the third post, we take a (welcome) break from learning new combinators and concepts, and instead discuss good parser *design*, and the best ways to deal with pesky whitespace in the input: from the first two posts, we'll already have seen all the tools we need to write correct parsers, just not the *best* ways to do so.

Whitespace, simply put, is annoying because it distracts from the rest of the grammar. Normally, a lexer is used to deal with whitespace, and the grammar more accurately describes the relationships between tokens. The basic idea behind this page is to demonstrate how we can start to use Scala's features to develop handy abstractions for ourselves that make whitespace disappear from the main parser. Again, there are better tools for dealing with these issues than hand-rolling them ourselves, but in order to use such tools effectively and really understand their implication, it's a very good idea to understand the fundamentals: the **fourth** page more effectively deals with the issues highlighted in this page *and* uses some of the techniques introduced, in the process refining them.

## Effective Lexing (*introduces* `Lexer`)

The fourth post builds on the ideas of its predecessor, first outlining the general principles behind how we write and structure the lexical parsers for our grammar, and then how to seemlessly integrate them into the parser proper. The ideas here are very similar to those already laid out previously.

Unlike the third post, here the mighty `Lexer` class is introduced. While it's not always needed to write parsers, its usefulness, even for just handling whitespace, can't be understated. It's not always the right tool for the job though, so definitely don't disregard all the lessons presented before it!

## The *Parser Bridges* Pattern (*introduces position tracking with* `line`, `col`, *and* `pos`)

This page takes a huge leap forward in terms of how parsers are designed and integrated with the Abstract Syntax Trees they so often produce. An important (and often overlooked) aspect of parsing with combinators is how position information in the parser is preserved in the generated structure. In my experience, I've found this is often done as an afterthought, when the programmer realises that the information is important: for instance, when performing Semantic Analysis in a compiler... And, usually, its introduction makes a complete mess of an otherwise nice looking parser.

The joy of the *Parser Bridges* pattern, which this page introduces, is that it separates the building of the AST during parsing from whether or not that AST node needs position information, or indeed the mechanics of putting together the components in the right way. This separation creates a pleasant cleanliness in the main body of the grammar, which by this point now retains the simple elegance we might expect to see with a plain old BNF representation. If you've been unconvinced so far that parser combinators look very similar to the grammar they represent, this may change your perspective.

## *Interlude 1*: Building a Parser for Haskell

By this point, you'll have covered a lot of information:

- Basics of what combinators are and what they are built from
- Cleanly handing expressions with varying precedence and associativities
- How to correctly deal with whitespace
- How to cleanly factor out lexing logic
- How to abstract away the construction of a resulting AST from the grammar

With this wealth of knowledge, you'll have the power to go and write all but the trickiest of parsers. To demonstrate that, the first of three interludes will work through the structure and design of a (simplified) Haskell parser with the tools we've accrued so far. Even though this will help consolidate everything you've been shown by putting it all into practice, there is still a big chunk of the story missing: error messages.

## Customising Error Messages (*introduces* `label`, `explain`, *and* `ErrorBuilder`)

With the mechanics of writing parsers that can *succeed* out of the way, it's about time to learn about how to improve error messages that *failing* parsers produce. By default, the error messages, whilst not *bad*, aren't nearly as they could be. The `Lexer` class does help with this for lexemes at least, but that doesn't mean all the work is done: especially in the main grammar, or for times when `Lexer` was a no-go. Writing good error messages is an art-form, and so this page takes a more subjective look at the process. For most people, this is just about as far as you'd need to go.

The second part of this post explains how to use the `ErrorBuilder` mechanism in Parsley to customise the format of error messages, or even change what type they have. This can be particularly useful for unit-testing or for standardising parser error messages as part of a larger compiler.

## Advanced Error Messages (*introduces* `unexpected` *and* `fail`)

The combinators introduced in the previous page are already pretty good! But there are still some neat patterns we can use to kick it up a notch or two. In particular, this page introduces patterns that can be used to *anticipate* common syntax errors and produce much more descriptive errors upon encountering them.

## *Interlude 2*: Adding Errors to the Haskell Parser

The second interlude takes the new-found lessons from the previous two pages to augment the Haskell parser with error messages, illustrating the considerations and patterns in practice. The reason that Interlude 1 comes before error messages is that, whilst they aren't particularly obstructive, the error message combinators provide a little extra noise that makes the core part of the parser a little bit harder to admire, especially for someone who is only getting to grips with the concepts for the very first time!

This, for almost all use-cases, the end of the story. By this point you'll have all the tools you need to parse context-free grammars, which make up the vast majority of practical languages and data formats *and* generate good error messages for them. If however, you are keen to learn about context-sensitive grammars, or you are thoroughly engrossed in the story up to this point, there is one final stretch.

## Indentation Sensitive Parsing

For most languages, the grammar is constructed in such a way that it remains context-free. This is, primarily, because context-sensitive grammars are a brutal combination of hard to express and hard to parse efficiently. Indentation-sensitive parsing *can* be considered an example of a context-sensitive grammar, though, in practice, some compilers like to shunt the work out to the lexer to make the grammar context-free again (this is the case with Python).

Using parser combinators though, context-sensitive grammars can be encoded comparatively naturally! In most other combinator libraries, the `flatMap` (or `>>=`) combinator is used to deal with context-sensitivity. However, in the Parsley family, the power that `flatMap` provides comes at a heavy cost to performance. Instead, we reach for stateful parsers called "references", evoking images of register machines vs stack machines: as we know, register machines are turing powerful, and can most certainly do the job, no matter the parsing task.

This page provides a more concrete and gentle introduction to using references specifically demonstrating how to use them to implement combinators for managing indentation-sensitive workloads in a clean and effective way.

## *Interlude 3*: Supporting the Haskell Offside Rule

As the final act of this series, the last "interlude" (by this point just a finale) takes the combinators built up in the previous page to add the off-side rule to the Haskell parser: this is the essence of Haskell's indentation-sensitive syntax.

# Basics of Combinators

Parsley is a *parser combinator* library. In contrast to a parser generator library, like ANTLR, this allows the users to build their parsers as part of the *host* language: in this case Scala. This is called being an *embedded Domain Specific Language* or eDSL. Practically, this is useful because it allows you to factor out repeated parts of your grammars and make them reusable, as well as using all the language features normally at your disposal to create the grammars too. This page will touch on both of those ideas. Another advantage is that there is less boiler-plate compared with *some* parser generators: you don't need to convert between the AST the generator produces and your own, you can parse straight into the desired type.

## 1 Basic Combinators and Sequencing

We'll start really basic: just reading a character or two and seeing how to combine the results using *combinators*. For a first look, we will just parse one of any character. If you are familar with regex, this would match the pattern `(.)`.

```
import parsley.Parsley
import parsley.character.item

val p: Parsley[Char] = item
```

```
p.parse("a")
// res0: parsley.Result[String, Char] = Success(x = 'a')
p.parse("1")
// res1: parsley.Result[String, Char] = Success(x = '1')
p.parse("")
// res2: parsley.Result[String, Char] = Failure(
// ...
```

The **Parsley** type is the type of parsers. The type parameter `Char` here represents what type the parser will return when it has been executed using `parse(input)`. Soon we will see an example with a different type. Parsers, when executed, return a **Result**`[Err, A]` for whatever `A` the parser returned: this is one of **Success** containing the value or **Failure** containing an error message of type `Err` (by default this is `String`). This is the basic workflow when using parsers. The `item` parser will read any single character, no matter what (so long as there is one to read). It isn't particularly useful though, so lets match specific characters instead and parse *two* of them this time. The regex for this would be `(ab)`.

```
import parsley.Parsley
import parsley.character.char

val ab: Parsley[(Char, Char)] = char('a') <~> char('b')
```

```
ab.parse("ab")
// res3: parsley.Result[String, (Char, Char)] = Success(x = ('a', 'b'))
ab.parse("a")
// res4: parsley.Result[String, (Char, Char)] = Failure(
// ...
```

A few new things have appeared in this new example. The `char` combinator is new: given a specific character it will parse that character only. We'll see how you can define this and `item` in terms of another, more general, combinator soon. Notice that the type of `ab` is no longer just a `Parsley[Char]`, but a `Parsley[(Char, Char)]`: this is due to the <~> (pronounced "zip") combinator with the following type (in a pseudo-syntax, for simplicity).

```
(_ <~> _): (p: Parsley[A], q: Parsley[B]) => Parsley[(A, B)]
```

What this combinator does (pronounced "zip") is that it first parses `p`, then `q` afterwards and then combines their results into a tuple. Suppose we had `char('a') <~> char('b') <~> char('c')` then this would have type `Parsley[((Char, Char), Char)]`. This is the first example of a sequencing combinator. There are two other combinators that look similar:

```
(_ ~> _): (p: Parsley[A], q: Parsley[B]) => Parsley[B]
(_ <~ _): (p: Parsley[A], q: Parsley[B]) => Parsley[A]
```

They are pronounced `then` and `then discard` respectively. Again, they both parse both `p` and then `q`, but they only return the result they are "pointing" at. Notice that <~> points at *both* results. These are more widely known as *> and <*, but they are otherwise identical, so use whatever resonates more strongly with you. We'll see soon how we can define them in terms of <~> to get a sense of how combinators can be built up in terms of more primitive ones.

## 1.1 What ties `char` and `item` together

We've seen both `char` and `item` can be used to read characters, there is, however, a more primitive one which can be used to implement them both. This combinator is called `satisfy` and has the following type:

```
def satisfy(predicate: Char => Boolean): Parsley[Char]

def char(c: Char): Parsley[Char] = satisfy(_ == c)
val item: Parsley[Char] = satisfy(_ => true)
```

The combinator `satisfy` takes a function, and will read a character when the predicate returns `true` on that character, and fails otherwise. This makes `satisfy` a bit more versatile and it can be used to implement a wide range of functionality. For example, we can implement a parser that reads digits using `satisfy`:

```
import parsley.Parsley
import parsley.character.satisfy

val digit = satisfy(_.isDigit)
```

```
digit.parse("1")
// res5: parsley.Result[String, Char] = Success(x = '1')
digit.parse("2")
// res6: parsley.Result[String, Char] = Success(x = '2')
digit.parse("a")
// res7: parsley.Result[String, Char] = Failure(
// ...
```

This is, however, already implemented by `parsley.character.digit`; `parsley` is very rich in terms of the combinators it supports out of the box, so do hunt around before reinventing the wheel!

## 1.2 Changing the result type

It's all well and good being able to sequence together reading single characters, but this doesn't exactly scale well to larger, more complex, parsers. Indeed, it's likely we aren't interested in an increasing deeply nested tuple! A good starting point for this is the humble `map` combinator:

```
_.map(_): (p: Parsley[A], f: A => B) => Parsley[B]
```

This can be used to change the result of a parser `p` with the parser `f`, presumably into something more useful. Let's see a couple of examples of this in action! Firstly, let's suppose we wanted our `digit` combinator from before to return an `Int` instead of a `Char`...

```
import parsley.Parsley
import parsley.character.satisfy

val digit: Parsley[Int] = satisfy(_.isDigit).map(_.asDigit)
```

```
digit.parse("1")
// res8: parsley.Result[String, Int] = Success(x = 1)
```

Here we can see that the digit parser is no longer type `Parsley[Char]` but type `Parsley[Int]`. This is because the `asDigit` method on `Char` returns an `Int`. To reinforce how this works, let's see how ~> and <~ can be made out of a combination of <~> and `map`:

```
p ~> q == (p <~> q).map(_._2)
p <~ q == (p <~> q).map(_._1)
```

The first definition pairs `p` and `q` together, and then takes the second element of the pair with `map`, and the second definition does the same but instead takes the *first* element of the pair. Now, using this tupling approach paired with `map`, we can do a lot of stuff! However, there is a more general strategy to do this:

```
def lift2[A, B, C](f: (A, B) => C, p: Parsley[A], q: Parsley[B]): Parsley[C]

// pairs p and q's results together
p <~> q = lift2[A, B, (A, B)]((_, _), p, q)
// adds the result of p onto the list result of ps
p <::> ps = lift2[A, List[A], List[A]](_ :: _, p, ps)
// applies a function from pf onto the value from px
pf <*> px = lift2[A => B, A, B]((f, x) => f(x), pf, px)
...
```

The `lift` *family* of combinators are great for combining `n` parsers with an arity `n` function. For instance, `map` is actually the same as a `lift1`. And above we can see that `lift2` can implement a bunch of useful combinators. In particular, let's see how we can use <::> to implement a way of reading `String`s instead of just `Char`s!

## 1.3 Putting the pieces together: Building `string`

Our new challenge is going to be making an implementation of the `string` combinator. Obviously, this combinator already exists in the library, so we can play around with it first to see how it works:

```
import parsley.Parsley
import parsley.character.string

val abc = string("abc")
```

```
abc.parse("abc")
// res9: parsley.Result[String, String] = Success(x = "abc")
abc.parse("abcd")
// res10: parsley.Result[String, String] = Success(x = "abc")
abc.parse("ab")
// res11: parsley.Result[String, String] = Failure(
// ...
```

```
abc.parse("a bc")
// res12: parsley.Result[String, String] = Failure(
// ...
abc.parse("dabc")
// res13: parsley.Result[String, String] = Failure(
// ...
```

Notice how the result of the parser is a string. The `string` combinator reads a specific string exactly. Here are a couple more examples to help you get your head around everything we've seen so far:

```
import parsley.character.{char, string}

(string("abc") <~ char('d')).parse("abcd")
// res14: parsley.Result[String, String] = Success(x = "abc")
(string("abc") ~> char('d')).parse("abcd")
// res15: parsley.Result[String, Char] = Success(x = 'd')
(string("abc") <~> char('d')).parse("abcd")
// res16: parsley.Result[String, (String, Char)] = Success(x = ("abc", 'd'))
```

Now let's start building the `string` combinator from scratch! Bear in mind, that unlike in Haskell, a Scala string is not `List[Char]` but is the Java `String`. This makes it a little more annoying to implement, since we'll have to convert a `List[Char]` into a `String` at the end, with `map`.

```
import parsley.Parsley

def string(str: String): Parsley[String] = {
    def helper(cs: List[Char]): Parsley[List[Char]] = ???
    helper(str.toList).map(_.mkString)
}
```

We've started here by defining the `string` function, and made the skeleton of an internal helper function that will turn a list of characters into a parser that reads that list of characters and returns them all. The main body of the function uses this, and afterwards maps the `_.mkString` method on lists to convert the result back into a string. Now we need to focus on the helper. The first step is to consider how to handle the empty string. For this we need another very handy combinator called `pure`, which reads no input and returns what's given to it:

```
import parsley.Parsley, Parsley.pure

// def pure[A](x: A): Parsley[A]
pure(7).parse("")
// res17: parsley.Result[String, Int] = Success(x = 7)

def helper(cs: List[Char]): Parsley[List[Char]] = cs match {
    case Nil    => pure(Nil)
    case _ :: _ => ???
}
```

Now the question is how to handle the recursive case? Well in the base case we transformed the empty list into a parser that returns the empty list. We'll follow that same shape here and use `<::>`!

```
import parsley.Parsley, Parsley.pure
import parsley.character.char

def helper(cs: List[Char]): Parsley[List[Char]] = cs match {
    case Nil     => pure(Nil)
    case c :: cs => char(c) <::> helper(cs)
}
```

What happens here is that we take each character in the string, convert it to a parser that reads that specific character, and then add that onto the front of reading the rest of the characters. In full:

```
import parsley.Parsley
import parsley.character.char

def string(str: String): Parsley[String] = {
    def helper(cs: List[Char]): Parsley[List[Char]] = cs match {
        case Nil     => pure(Nil)
        case c :: cs => char(c) <::> helper(cs)
    }
    helper(str.toList).map(_.mkString)
}

// string "abc" == (char('a') <::> (char ('b') <::> (char 'c' <::>
 pure(Nil)))).map(_.mkString)
```

Hopefully, this gives some intuition about how we can start to sequence together larger and larger building blocks out of smaller ones. It's also a lesson in how Scala can be used to help you build your parsers up! Again, the `string` combinator is already provided to you (and optimised) so be sure to check around in `parsley.character` and `parsley.combinator` for combinators that might already fit your needs. That's about everything there is to say about sequencing and combining results, so next up is looking at *choice*.

### 1.3.1 Takeaways

- Characters can be read using combinators found in `parsley.character`
- To sequence two things but only use the result of one, you'll want `*>`/`~>` or `<*`/`<~`
- The result of a parser can be changed using `map`
- `N` parser's results can be combined using the `liftN` combinators with an arity `N` function
- Larger combinators are built out of smaller ones using regular Scala functionality

## 2 Choice and Handling Failure

Most practical parsers aren't just a straight line like `string` or reading a bunch of characters, usually there are choices to be made along the way.

### 2.1 Alternatives

When parsers fail to recognise some input, most of the time, there is an alternative branch that could have been taken instead. Let's take a simple example again, say matching the regex `(a|b)`. From now on, I'm going to use some syntactic sugar from `parsley.implicits` so I don't have to write `char` or `string`.

```
import parsley.Parsley
import parsley.syntax.character.charLift

val aOrB = 'a' <|> 'b'
```

```
aOrB.parse("a")
// res18: parsley.Result[String, Char] = Success(x = 'a')
aOrB.parse("b")
// res19: parsley.Result[String, Char] = Success(x = 'b')
aOrB.parse("c")
// res20: parsley.Result[String, Char] = Failure(
// ...
```

Here, the `<|>` combinator (pronounced "alt" or "or") allows the parser to try an alternative branch when the first fails (and so on for a longer chain of them). To be well typed, the `<|>` combinator requires both branches to return the same type (or at least a super-type of both). There is also a shorter way of writing this combinator, called `|` - this doesn't work properly on the character literals though, for obvious reasons. For this specific usecase, `character.oneOf('a', 'b')` would probably have been more appropriate.

Let's carry on reinforcing the connections with what we've seen so far, and see how sequencing and branching interact:

```
import parsley.Parsley
import parsley.syntax.character.{charLift, stringLift}

val p = 'a' ~> ("a" | "bc") <~ 'd'

p.parse("bcd") // fails, there isn't an 'a'
p.parse("ad") // fails, why?
p.parse("aae") // fails, why?
p.parse("abcde") // what happens here, what does it return (and the type)?
p.parse("aad") // what happens here, what is the type it returns?
p.parse(" aad") // what about this
```

Think about the what results you expect from each of these, and then try them out in a REPL to see if you're right: also check out the error messages from the failing ones! Recall that `string` basically reads a bunch of characters in sequence, one after the other. Let's see what happens when we put longer strings inside the branches:

```
import parsley.Parsley
import parsley.syntax.character.stringLift

val p = "abc" | "def" | "dead"
```

```
p.parse("abc")
// res27: parsley.Result[String, String] = Success(x = "abc")
p.parse("def")
// res28: parsley.Result[String, String] = Success(x = "def")
p.parse("dead")
// res29: parsley.Result[String, String] = Failure(
// ...
```

Ah, we have a problem! The first two alternatives parse fine, but the last one does not? The answer to this is fairly simple, but I want to illustrate how we can make steps towards diagnosing this problem ourselves using the combinators found in `parsley.debug`:

```
import parsley.Parsley
import parsley.syntax.character.stringLift
import parsley.debug._

val p = ("abc".debug("reading abc") |
            ("def".debug("reading def") | "dead".debug("reading
 dead")).debug("second branch")
        ).debug("first branch")
```

The `debug` combinator can be attached to any operation (by default Parsley associates `<|>` to the right, which is why I've bracketed them this way round). It will provide printouts when it enters the debug and when it exits, along with information about the state of the parser. Let's see the three printouts:

```
p.parse("abc")
// >first branch> (1, 1): abc•
//                        ^
//   >reading abc> (1, 1): abc•
//                         ^
//   <reading abc< (1, 4): abc• Good
//                           ^
// <first branch< (1, 4): abc• Good
//                          ^
// res31: parsley.Result[String, String] = Success(abc)
p.parse("def")
// >first branch> (1, 1): def•
//                        ^
//   >reading abc> (1, 1): def•
//                         ^
//   <reading abc< (1, 1): def• Fail
//                         ^
//   >second branch> (1, 1): def•
//                           ^
//     >reading def> (1, 1): def•
//                           ^
//     <reading def< (1, 4): def• Good
//                             ^
//   <second branch< (1, 4): def• Good
//                            ^
// <first branch< (1, 4): def• Good
//                          ^
// res32: parsley.Result[String, String] = Success(def)
p.parse("dead")
// >first branch> (1, 1): dead•
//                        ^
//   >reading abc> (1, 1): dead•
//                         ^
//   <reading abc< (1, 1): dead• Fail
//                         ^
//   >second branch> (1, 1): dead•
//                           ^
//     >reading def> (1, 1): dead•
//                           ^
//     <reading def< (1, 3): dead• Fail
//                             ^
//   <second branch< (1, 3): dead• Fail
//                            ^
// <first branch< (1, 3): dead• Fail
//                          ^
// res33: parsley.Result[String, String] = Failure((line 1, column 1):
//   unexpected "dea"
//   expected "abc" or "def"
//   >dead
//    ^^^)
```

Crucially, in the last printout, we can see the trace of the parser as it went wrong. It started by executing the outermost branch, and tried to read `"abc"` and failed, but the caret is still pointing at the first character. Then the second branch is taken as the alternative to the first: this time the parser tries to read `"def"` and gets *two characters* of the way through it before failing at the `'a'`. Notice though, that the second branch immediately exited without attempting the third alternative! This is the key: when a parser fails but has *consumed input*, the `|` combinator will not work. The reason for this is to improve the quality of error messages, as well as keeping parsers efficient. The "best" solution to this problem is to change our parser slightly to remove the common leading string of the last two alternatives like so:

```
import parsley.Parsley
import parsley.syntax.character.{charLift, stringLift}

val p = "abc" | ("de" ~> ('f'.as("def") | "ad".as("dead")))
```

```
p.parse("abc")
// res34: parsley.Result[String, String] = Success(x = "abc")
p.parse("def")
// res35: parsley.Result[String, String] = Success(x = "def")
p.parse("dead")
// res36: parsley.Result[String, String] = Success(x = "dead")
```

In this version of the parser, the leading `"de"` has been factored out, leaving an alternative of `"f" | "ad"` remaining. But the original parser returned the full string, and this wouldn't: it would return "abc", "f", or "ad". The `as` combinator (which can be written as `#>`) will replace the result of parser on the left with the value found on the right (e.g. `"123".as(123) | "456".as(456)` would be `Parsley[Int]`). You can think of it as a `map` with the constant function or as `p ~> pure(x)`. Using `as` we can replace the results of the last two parsers with the results we expected from before. This is the most efficient way of dealing with our problem, because this parser is still linear time in the worst case. But sometimes this transformation isn't so straight-forward, especially in deeply nested grammars. In this case we can reach for another, easier to read, tool.

## 2.2 Backtracking

In the last section, we saw that the `|` doesn't proceed with the second alternative if the first consumed input before failing. That is to say it doesn't *backtrack*. There is, however, a combinator that allows `|` to backtrack in these circumstances, called `atomic`. Let's see it in action:

```
import parsley.Parsley, Parsley.atomic
import parsley.syntax.character.stringLift
import parsley.debug._

val p = "abc" | atomic("def") | "dead"
// p: Parsley[String] = parsley.Parsley@4b632ae8
```

```
val q = "abc" | (atomic("def".debug("reading def")).debug("backtrack!") |
                  "dead".debug("reading dead")).debug("branch")
// q: Parsley[String] = parsley.Parsley@290278d

p.parse("dead")
// res37: parsley.Result[String, String] = Success(dead)
q.parse("dead")
// >branch> (1, 1): dead•
//                   ^
//   >backtrack!> (1, 1): dead•
//                        ^
//     >reading def> (1, 1): dead•
//                           ^
//     <reading def< (1, 3): dead• Fail
//                             ^
//   <backtrack!< (1, 1): dead• Fail
//                        ^
//   >reading dead> (1, 1): dead•
//                          ^
//   <reading dead< (1, 5): dead• Good
//                                ^
// <branch< (1, 5): dead• Good
//                       ^
// res38: parsley.Result[String, String] = Success(dead)
```

Here we can see `atomic` in action, as well as a debug trace so you can see what's going on. When we wrap the left hand side of a branch with `atomic`, when it fails we will rollback any input it consumed, which then allows the branch to accept the alternative. We can see that in the debug trace. You only need to use `atomic` where you know that two branches share a common leading edge. Knowing when to do this is just based on practice. Adding an `atomic` never makes a parser *wrong*, but it can make the error messages worse, and also excessive backtracking can increase the time complexity of the parser significantly. If you know that if a branch consumes input and fails then its alternatives wouldn't succeed either, then you shouldn't be using `atomic`. It is also useful to make a specific sub-parser behave as if it were indivisible: think reading keywords, which are all or nothing.

## 2.3 Lookahead

Usually, a good ordering of your alternatives is most of what you need to write a functioning parser for just about anything. However, every now and then it's convenient to look-ahead at what is to come (in either a positive or a negative way): for instance checking if there is no remaining input with the `eof` combinator is an example of negative look-ahead. There are two combinators for doing this, which we'll explore now:

```
import parsley.Parsley, Parsley.{notFollowedBy, lookAhead}
import parsley.character.item
import parsley.syntax.character.stringLift
import parsley.debug._
```

```
// def lookAhead[A](p: Parsley[A]): Parsley[A]
// def notFollowedBy(p: Parsley[_]): Parsley[Unit]

val eof = notFollowedBy(item)
// eof: Parsley[Unit] = parsley.Parsley@6ace6601
val abcOnly = "abc" <~ eof
// abcOnly: Parsley[String] = parsley.Parsley@3e6aeaf3

abcOnly.parse("abc")
// res39: parsley.Result[String, String] = Success(abc)
abcOnly.parse("abcd")
// res40: parsley.Result[String, String] = Failure((line 1, column 4):
//   unexpected "d"
//   >abcd
//       ^)

val p = "abc".debug("abc") <~ lookAhead("!!".debug("!!")).debug("lookahead")
// p: Parsley[String] = parsley.Parsley@364952ac

p.parse("abc!!")
// >abc> (1, 1): abc!!•
//               ^
// <abc< (1, 4): abc!!• Good
//                  ^
// >lookahead> (1, 4): abc!!•
//                        ^
//   >!!> (1, 4): abc!!•
//                  ^
//   <!!< (1, 6): abc!!• Good
//                    ^
// <lookahead< (1, 4): abc!!• Good
//                        ^
// res41: parsley.Result[String, String] = Success(abc)

p.parse("abc!")
// >abc> (1, 1): abc!•
//               ^
// <abc< (1, 4): abc!• Good
//                  ^
// >lookahead> (1, 4): abc!•
//                        ^
//   >!!> (1, 4): abc!•
//                  ^
//   <!!< (1, 5): abc!• Fail
//                   ^
// <lookahead< (1, 5): abc!• Fail
//                        ^
// res42: parsley.Result[String, String] = Failure((line 1, column 4):
//   unexpected "!"
//   expected "!!"
//   >abc!
//       ^)
```

Some key things to note here: the result of backtracking is always `()`. This is because the parser has to fail for `notFollowedBy` to succeed, so it can't return the result of the look-ahead! On the other hand, `lookAhead` can return the result of the parser that was ran. You can see from the debug traces that when it succeeds, `lookAhead` does *not* consume input, but if it fails having consumed input, that input *remains consumed*. However, `notFollowedBy` never consumes input.

### 2.3.1 Takeaways

- Alternative branches of a grammar are encoded by `|` (also known as `<|>` or `orElse`)
- Within a branch, you are free to do whatever you want, but you must ensure both branches' types match
- When a branch fails having consumed input it won't take the second branch.
- The `atomic` combinator can be used to enable backtracking so that consumed input is undone when passing back through (it doesn't affect any `|`s that execute inside it, however)
- When parsers go wrong, `debug` is a fantastic tool to investigate it with: use it early and often!
- Negative and positive look-ahead can be done with `lookAhead` and `notFollowedBy`

## 3 Interlude: Regex Parser Examples

Before we move on to slightly more realistic parsing problems that can't just be captured by regex, we'll consolidate what we've seen so far by writing a few parsers for some regular expressions. For all of these examples, I'll simplify it by using `void`, which ignores the result of a parser and replaces it with `(): Unit`. This turns our parsers into *recognisers*. I'll also be introducing a couple of new ideas so we can complete the functionality we need to properly capture regex (namely the equivalents of optional `?`, zero-or-more `*` and one-or-more `+`). Let's start there:

```
// This is the regex *
// it will perform `p` zero or more times (greedily) and collect all its
 results into a list
def many[A](p: Parsley[A]): Parsley[List[A]] = ???
def skipMany(p: Parsley[_]): Parsley[Unit] =
    many(p).void // ideally, it wouldn't build the list

// This is the regex +
// similar to many, except it requires at least 1 `p` to succeed
def some[A](p: Parsley[A]): Parsley[List[A]] = p <::> many(p)
def skipSome(p: Parsley[_]): Parsley[Unit] = p ~> skipMany(p)

// This is the regex ?
// it will either parse `p` or will return `x` otherwise
def optionally[A](p: Parsley[_], x: A): Parsley[A] = p.as(x) <|> pure(x)
def optional(p: Parsley[_]): Parsley[Unit] = optionally(p, ())
```

```
def option[A](p: Parsley[A]): Parsley[Option[A]] =
    p.map(Some(_)) | pure(None)

// This is the regex [^ .. ]
// it will parse any character _not_ passed to it
def noneOf[A](cs: Char*): Parsley[Char] = satisfy(!cs.contains(_))
```

With the exception of `many`, which we can't define just yet, all of these handy combinators are implemented with everything we've seen so far. You can find them all, and many more, in `parsley.combinator`.

```
import parsley.Parsley, Parsley.{atomic, eof, many, some}
import parsley.combinator.optional
import parsley.syntax.character.{charLift, stringLift}
import parsley.character.{noneOf, oneOf, item}

// regex .at
val r1: Parsley[Unit] = item ~> "at".void

// regex [hc]at
val r2: Parsley[Unit] = oneOf('h', 'c') ~> "at".void

// regex [^hc]at
val r3: Parsley[Unit] = noneOf('h', 'c') ~> "at".void

// regex [hc]at$
val r4: Parsley[Unit] = oneOf('h', 'c') ~> "at" ~> eof

// regex s.*
val r5: Parsley[Unit] = 's' ~> many(item).void

// regex [hc]?at
val r6: Parsley[Unit] = optional(oneOf('h', 'c')) ~> "at".void

// regex [hc]+at
val r7: Parsley[Unit] = some(oneOf('h', 'c')) ~> "at".void

// regex h(i|ello|ey)( world)?(\!|\.)?
val r8: Parsley[Unit] = 'h' ~> ("i" | atomic("ello") | "ey") ~>
                        optional(" world") ~>
                        optional('!' <|> '.').void
```

Have a play around with those in a REPL and make sure you understand how they work and what inputs they will succeed on. The `void` combinator discards the result of a parser and makes it `Unit`.

## 4 Recursive Context-Free Parsers

Everything we've seen so far has been as powerful as a regular expression. While regular expressions are certainly useful for writing lexers they are normally not powerful enough to parse the syntax of a

programming language. It's worth nothing here that, *usually*, parser combinators don't make a distinction between lexing and parsing: you build your lexers out of the combinators and then use them in the parser. That has some advantages, but it does mean that backtracking is more expensive than it would otherwise be in a parser with a dedicated lexing phase. The reason this is considered good style is because of the higher-order nature of parser combinators: parsers are a first-class value that can be manipulated freely by the combinators, as opposed to more rigid grammar rules and terminals.

In this section, we'll explore how to extend the knowledge we've already built to start writing more complex parsers with recursive control-flow: this is sufficient to parse context-free grammars. It just takes the addition of the `flatMap` combinator to recover context-sensitive grammars from this, but grammars that require `flatMap` are rare in practice, so I won't touch on it here.

## 4.1 Recursion via Laziness

Writing recursive parsers is, fortunately, quite straight-forward but it does rely on Scala's *lazy value* feature. Let's start with the classic matching brackets parser (which cannot be parsed with regex: **here** comes to mind...):

```
import parsley.Parsley, Parsley.{eof, many}
import parsley.syntax.character.charLift

lazy val matching: Parsley[Unit] = many('(' *> matching <* ')').void
val onlyMatching = matching <* eof

onlyMatching.parse("") // succeeds
onlyMatching.parse("(") // fails
onlyMatching.parse("()") // succeeds
onlyMatching.parse("()()()") // succeeds
onlyMatching.parse("(((((())))") // succeeds
onlyMatching.parse("((((()))()()(()))") // succeeds
onlyMatching.parse("((((()))(()()(()))") // fails
```

There's a bit to unpack here! Firstly, the type ascription here on `matching` isn't optional: when we write recursive parsers, at least one thing in the recursive *group* needs to have a type, since Scala cannot infer the types of recursive functions (or in this case values). The `lazy` keyword here makes the `matching` parser only evaluated on demand. In this case that will be in the `onlyMatching` parser, which is only a `val`. The reason this is important is that it means that the reference to `matching` *inside* the parser isn't forced immediately, causing an infinite loop. That being said, every combinator in Parsley is defined using lazy function arguments (including a lazy `this` for methods), so sometimes it isn't actually necessary to get the right behaviour.

My advice is to use `lazy val` whenever you do write a parser that references itself, even indirectly. If your parser infinite loops before running it, you'll know you've missed one: but there are other, more obscure symptoms. Laziness is also important when you need to forward reference another parser. Here is an example:

```
lazy val q = ??? ~> p
lazy val p = ???

// vs

val p = ???
val q = ??? ~> p
```

Usually, the parsers can be re-ordered so that their definitions do not "cross over" each other in the words of scalac. But when writing recursive parsers with multiple parts you may need to use `lazy val` to get scalac to accept the ordering. This can also depend on whether or not they are defined locally in a function or as attributes of a class. Indeed, the above example is fine if `p` and `q` are both attributes of a class, even without `lazy val`!

**Much** more importantly, however, is noticing that when parsers *are* recursive, they absolutely **must** be references (i.e. `val`). A recursive parser using `def` *will*, without question, infinite loop. This is because Parsley will need the recursive point to be the same physical object as the original definition. When using a `def`, each time recursion is encountered it will create a new object and generate a truly infinite parser, instead of a cyclic one. We'll see an example of where we need to be careful about this later.

Before we move on with a more fleshed out example, I want to annotate the `matching` parser with `debug` to give you a sense of how recursive parsing works out (I marked both parentheses and the `matching` parser itself):

```
onlyMatchingDebug.parse("((()()))")
// >matching> (1, 1): ((()()
//                     ^
//   >left> (1, 1): ((()()
//                   ^
//   <left< (1, 2): ((()()) Good
//                    ^
//   >matching> (1, 2): ((()())
//                       ^
//     >left> (1, 2): ((()())
//                     ^
//     <left< (1, 3): ((()()))• Good
//                      ^
//     >matching> (1, 3): ((()()))•
//                         ^
//       >left> (1, 3): ((()()))•
//                       ^
//       <left< (1, 3): ((()()))• Fail
//                        ^
//     <matching< (1, 3): ((()()))• Good
//                         ^
//     >right> (1, 3): ((()()))•
//                      ^
```

```
//     <right< (1, 4): (()(()))• Good
//                      ^
//     >left> (1, 4): (()(()))•
//                     ^
//     <left< (1, 5): (()(()))• Good
//                     ^
//     >matching> (1, 5): (()(()))•
//                        ^
//       >left> (1, 5): (()(()))•
//                       ^
//       <left< (1, 6): (()(()))• Good
//                       ^
//       >matching> (1, 6): (()(()))•
//                          ^
//         >left> (1, 6): (()(()))•
//                         ^
//         <left< (1, 6): (()(()))• Fail
//                         ^
//       <matching< (1, 6): (()(()))• Good
//                          ^
//       >right> (1, 6): (()(()))•
//                       ^
//       <right< (1, 7): ()(()))• Good
//                       ^
//       >left> (1, 7): ()(()))•
//                      ^
//       <left< (1, 7): ()(()))• Fail
//                      ^
//     <matching< (1, 7): ()(()))• Good
//                        ^
//     >right> (1, 7): ()(()))•
//                     ^
//     <right< (1, 8): )(()))• Good
//                     ^
//     >left> (1, 8): )(()))•
//                    ^
//     <left< (1, 8): )(()))• Fail
//                    ^
//   <matching< (1, 8): )(()))• Good
//                      ^
//   >right> (1, 8): )(()))•
//                   ^
//   <right< (1, 9): (()))• Good
//                   ^
//   >left> (1, 9): (()))•
//                  ^
//   <left< (1, 9): (()))• Fail
//                  ^
// <matching< (1, 9): (()))• Good
//                    ^
// res52: parsley.Result[String, Unit] = Success(())
```

Take a moment to just absorb that and be comfortable with how recursive parsing works out.

## 4.2 Example: Parsing Boolean Expressions

The matching bracket parser was a simple example of a recursive parser. For our second to last example on this page (phew), we are going to parse (and evaluate!) boolean expressions with right associative operators. I'm going to start by giving the EBNF for this grammar to give a sense of what we are working with (and for you to be able to compare the approaches).

```
<expr> ::= <term> '||' <expr> | <term>
<term> ::= <not> '&&' <term> | <not>
<not> ::= '!' <not> | <atom>
<atom> ::= 'true' | 'false' | '(' <expr> ')'
```

We can see from this already it is a very recursive grammar, with almost every rule being recursive, as well as a recursive call to `<expr>` in `<atom>`. Now, it's perfectly possible to translate this grammar almost directly, but notice in `<expr>` and `<term>` that both alternatives in the grammar share a common leading prefix: as we identified earlier, this would require us to enable backtracking with `atomic` and will affect the time-complexity of the parse (here it would be exponential!). So, as a quick refactor, we will extract the common edge and represent the grammar as follows (where square brackets indicate optional component):

```
<expr> ::= <term> ['||' <expr>]
<term> ::= <not> ['&&' <term>]
<not> ::= '!' <not> | <atom>
<atom> ::= 'true' | 'false' | '(' <expr> ')'
```

Now, this grammar can be parsed in linear time, even when translated directly. This is much better! However, I'll make the inefficient parser first, as it has the simpler translation (even if it's less efficient) and will give a sense of how the solution works out.

```
import parsley.Parsley, Parsley.atomic
import parsley.syntax.character.stringLift
import parsley.syntax.lift.Lift2
import parsley.syntax.zipped.Zipped2

val or = (x: Boolean, y: Boolean) => x || y

// <expr> ::=        <term>  '||' <expr>  | <term>
lazy val expr: Parsley[Boolean] =
      atomic(or.lift(term <* "||", expr)) |  term

// <term> ::= <not> '&&'   <term>                | <not>
lazy val term: Parsley[Boolean] =
       atomic((not, "&&" ~> term).zipped(_ && _)) |  not

// <not> ::=                    '!'   <not>        | <atom>
```

```
lazy val not: Parsley[Boolean] = "!" *> not.map(!_) |  atom

// <atom> ::= 'true'          | 'false'           |  '('   <expr>   ')'
val atom    = "true".as(true) | "false".as(false) | ("(" ~> expr <~ ")")
```

```
expr.parse("!false")
// res53: parsley.Result[String, Boolean] = Success(true)
expr.parse("true&&!(false||true)")
// res54: parsley.Result[String, Boolean] = Success(false)
```

Here I've introduced a tiny bit of sugar: by importing `implicits.lift.Lift2`, I've enabled the `lift` method on two argument functions: essentially the same as `lift2` itself: `(or.lift _): (Parsley[Boolean], Parsley[Boolean]) => Parsley[Boolean]`. The `lift` is used to actually perform our desired actions: when we read `||` we want to actually combine the results with `||`! However, you'll notice I had to define it as a function with an explicit type signature: this is because Scala is reluctant to perform inference on the lambda when the argument types aren't known. To mitigate this, `implicits.zipped.Zipped2` provides the same functionality, but where `.zipped` is called on a tuple, notice how this means that a raw unannotated lambda can be used: this is because Scala has already learnt information about what types the arguments should have! Try to use the tupled `zipped` notation sparingly, however: the backwards application makes it a little trickier to notice the `,`s in the brackets. Try to only use it when you only have *small* parsers as arguments and `lift` when it works fine.

The parser itself has a close resemblance to the original grammar, just with the extra processing of the result. Notice, of course, that because `expr`, `term` and `not` are self-recursive, they all need explicit type signatures, and have been marked as `lazy`. This also allows me to use `atom` before it's defined in the lazy `not`. However, as I mentioned before, this is not ideal because of the heavy backtracking implied by the use of `atomic`. The solution, as I've said, is to implement the second grammar. This is, as we'll see, a little tricker:

```
import parsley.Parsley
import parsley.syntax.character.stringLift
import parsley.syntax.lift.Lift2
import parsley.combinator.option

val and = (y: Boolean) => (x: Boolean) => x && y

// This is possible here, because false is the "zero" of ||
// but more generally we'd want the other definition
// val or = (x: Boolean, y: Option[Boolean]) => x || y.getOrElse(false)
val or = (x: Boolean, y: Option[Boolean]) => y.foldLeft(x)(_ || _)

// <expr> ::=                          <term>       ['||'   <expr> ]
lazy val expr: Parsley[Boolean] = or.lift(term, option("||" ~> expr  ))

// <term> ::= <not>     ('&&'   <term>          |      epsilon      )
```

```
lazy val term: Parsley[Boolean] =
               not <**> ("&&" ~> term.map(and) </> identity[Boolean])

// <not> ::=                    '!'   <not>        | <atom>
lazy val not: Parsley[Boolean] = "!" ~> not.map(!_) | atom

// <atom> ::= 'true'          | 'false'           |  '('   <expr>   ')'
val atom    = "true".as(true) | "false".as(false) | ("(" ~> expr <~ ")")
```

```
expr.parse("!false")
// res56: parsley.Result[String, Boolean] = Success(true)
expr.parse("true&&!(false||true)")
// res57: parsley.Result[String, Boolean] = Success(false)
```

The new example is the more efficient, linear time, form of the parser. Here I've employed two different approaches of compiling the first `term`/`not` with the *possible* result after a *possible* operator. In the `expr` case, I've used a form very similar to our original parser, except by using the `option` combinator, I can try and parse what's inside and return `None` if it's not there. The `or` function will then have to handle both the case where it was only a `term` *and* the case where it was a `term` with another `expr`. Here I've used `Option.foldLeft` to do this, but there are many other ways of writing the function.

In the second case, with `term`, I've adopted an approach using the `<**>` combinator, which has the following type:

```
(_ <**> _): (Parsley[A], Parsley[A => B]) => Parsley[B]
```

It is essentially `<*>` but with the function and the argument reversed. So inside the brackets, I have to read a `term`, and if I'm successful I can apply the `and` function to it (notice the order of the arguments in the `and` function has been deliberately reversed and it has been curried). If I'm not successful I should return the identity function on `Boolean`, `identity[Boolean]`. The `p </> x` combinator is the same as saying `p <|> pure(x)`. This means our initial `not` result will either be applied to the identity function or the partially applied `and` function.

The reason I've shown both styles is so that you can decide for yourself which you prefer: `Option` or curried. This really isn't the best we could have done though! The page on building expression parsers will show you how you can write this parser without having to fiddle with the `and` and `or` functions at all! (spoiler: it involves a combinator that builds the expression parsers for you).

## 4.3 Higher-Order Example: Defining `many`

As a final exercise, I want to show how we can implement the `many` combinator: recall that it will execute its argument zero or more times and collect all the results. It's a nice exercise in how the concepts we've

already seen apply to an example where the parser we are constructing has a parameter of its own: in other words, a higher-order parser. It will also highlight a gotcha when writing your own combinators, just in case you become comfortable enough to do so.

The first step will be to think about what `many(p)` should do: if the parser `p` fails *without* consuming input, then we shouldn't fail but instead stop and return the results we've collected so far. If no `p`s were parsed, then the empty list should be returned. This gives us a hint about a use of `</>`, where we want to handle a failure by returning a known value. If a `p` is parsed, we need to try reading yet more `p`s, so this is an indication of recursion creeping in. So, with this in mind, let's see the definition:

```
import parsley.Parsley

// many p = p <:> many p <|> pure []
def many[A](p: =>Parsley[A]): Parsley[List[A]] = {
    lazy val go: Parsley[List[A]] = (p <::> go) </> Nil
    go
}
```

The definition isn't so complex, but comparing it with the Haskell equivalent in the comments does shed light on what extra things we need to be careful of here. The first is noticing that the argument `p` has been marked as `=>Parsley[A]`. This is because, like all of Parsley's combinators, we ideally want the argument to be lazy: this is what `=>A` means (except unlike Haskell, the argument isn't memoised). Then we can see the familar `lazy val` with explicit type signature that we expect from recursive parsers by now. What might seem a bit strange, however, is why I created the `go` value in the first place. You may be tempted to write something like this instead:

```
import parsley.Parsley

def many[A](p: =>Parsley[A]): Parsley[List[A]] = (p <::> many(p)) </> Nil
```

And the answer ties back to what I mentioned earlier: there is a difference in quite how recursive each of the two are. In the first example, `go` physically refers back to itself, and so that is a morally *finite* parser. The second example creates a new parser at each recursive step, so it is morally *infinite*. In Parsley, we must be careful to only work with *finite* parsers, as they are actually represented by Abstract Syntax Trees. So the solution here is to create a value that can reference the parameter `p`, without needing to pass it around itself. You might wonder if it's possible to make parsers that, say, have a value they pass around. The answer is yes, but it's quite uncommon to *need* to do that. For these circumstances, the functionality in `parsley.state` is useful, but this is certainly out of scope for this page!

### 4.3.1 Takeaways

- Recursive parsers are where the real work happens

- Using `lazy val` with any parser that is recursive is the safest way of writing them
- Recursive parsers need explicit types, as Scala can't infer them
- Parameterised recursion must be avoided: if the argument doesn't change then hoist it out!
- With expression grammars in particular, we should be mindful about the adverse effects of backtracking

## 5 What Next?

The next logical step once you've digested this page, is to go and have a play around yourself! When you feel ready, you should take a look at the **Building Expression Parsers** page to start seeing how recursive parsers can go wrong, and what the typical strategies are of addressing this.

# Building Expression Parsers

This page builds from the ground up on expression parsing. For a less discussion-based explanation see **precedence**, as well as **chain combinators** and **heterogeneous chain combinators** for more specific use-cases.

Expression parsing is a ubiquitous problem in parsing. It concerns the correct reading of operators and values, which are usually organised into precedence and fixities. For the purposes of this page a fixity will represent both the fixity *and* the associativity: infix-left, infix-right, prefix, and postfix. For example, this is a grammar for reading simple expressions consisting of numbers, brackets, addition, subtraction and multiplication with the standard precedences and left associativity.

```
<number> ::= <digit>+
<expr> ::= <expr> '+' <term> | <expr> '-' <term> | <term>
<term> ::= <term> '*' <atom> | <atom>
<atom> ::= '(' <expr> ')' | <number>
```

Here, the precedence is encoded by the fact that `<expr>` contains `<term>`, but `<term>` only contains `<atom>`. The `<expr>` on the left of the `'+'` denotes that addition binds more tightly to the left, which is what we expect.

## 1 The Problem with Left-Recursion

For a first atomic, let's directly translate this grammar into Parsley (for now, we'll parse into an `Int`: behold, the magic of combinators!):

```
import parsley.Parsley, Parsley.atomic
import parsley.character.digit
import parsley.syntax.character.charLift
import parsley.syntax.zipped.Zipped2

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

lazy val expr: Parsley[Int] =
  atomic((expr <* '+', term).zipped(_ + _)) |
  (expr <* '-', term).zipped(_ - _) |
  term
lazy val term: Parsley[Int] = (term <* '*', atom).zipped(_ * _) | atom
lazy val atom: Parsley[Int] = '(' *> expr <* ')' | number
```

This parser has a few glaring issues: for a start, the `atomic` is causing excessive backtracking! While there are ways to improve this, the real problem here is the *left-recursion*. Imagine you are evaluating this parser, first you look at `expr`, and then your first task is to evaluate `expr`! In fact, due to the strictness of Parsley's combinators, this example breaks before the parser runs: on Scala 2, it will `StackOverflowError` at runtime when constructing the parser, and on Scala 3, it will report an infinitely recursive definition for `expr` and `term`. The solution is to turn to the `chain` combinators, but before we do that, let's eliminate the atomics and refactor it a little to make the transition less jarring:

```
import parsley.Parsley, Parsley.atomic
import parsley.character.digit
import parsley.syntax.character.charLift

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

val add = (y: Int) => (x: Int) => x + y
val sub = (y: Int) => (x: Int) => x - y
val mul = (y: Int) => (x: Int) => x * y

lazy val expr: Parsley[Int] =
  atomic(expr <**> ('+'.as(add) <*> term)) |
  expr <**> ('-'.as(sub) <*> term) |
  term
lazy val term: Parsley[Int] = term <**> ('*'.as(mul) <*> atom) | atom
lazy val atom: Parsley[Int] = '(' ~> expr <~ ')' | number
```

The first step is to perform the translation from the previous post, where we make the operator result a function and apply that (flipped) to the right hand side (with `<*>`) and then the left (with `<**>`). Now, in this form, hopefully you can notice we've exposed the leading `expr` so that its on its own: now we can factor a bit more:

```
lazy val expr: Parsley[Int] =
  expr <**> (('+'.as(add) <*> term) | ('-'.as(sub) <*> term)) |
  term
```

Now we've eliminated the "backtracking" (if only we could make it that far!), but we can right factor the `|` too to obtain the simplest form for the parser:

```
lazy val expr: Parsley[Int] =
  expr <**> (('+'.as(add) | '-'.as(sub)) <*> term) |
  term
```

Now, at this point, I could demonstrate how to left-factor this grammar and produce something that is right recursive whilst preserving left-associativity. However, there isn't much point in doing this, as now we are in a good position to use the `chain.left1` combinator, which perfectly embodies the translation.

## 2 Using `expr.chain`

The left-recursion problem is not a new one, the parser combinator community has known about it for a long time. For parser combinator libraries it is necessary to *left-factor* the grammar. Thankfully, the left-factoring algorithm can be itself encoded nicely as a combinator: this is embodied by the `chain`-family. Here is the same example as before, but fixed using `chain.left1`:

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

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

val add = (x: Int, y: Int) => x + y
val sub = (x: Int, y: Int) => x - y

// chain.left1[A](p: Parsley[A], op: Parsley[(A, A) => A]): Parsley[A]
lazy val expr: Parsley[Int] = chain.left1(term, '+'.as(add) | '-'.as(sub))
lazy val term               = chain.left1[Int](atom, '*' as (_ * _))
lazy val atom               = '(' ~> expr <~ ')' | number
```

The structure of the parser is roughly the same, however now you'll notice that `expr` and `term` are no longer self-recursive, and neither `term` nor `atom` need to be lazy (or have explicit types). Just to illustrate, if we provide the type argument to `chain.left1` we can continue to use `_ * _`, but without it, we need explicit type signatures: see `add` and `sub`.

To make the relationship very clear between what we had before and what we have now, observe that the transformation from recursive to `chains` follows these shape:

```
self <**> (op <*> next) | next        == chain.left1(next, op)  // flipped op
self <**> op <*> next | next          == chain.left1(next, op)  // normal op
next <**> (op <*> self </> identity)  == chain.right1(next, op) // no
 backtracking, flipped
atomic(next <**> op <*> self) | next  == chain.right1(next, op) //
 backtracking, normal op
```

In this parser, the nesting of the chains dictates the precedence order (again, terms are found *inside* expressions and atoms *inside* terms). Since the addition and subtraction are on the same level, they belong

in the same chain. The `left1` indicates that the operator/s are left-associative and that there should be at least *one* of the next layer down. There are also `chain.right1`, `chain.prefix`, and `chain.postfix` combinators. The building of these parsers, however, is fairly mechanical, and it is tiresome to keep finding new names for new layers of the precedence table. For instances where there is more than one chain interacting together then `expr.precedence` comes in handy (but note that `expr.precedence` is complete overkill to replace a single chain!).

## 3 Using `expr.precedence`

The final form of this parser uses a expression parser builder, called `precedence`. Since Parsley parsers are implemented in pure Scala, there is nothing to stop you from developing tools like this yourself: the ability to work with parsers as values and develop combinators with them is the biggest advantage of the approach. That being said, most combinator libraries provide this sort of functionality out of the box and Parsley is no exception. Let's see the same parser one last time and see what's changed:

```
import parsley.Parsley
import parsley.character.digit
import parsley.syntax.character.charLift
import parsley.expr.{precedence, Ops, InfixL}

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

lazy val expr: Parsley[Int] = precedence[Int]('(' ~> expr <~ ')', number)(
    Ops(InfixL)('*' as (_ * _)),
    Ops(InfixL)('+' as (_ + _), '-' as (_ - _)))
```

This is a *lot* smaller! The way `precedence` works is that it is first provided with the `atom`s of the expression, and then each precedence level in turn (as many as needed), starting with the tightest binding operators. These levels are provided in the `Ops`, which take a fixity, and then any number of parsers which return functions matching the fixity given. Under the hood it will form the same nested chains that were used in the previous section. In essence, there is no practical difference between the two implementations.

The precedence table can actually also be reversed so that it works the other way round:

```
lazy val expr: Parsley[Int] = precedence[Int](
    Ops[Int](InfixL)('+' as (_ + _), '-' as (_ - _)),
    Ops[Int](InfixL)('*' as (_ * _)))(
    '(' ~> expr <~ ')', number)
```

But due to the ordering that type inference happens, this form is a bit more cumbersome.

As mentioned before, the fixity given to `Ops` influences what type the operators need to have. This works by a Scala feature called *path-dependent* typing, which is extraordinarily useful. If you want to know

more about this, see the relevant sub-section: you don't need to know about it or understand it to use `precedence`, however.

There is still a little more to this story though. So far we've been working with a homogenous datatype: every level in the precedence table shares the same type `Int`. Now, in an abstract syntax tree, which is the far more common result of parsing, you *could* represent all expressions homogenously (which I call a *monolithic* AST). But sometimes, it's desirable to maintain stronger guarantees about how the AST is structured, and for that we need a heterogenous precedence table.

## 3.1 Generalising `precedence` with `GOps`, `SOps`, and `Levels`

### 3.1.1 Subtyped ASTs with `SOps`

In some circumstances, it might be desirable to change the type of the parsers at each layer of the precedence table. This allows for a more strongly-typed AST, for example. Compared to Haskell, this can be easily achieved in Scala using subtyping.

For example, we can make an AST for our expressions like so:

```
sealed trait Expr
case class Add(x: Expr, y: Term) extends Expr
case class Sub(x: Expr, y: Term) extends Expr

sealed trait Term extends Expr
case class Mul(x: Term, y: Atom) extends Term

sealed trait Atom extends Term
case class Number(x: Int) extends Atom
case class Parens(x: Expr) extends Atom
```

The magic of subtyping means that `Number(10)` is a valid value of type `Expr`. That being said, we have a guarantee that an `Expr` can only be found inside a `Mul` if it is wrapped in `Parens`: since `Expr` is not a subtype of `Term` or `Atom`, `Mul(Add(Number(6), Number(7)), Number(8))` does *not* type-check!

Let's see what happens if we try and use our existing `precedence` knowledge with `Ops`:

```
val mul = (x: Expr, y: Expr) => Mul(x, y)
val add = (x: Expr, y: Expr) => Add(x, y)
val sub = (x: Expr, y: Expr) => Sub(x, y)

lazy val atom: Parsley[Atom] = number.map(Number) | '(' ~> expr.map(Parens)
 <~ ')'
lazy val expr = precedence[Expr](atom)(
  Ops(InfixL)('*' as mul),
  Ops(InfixL)('+' as add, '-' as sub))
// error: type mismatch;
```

```
//  found   : Expr
//  required: Term
// val mul = (x: Expr, y: Expr) => Mul(x, y)
//                                 ^
// error: type mismatch;
//  found   : Expr
//  required: Atom
// val mul = (x: Expr, y: Expr) => Mul(x, y)
//                                     ^
// error: type mismatch;
//  found   : Expr
//  required: Term
// val add = (x: Expr, y: Expr) => Add(x, y)
//                                     ^
// error: type mismatch;
//  found   : Expr
//  required: Term
// val sub = (x: Expr, y: Expr) => Sub(x, y)
//                                     ^
```

The problem is that, though all `Term`s are `Expr`s (and ditto for `Atom`), we are forced to create operators of the shape `(Expr, Expr) => Expr` to fit into the precedence table, but we can't guarantee that those `Expr`s we are passing into the function are actually `Term`s (even though we know intuitively that they will be). In other words, `(Term, Atom) => Term` is not a subtype of `(Expr, Expr) => Expr`!

So, how do we fix this? Well, we need to stop using `Ops` and use `SOps`: instead of requiring an `(A, A) => A` operator for `InfixL`, `SOps` will demand those with shape `(B, A) => B` such that `A <: B`. So, does our `(Term, Atom) => Term` match this type? Yes: `A = Atom`, `B = Term` and `Atom <: Term`; all is good. Why do we require that `A <: B` exactly? Well, consider that we didn't read any multiplication operators, then we are going to be handing just an `Atom` to the layer above, but we are making the claim that we produce `Term`s. Of course, this is ok because `Atom`s are themselves `Term`s.

Unfortunately, we can't just provide `SOps` as variadic arguments to the combinator, since they all have different types to each other (that is the point, after all). Instead we use a heterogenous list of precedence levels called, well, `Levels`.

```
trait Levels[+A]
case class Atoms[+A](atoms: Parsley[A]*) extends Levels[A]
// and
case class Level[A, B](
    nextLevels: Levels[A],
    ops: Ops[A, B])
  extends Levels[B]
```

Basically, the type parameter to `Levels` is saying that we *produce* values of type `A` from the outer-most level in the structure. There are two choices of constructor in the list: `Atoms` is the end of the list, it produces `A`s. The equivalent to `::`, `Level[A, B]` is a bit more complex: it says that, if you give it a precedence table that produces `A`s, then it can use its own operators that work on `A` to produce values of type `B`. As a result, the larger table produces `B`s.

Now, to make life nicer for us, the `Levels` list supports the common-place Scala collections operators of `+:` and `:+`, which can be used in place of `Level`. Just like other Scala collections, the rest of the table appears on the side of the `:`. As a result, we can build tables like:

```
Atoms(atom1, atom2, .., atomN) :+ ops1 :+ ops2 :+ .. :+ opsN
// or
opsN +: .. +: ops2 +: ops1 +: Atoms(atom1, atom2, .., atomN)
```

The first form is the tightest first approach, and the second is the weakest first approach. So, what does our parser look like if we use `Levels` and `SOps`?

```
import parsley.Parsley
import parsley.character.digit
import parsley.syntax.character.charLift
import parsley.expr.{precedence, SOps, InfixL, Atoms}

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

sealed trait Expr
case class Add(x: Expr, y: Term) extends Expr
case class Sub(x: Expr, y: Term) extends Expr

sealed trait Term extends Expr
case class Mul(x: Term, y: Atom) extends Term

sealed trait Atom extends Term
case class Number(x: Int) extends Atom
case class Parens(x: Expr) extends Atom

lazy val expr: Parsley[Expr] = precedence {
  Atoms(number.map(Number), '(' ~> expr.map(Parens) <~ ')') :+
  SOps(InfixL)('*' as Mul) :+
  SOps(InfixL)('+' as Add, '-' as Sub)
}
```

Not so bad! We've constructed the `Levels` list using `:+`, so this is strongest-first. This time, if we turn the list around it isn't going to make us need to add type-annotations like it did when we turned the `Ops` based table round earlier. Nice! An extra advantage of using this approach now is that if we tried to use `InfixR` instead of `InfixL`, this happens:

```
type mismatch;
    found   : (Term, Atom) => Term
    required: (Atom, Term) => Term
```

This means that, by using `SOps`, we get a guarantee that our parser correctly matches the intended associativity advertised by our ASTs constructors!

### 3.1.2 Non-Subtyped Heterogenous ASTs with `GOps`

So far we've seen how to generalise our expression parsers to work with heterogenous trees that rely on subtyping. However, there may be cases where the subtyping is undesirable, or otherwise not possible (for example, if you want layers from `Int` to `Expr`) but we still want these strongly typed guarantees about the shape of the tree. In this case we would change the data-type as follows:

```
sealed trait Expr
case class Add(x: Expr, y: Term) extends Expr
case class Sub(x: Expr, y: Term) extends Expr
case class OfTerm(t: Term) extends Expr

sealed trait Term
case class Mul(x: Term, y: Atom) extends Term
case class OfAtom(x: Atom) extends Term

sealed trait Atom
case class Number(x: Int) extends Atom
case class Parens(x: Expr) extends Atom
```

Now the question is, how do we use the `precedence` parser now? The types of each of these constructors no longer match `(B, A) => B` with `A <: B`! This is where `GOps` comes in. It's very similar to `SOps`, except it doesn't come with the constraint that `A` is a subtype of `B`. Instead, a `GOps` constructor requires you to provide a function of type `A => B` too! In our case, these will correspond to the `OfAtom` and `OfTerm` functions from above. Note that, if there are any implicit conversion available from `A` to `B`, `GOps` will happily use those (this includes the implicit conversions called `A =:= A` and `A <:< B` for type equality and subtyping respectively: `GOps` can implement the behaviour of `Ops` and `SOps` via these conversions). So, what does this look like in practice?

```
import parsley.Parsley
import parsley.character.digit
import parsley.syntax.character.charLift
import parsley.expr.{precedence, GOps, InfixL, Atoms}

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

lazy val expr: Parsley[Expr] = precedence {
  Atoms(number.map(Number), '(' ~> expr.map(Parens) <~ ')') :+
  GOps[Atom, Term](InfixL)('*' as Mul)(OfAtom) :+
  GOps[Term, Expr](InfixL)('+' as Add, '-' as Sub)(OfTerm)
}
```

Not so different from the original using `SOps`, but if you can allow subtyping in your AST, you can use the much less brittle `SOps` form. What makes it brittle? Well, notice that this time we've had to manually specify the types that each level deals with: this is because, without a subtyping constraint, Scala is reluctant to make `Mul` be of type `(Term, Atom) => Term`. Instead it makes it `(Term, Atom) => Mul` and complains that `OfAtom` hasn't got type `Atom => Mul`. Oops!

By the way, you can actually intermingle `Ops`, `SOps`, *and* `GOps` all in the same table, just as long as you are using `Levels`. Each of them are just builders for values of type `Ops[A, B]`.

## 3.2 Path-Dependent Typing and `Ops`/`SOps`/`GOps`

To support the advertised behaviour that the type of an operator depends on the fixity it has, the `Fixity` trait has an *abstract type* called `Op`. Let's take the machinery behind the simpler `Ops` as an example.

```
sealed trait Fixity {
    type Op[A, B]
}

object Ops {
    def apply[A](fixity: Fixity)(ops: Parsley[fixity.Op[A, A]]*): Ops[A, A]
 = ???
}
```

This is saying that the types of the parsers we pass to a call to `Ops.apply` should depend on the type of the `Op` supported by the `fixity`. For instance, let's take `InfixL` and `Prefix`:

```
case object InfixL extends Fixity {
    override type Op[-A, B] = (B, A) => B
}
case object Prefix extends Fixity {
    override type Op[A, B] = B => B
}
```

Why `Op` works with `A`s and `B`s is explained in the very last subsection, so for now just always assume that `A =:= B`. Now observe the types of the partial applications of `Ops.apply` to the different fixities:

```
def infixLefts[A](ops: Parsley[(A, A) => A]*): Ops[A, A] =
  Ops(InfixL)(ops: _*)
def prefixes[A](ops: Parsley[A => A]*): Ops[A, A] =
  Ops(Prefix)(ops: _*)
```

The path-dependent type of `fixity.Op[A, A]` allows the types of the parsers to change accordingly. There is a similar story for the `GOps` and `SOps` objects, but they instead rely on `Levels` as opposed to variadic arguments.

## 3.3 Afternote: Why `(B, A) => B` and `B => B`?

The types given to each fixity are as follows:

- `InfixL` is `(B, A) => B`
- `InfixR` is `(A, B) => B`
- `Prefix` and `Postfix` are both `B => B`

This might seem confusing at first: why, for instance, do the unary operators not mention `A` at all? Well, let's first understand why `(B, A) => B` is appropriate for left-associative things but not right ones.

```
sealed trait Expr
case class LOp(x: Expr, y: Int) extends Expr
case class ROp(x: Int, y: Expr) extends Expr
case class Number(x: Int) extends Expr
```

Notice that `LOp(LOp(Number(6), 5), 4)` is ok, because the right hand argument to `LOp` is always an `Int` and the left-hand argument is always an expression. In the case of `6`, `Int` is not an `Expr`, so we wrap it up in the `Number` constructor. So, for `LOp`, if we take `A = Int` and `B = Expr`, it has the shape `(B, A) => B`. On the other hand, `ROp(ROp(Number(6), 5), 4)` is not ok, because `ROp(...)` is not an `Int`! This justifies the `(A, B) => B` type: like-expressions can appear on the right, but not the left. The level for this would

be `GOp[Int, Expr](InfixL('@'.as(LOp))(Number)` or `GOp[Int, Expr](InfixR('@'.as(ROp))(Number)` (notice that switching them round wouldn't type-check!)

For `Prefix` and `Postfix` it's a similar story:

```
sealed trait BoolExpr
case class Not(x: BoolExpr) extends BoolExpr
case class Literal(b: Boolean) extends BoolExpr
```

We would like to be able to write `Not(Not(Literal(False)))`, which means that `Not` needs to accept the same type as itself. This explains `B => B`, and in this case, the booleans themselves need to be wrapped up with `Literal` of shape `A => B`. This is the same role of `Number` before, which also has shape `A => B`. The level for this would be `GOp[Boolean, BoolExpr]("not" #> Not)(Literal)`.

# Effective Whitespace Parsing

Note that this page describes conventions for whitespace handling, but these are handled directly by the `parsley.token.Lexer` and its subfunctionality, which is expored at the end of the next page.

Previously, in **Basics of Combinators** and **Building Expression Parsers**, we've seen parsers for languages that do not account for whitespace. In this page I'll discuss the best practices for handling whitespace in your grammars.

## 1 Defining whitespace readers

The first step in the correct handling of whitespace is to build the small parsers that recognise the grammar itself. The two concerns usually are spaces and comments. For comments, the combinator `combinator.manyTill` is *very* useful. For example:

```
import parsley.Parsley, Parsley.{atomic, many}
import parsley.character.{whitespace, string, item, endOfLine}
import parsley.combinator.manyTill
import parsley.errors.combinator.ErrorMethods //for hide

def symbol(str: String): Parsley[String] = atomic(string(str))

val lineComment = symbol("//") ~> manyTill(item, endOfLine).void
val multiComment = symbol("/*") ~> manyTill(item, symbol("*/")).void
val comment = lineComment | multiComment

val skipWhitespace = many(whitespace.void | comment).void.hide
```

Here, the `manyTill` combinator is used to read up until the end of the comment. You may notice the `hide` method having been called on `skipWhitespace`. This handy operation hides the "expected" error message from a given parser. In other words, when we have a parse error, it isn't particularly useful to see in the suggestions of what would have worked that we could type some whitespace! Producing informative and tidy error messages, however, is a more general topic for another post. Now that we have the `skipWhitespace` parser we can start using it!

## 2 Lexemes

Lexemes are indivisible chunks of the input, the sort usually produced by a lexer in a classical setup. The `symbol` combinator I defined above forms part of this: it uses `atomic` to make an indivisible string, either

you read the entire thing or none of it. The next piece of the puzzle is a combinator called `lexeme`, which should perform a parser and then always read spaces after it:

```
def lexeme[A](p: Parsley[A]): Parsley[A] = p <~ skipWhitespace
def token[A](p: Parsley[A]): Parsley[A] = lexeme(atomic(p))

implicit def implicitSymbol(s: String): Parsley[String] = lexeme(symbol(s))
```

The `token` combinator is a more general form of `symbol`, that works for all parsers, handling them atomically *and* consuming whitespace after. Note that it's important to consume the whitespace outside the scope of the `atomic`, otherwise malformed whitespace might cause backtracking for an otherwise legal token!

With the `implicitSymbol` combinator, we can now treat all string literals as lexemes. This can be very useful, but ideally this could be improved by also recognising whether or not the provided string is a keyword, and if so, ensuring that it is not followed by another alphabetical character. This is out of scope for this post, however.

Now let's take the example from **Building Expression Parsers** and see what needs to change to finish up recognising whitespace.

```
import parsley.character.digit
import parsley.expr.{precedence, Ops, InfixL}

val number = token(digit.foldLeft1[BigInt](0)((n, d) => n * 10 + d.asDigit))
// number: Parsley[BigInt] = parsley.Parsley@441519b9

lazy val atom: Parsley[BigInt] = "(" ~> expr <~ ")" | number
lazy val expr = precedence[BigInt](atom)(
  Ops(InfixL)("*" as (_ * _)),
  Ops(InfixL)("+" as (_ + _), "-" as (_ - _)))
```

Other than introducing our new infrastructure, I've changed the characters in the original parser to strings: this is going to make them use our new `implicitLexeme` combinator! Notice how I've also marked the *whole* of `number` as a token: we don't want to read whitespace between the digits, but instead after the entire number has been read, and a number should be entirely atomic. Now that we've done this we can try running it on some input and see what happens:

```
expr.parse("5 + \n6 /*hello!*/ * 7")
// res0: parsley.Result[String, BigInt] = Success(47)

expr.parse(" 5 * (\n2 + 3)")
// res1: parsley.Result[String, BigInt] = Failure((line 1, column 1):
//   unexpected space
//   expected "(" or digit
//   > 5 * (
//     ^
//   >2 + 3))
```

Ah, we've forgotten one last bit! The way we've set it up so far is that every lexeme reads whitespace *after* the token. This is nice and consistent and reduces any unnecessary extra work reading whitespace before and after a token (which inevitably means whitespace will be unnecessarily checked in between tokens *twice*). But this means we have to be careful to read whitespace once at the very beginning of the parser. Using `skipWhitespace ~> expr` as our parser we run is the final step we need to make it all work. If we use `expr` in another parser, however, we don't want to read the whitespace at the beginning in that case. It should **only** be at the very start of the parser (so when `parse` is called).

## 3 A Problem with Scope

The eagle-eyed reader might have spotted that there is a distinction between the string literals we are using in the main parser and the `symbol`s we are using in the definitions of whitespace. Indeed, because we are using an implicit that consumes whitespace, it would be inappropriate to use it in the *definition* of whitespace! If we were to pull in the `stringLift` implicit as we're used to, then Scala will report and ambiguous implicit and we'll be stuck. It's a *much* better idea to limit the scope of these implicits, so we can be clear about which we mean where. To illustrate what I mean, let's restructure the code a little for the parser and ensure we don't run into any issues.

```
import parsley.Parsley, Parsley.{atomic, eof, many}
import parsley.character.{digit, whitespace, string, item, endOfLine}
import parsley.combinator.manyTill
import parsley.expr.{precedence, Ops, InfixL}
import parsley.errors.combinator.ErrorMethods //for hide

object lexer {
    private def symbol(str: String): Parsley[String] = atomic(string(str))

    private val lineComment = symbol("//") ~> manyTill(item, endOfLine).void
    private val multiComment = symbol("/*")
 ~> manyTill(item, symbol("*/")).void
    private val comment = lineComment | multiComment
    private val skipWhitespace = many(whitespace.void | comment).void.hide

    private def lexeme[A](p: Parsley[A]): Parsley[A] = p <~ skipWhitespace
    private def token[A](p: Parsley[A]): Parsley[A] = lexeme(atomic(p))
    def fully[A](p: Parsley[A]): Parsley[A] = skipWhitespace ~> p <~ eof
```

```
    val number = token(digit.foldLeft1[BigInt](0)((n, d) => n * 10
 + d.asDigit))

    object implicits {
        implicit def implicitSymbol(s: String): Parsley[String] =
            lexeme(symbol(s)) // or `token(string(s))
    }
}

object expressions {
    import lexer.implicits.implicitSymbol
    import lexer.{number, fully}

    private lazy val atom: Parsley[BigInt] = "(" ~> expr <~ ")" | number
    private lazy val expr = precedence[BigInt](atom)(
        Ops(InfixL)("*" as (_ * _)),
        Ops(InfixL)("+" as (_ + _), "-" as (_ - _)))

    val parser = fully(expr)
}
```

In the above refactoring, I've introduced three distinct scopes: the `lexer`, the `lexer.implicits` and the `expressions`. Within `lexer`, I've marked the internal parts as `private`, in particular the `implicitSymbol` combinator that I've introduced to allow the lexer to use string literals in the description of the tokens. By marking `implicitSymbol` as `private`, we ensure that it cannot be accidentally used within `expressions`, where the main part of the parser is defined. In contrast, the `implicits` object nested within `lexer` provides the ability for the `expressions` object to hook into our whitespace sensitive string literal parsing (using `implicitToken`), and, but enclosing it within the object, we prevent it being accidentally used inside the rest of the lexer (without an explicit import, which we know would be bad!). This is a good general structure to adopt, as it keeps the lexing code cleanly separated from the parser. If, for instance, you wanted to test these internals, then you could leave them public, but I would advise adding a private to its internal implicits at *all* times (however, ScalaTest does have the ability to test `private` members!)

# Effective Lexing

Parsley offers a user-friendly API for lexing, which is detailed at the end of this document. We recommend using this API for most purposes.

The initial sections of this document explain the fundamental principles behind lexing with parser combinators.

In the previous post, we saw the basic principles behind handling whitespace in a transparent manner. To remind ourselves of what we ended up lets pick up where we left off:

```
import parsley.Parsley, Parsley.{atomic, eof, many}
import parsley.character.{digit, whitespace, string, item, endOfLine}
import parsley.combinator.manyTill
import parsley.expr.{precedence, Ops, InfixL}
import parsley.errors.combinator.ErrorMethods

object lexer {
    private def symbol(str: String): Parsley[String] = atomic(string(str))

    private val lineComment = symbol("//") ~> manyTill(item, endOfLine).void
    private val multiComment = symbol("/*")
 ~> manyTill(item, symbol("*/")).void
    private val comment = lineComment | multiComment
    private val skipWhitespace = many(whitespace.void | comment).void.hide

    private def lexeme[A](p: Parsley[A]): Parsley[A] = p <~ skipWhitespace
    private def token[A](p: Parsley[A]): Parsley[A] = lexeme(atomic(p))
    def fully[A](p: Parsley[A]): Parsley[A] = skipWhitespace ~> p <~ eof

    val number = token(digit.foldLeft1[BigInt](0)((n, d) => n * 10
 + d.asDigit))

    object implicits {
        implicit def implicitSymbol(s: String): Parsley[String]
 = lexeme(symbol(s))
    }
}

object expressions {
    import lexer.implicits.implicitSymbol
    import lexer.{number, fully}

    private lazy val atom: Parsley[BigInt] = "(" ~> expr <~ ")" | number
    private lazy val expr = precedence[BigInt](atom)(
        Ops(InfixL)("*" as (_ * _)),
        Ops(InfixL)("+" as (_ + _), "-" as (_ - _)))
```

```
    val parser = fully(expr)
}
```

So far, we've broken the parser into two distinct chunks: the lexer and the main parser. Within the lexer we need to be careful and *explicit* about where we handle whitespace and where we don't; within the parser we can assume that all the whitespace has been correctly dealt with and can focus on the main content. To help motivate the changes we are going to make to the lexer object later on in the post, I want to first extend our "language" to add in variables into the language and a `negate` operator. In the process I'm going to swap the integer result for an abstract syntax tree.

```
import parsley.expr.{precedence, Ops, InfixL, Prefix}
object expressions {
    import lexer.implicits.implicitSymbol
    import lexer.{number, fully, identifier}
    // for now, assume that `identifier` is just 1 or more alphabetical
 characters

    sealed trait Expr
    case class Add(x: Expr, y: Expr) extends Expr
    case class Mul(x: Expr, y: Expr) extends Expr
    case class Sub(x: Expr, y: Expr) extends Expr
    case class Neg(x: Expr) extends Expr
    case class Num(x: BigInt) extends Expr
    case class Var(x: String) extends Expr

    private lazy val atom: Parsley[Expr] =
        "(" ~> expr <~ ")" | number.map(Num) | identifier.map(Var)
    private lazy val expr = precedence[Expr](atom)(
        Ops(Prefix)("negate" as Neg),
        Ops(InfixL)("*" as Mul),
        Ops(InfixL)("+" as Add, "-" as Sub))

    val parser = fully(expr)
}
```

Now, we can assume that, since `identifier` comes from the lexer, this parser handles all the whitespace correctly. The question is, does it work?

```
expressions.parser.parse("x + y")
// res1: parsley.Result[String, expressions.Expr] = Success(Add(Var(x),Var(y)))

expressions.parser.parse("negate + z")
// res2: parsley.Result[String, expressions.Expr] = Failure((line 1, column 8):
//   unexpected "+"
//   expected "(", "negate", digit, or letter
//   >negate + z
//           ^)
```

```
expressions.parser.parse("negate x + z")
// res3: parsley.Result[String, expressions.Expr] =
 Success(Add(Neg(Var(x)),Var(z)))

expressions.parser.parse("negatex + z")
// res4: parsley.Result[String, expressions.Expr] =
 Success(Add(Neg(Var(x)),Var(z)))
```

So, looking at these examples, the first one seems to work fine. The second one also works fine, but given that we've said that identifiers are just alpha-numeric characters, you might assume it was legal (indeed, it really *shouldn't* be legal in most languages that don't have "soft" keywords). The third example again works as intended, but the fourth is suspicious: `negatex` is clearly an identifier but was parsed as `negate x`! This now gives us a set up for refining our lexer for the rest of the page. We won't be touching the `expressions` object again, so take a long hard look at it.

## 1 Keywords, Identifiers and Operators

The crux of the problem we unearthed in the last section is that the implicit used to handle strings has no awareness of keywords (or indeed operators) and identifiers that work for *any* alpha-numeric sequence are an accident waiting to happen. Let's start by creating a couple of sets to represent the valid keywords and operators in the language:

```
val keywords = Set("negate")
val operators = Set("*", "+", "-")
```

Now we can use these to define some more specialist combinators for dealing with these lexemes:

- We want to ensure that identifiers are not valid keywords.
- We want to ensure reading a keyword does not have a valid identifier "letter" after it.
- We want to ensure that a specific operator does not end up being the prefix of another, parsable, operator. This satisfies the "maximal-munch" style of parsing.

We'll start with `identifier`: to check that an identifier we've just read is not itself a valid keyword we can use the `filter` family of combinators. In particular, `filterOut` provides an error message that explains why the parser has failed. Here is our new and improved identifier:

```
import parsley.errors.combinator.ErrorMethods //for filterOut
import parsley.character.{letter, stringOfSome}

// `stringOfSome(letter)` is loosely equivalent to
 `some(letter).map(_.mkString)`
val identifier = token(stringOfSome(letter).filterOut {
    case v if keywords(v) => s"keyword $v may not be used as an identifier"
})
```

The `filterOut` combinator takes a `PartialFunction` from the parser's result to `String`. If the partial function is defined for its input, that produces the error message that the parser fails with. Notice that I've been very careful to make sure the filter happens *inside* the scope of the `token` combinator. If we do read an identifier and then rule it out because its a keyword, we want the ability to backtrack. Filtering after the input has been irrevocably consumed will mean this parser fails more strongly than it should.

Next we'll tackle the keywords, using the handy `notFollowedBy` combinator that was briefly referenced in the very first post:

```
import parsley.Parsley.notFollowedBy
def keyword(k: String): Parsley[Unit] = token(string(k)
 ~> notFollowedBy(letter))
```

Again, notice that I've been very careful to perform the negative-lookahead within the scope of the `token` so that we can backtrack if necessary (indeed, it's likely that a valid alternative was an identifier!). Additionally, we also need to ensure that whitespace isn't read *before* we try and check for the `letterOrDigit`, otherwise `negate x` would *also* fail, so that's another reason to keep it within `token`.

Finally, let's look at how operator is dealt with. It's a bit trickier, and in this case is meaningless, because all of our operators are single character and don't form valid prefixes of each other. But it will be useful to discuss anyway:

```
import parsley.character.strings

def operator(op: String): Parsley[Unit] = {
    val biggerOps = operators.collect {
        case biggerOp if biggerOp.startsWith(op)
                      && biggerOp > op => biggerOp.substring(op.length)
    }.toList
    biggerOps match {
        case Nil => lexeme(symbol(op)).void
        // strings requires one non-varargs argument
        case biggerOp :: biggerOps =>
            token(string(op)
 ~> notFollowedBy(strings(biggerOp, biggerOps: _*)))
    }
}
```

Let's unpack what's going on here: first we read the op as normal, then we ensure that it's not followed by the *rest* of any operators for which it forms a valid prefix. This is using the regular `collect` method on Scala `Set`. As an example, if we have the operator set `Set("*", "**", "*-", "++")` and we call `operator("*")`, we would be checking `notFollowedBy(strings("+", "-"))`, since `**` and `*-` are both prefixed by `*`. This is, again, a great example of how powerful access to regular Scala code in our parsers is! This would be quite tricky to define in a parser generator!

So, the question is, what do we do with our new found combinators? We could just expose them to the rest of the parser as they are, but that leaves room for error if we forget, or miss out, any of the replacements. And, in addition, we lose the nice string literal syntax we've made good use of until this point. So, a better solution would be to change our definition of `implicitSymbol`:

```
object implicits {
    implicit def implicitSymbol(s: String): Parsley[Unit] = {
        if (keywords(s))       keyword(s)
        else if (operators(s)) operator(s)
        else                   lexeme(symbol(s)).void
    }
}
```

Now, when we use a string literal in our original parser, it will first check to see if that is a valid keyword or an operator and, if so, it can use our specialised combinators: neat! With this done, let's see what the new lexer looks like and relook at the problematic example:

```
object lexer {
    private val keywords = Set("negate")
    private val operators = Set("*", "+", "-")

    private def symbol(str: String): Parsley[String] = atomic(string(str))

    private val lineComment = symbol("//") ~> manyTill(item, endOfLine).void
    private val multiComment = symbol("/*")
 ~> manyTill(item, symbol("*/")).void
    private val comment = lineComment | multiComment
    private val skipWhitespace = many(whitespace.void | comment).void.hide

    private def lexeme[A](p: Parsley[A]): Parsley[A] = p <~ skipWhitespace
    private def token[A](p: Parsley[A]): Parsley[A] = lexeme(atomic(p))
    def fully[A](p: Parsley[A]): Parsley[A] = skipWhitespace ~> p <~ eof

    val number = token(digit.foldLeft1[BigInt](0)((n, d) => n * 10
 + d.asDigit))
    val identifier = token(stringOfSome(letter).filterOut {
        case v if keywords(v) => s"keyword $v may not be used as an identifier"
    })

    private def operator(op: String): Parsley[Unit] = {
        val biggerOps = operators.collect {
            case biggerOp if biggerOp.startsWith(op)
                          && biggerOp > op => biggerOp.substring(op.length)
        }.toList
        biggerOps match {
            case Nil => lexeme(symbol(op)).void
            // strings requires one non-varargs argument
            case biggerOp :: biggerOps =>
                token(string(op)
 ~> notFollowedBy(strings(biggerOp, biggerOps: _*)))
```

```
        }
    }

    private def keyword(k: String): Parsley[Unit] =
        token(string(k) ~> notFollowedBy(letter))


    object implicits {
        implicit def implicitSymbol(s: String): Parsley[Unit] = {
            if (keywords(s))       keyword(s)
            else if (operators(s)) operator(s)
            else                   lexeme(symbol(s)).void
        }
    }
}
```

And the original failing example:

```
expressions.parser.parse("negatex + z")
// res8: parsley.Result[String, expressions.Expr] =
 Success(Add(Var(negatex),Var(z)))
```

Exactly as desired!

## 2 Using `token.descriptions.LexicalDesc` and `token.Lexer`

Whilst everything we have done above is nice and instructive, in practice all this work is already done for us with **`token.Lexer`**. By providing a suitable **`token.descriptions.LexicalDesc`**, we can get a whole bunch of combinators for dealing with tokens for free. There is a lot of functionality found inside the **`Lexer`**, and most of it is highly configurable with the **`LexicalDesc`** and its sub-components. Let's make use of this new found power and change up our **`lexer`** object one last time:

```
object lexer {
    import parsley.token.{Lexer, predicate}
    import parsley.token.descriptions.{LexicalDesc, NameDesc, SymbolDesc}

    private val desc = LexicalDesc.plain.copy(
        nameDesc = NameDesc.plain.copy(
            // Unicode is also possible instead of Basic
            identifierStart = predicate.Basic(_.isLetter),
            identifierLetter = predicate.Basic(_.isLetter),
        ),
        symbolDesc = SymbolDesc.plain.copy(
            hardKeywords = Set("negate"),
            hardOperators = Set("*", "+", "-"),
        ),
    )
```

```
    private val lexer = new Lexer(desc)

    val identifier = lexer.lexeme.names.identifier
    val number = lexer.lexeme.natural.decimal

    def fully[A](p: Parsley[A]) = lexer.fully(p)
    val implicits = lexer.lexeme.symbol.implicits
}
```

The `implicitSymbol` function we developed before, along with `operator` and `keyword` are all implemented by `lexer.lexeme.symbol`. The `names.identifier` parser accounts for the keyword problem for us. The basic `natural.decimal` parser meets our needs without any additional configuration: it also returns `BigInt`, which is arbitrary precision. By using `token.lexeme`, this will already handle the whitespace and atomicity of the token for us. This is just the tip of the iceberg when it comes to the lexer functionality within Parsley. It is well worth having a play around with this functionality and getting used to it!

A more detailed description of this functionality can be found in the **API Guide**.

# The Parser Bridge Pattern

The first part of this page helps to motivate the *Parser Bridge* pattern, and the second part shows how to implement it from scratch. This is useful to know, but the API Guide **Generic Bridges** page can get you started with the technique faster. The latter parts of this page can be helpful when the generic bridges no longer suffice.

By this point, we've seen how to effectively build expression parsers, lexers, and how to handle whitespace in a clean way. However, something we've not touched on yet is how to encode the position information into any data-types produced by our parsers. In fact, the way we can build our results from our parsers can be greatly improved. We'll focus on expanding the same parser from the previous pages, since in its current form it has a variety of different types of constructors. What I will do, however it expand it with some basic `let`-binding expressions. We'll use the same `lexer` object as before, but I will add the keywords `let` and `in` to the keyword set. Previously, the grammar we were working with would have been:

```
<number>     ::= ...
<identifier> ::= ...
<atom>       ::= '(' <expr> ')' | <number> | <identifier>
<negated>    ::= 'negate' <negated> | <atom>
<term>       ::= <term> '*' <atom> | <atom>
<expr>       ::= <expr> ('+' | '-') <term> | <term>
```

We'll extend this to include a let syntax as follows:

```
<let-binding> ::= 'let' <bindings> 'in' <expr> | <expr>
<bindings>    ::= <binding> [';' [<bindings>]]
<binding>     ::= <identifier> '=' <let-binding>
```

This will allow us to write programs like:

```
let x = 10;
    y = let z = x + 4 in z * z;
in x * y
```

Now let's see how this changes the parser:

```
import parsley.Parsley

object ast {
    sealed trait LetExpr
    case class Let(bindings: List[Binding], x: Expr) extends LetExpr
    case class Binding(v: String, x: LetExpr)
```

```
    sealed trait Expr extends LetExpr
    case class Add(x: Expr, y: Expr) extends Expr
    case class Mul(x: Expr, y: Expr) extends Expr
    case class Sub(x: Expr, y: Expr) extends Expr
    case class Neg(x: Expr) extends Expr
    case class Num(x: BigInt) extends Expr
    case class Var(x: String) extends Expr
}

object expressions {
    import parsley.expr.{precedence, Ops, InfixL, Prefix}
    import parsley.combinator.sepEndBy1
    import parsley.syntax.lift.Lift2

    import lexer.implicits.implicitSymbol
    import lexer.{number, fully, identifier}
    import ast._

    lazy val atom: Parsley[Expr] =
        "(" ~> expr <~ ")" | number.map(Num) | identifier.map(Var)
    lazy val expr = precedence[Expr](atom)(
        Ops(Prefix)("negate" as Neg),
        Ops(InfixL)("*" as Mul),
        Ops(InfixL)("+" as Add, "-" as Sub))

    lazy val binding = Binding.lift(identifier, "=" ~> letExpr)
    lazy val bindings = sepEndBy1(binding, ";")
    lazy val letExpr: Parsley[LetExpr] =
      Let.lift("let" ~> bindings, "in" ~> expr) | expr

    val parser = fully(letExpr)
}
```

So far, so good. I've added a couple of now nodes to the AST, and three extra parser definitions. The only new thing here is the helpful `sepEndBy1` combinator, which is particularly good (along with its cousins, `sepBy1` and `endBy1`) at dealing with things like commas and semi-colons. However, if I now said that we need to encode position information into our language's AST then things are going to need to change.

Let's start by adding the information into the AST. There is a trick to this depending on whether or not we want the information to be visible during pattern matches or not. Essentially, in Scala, if a second (or third etc) set of arguments is added to a `case class`, these arguments will not appear in the pattern match, but *are* required to build an instance. So we're going to add an extra argument to each constructor containing the position information like so:

```
object ast {
    sealed trait LetExpr
    case class Let(bindings: List[Binding], x: Expr)(val pos:
 (Int, Int)) extends LetExpr
    case class Binding(v: String, x: LetExpr)(val pos: (Int, Int))
```

```
    sealed trait Expr extends LetExpr
    // We will add the position information to these nodes later
    case class Add(x: Expr, y: Expr) extends Expr
    case class Mul(x: Expr, y: Expr) extends Expr
    case class Sub(x: Expr, y: Expr) extends Expr
    case class Neg(x: Expr) extends Expr
    // But we can do these ones now
    case class Num(x: BigInt)(val pos: (Int, Int)) extends Expr
    case class Var(x: String)(val pos: (Int, Int)) extends Expr
}
```

Urgh. This isn't ideal, but realistically it's the best Scala has got. The real wart here is how this affects our parsers. Let's just take a look at a single parser and see what damage this will do:

```
val binding: Parsley[Binding] = Binding.lift(identifier, "=" ~> letExpr)
```

This no longer compiles for *several* reasons. The first is that `Binding.lift` doesn't work anymore, because `Binding` does not have the shape `(A, B) => C`. Instead it has the shape `(A, B) => C => D`, and Scala will be reluctant to make the translation. The second is that, even if we suppose that wasn't a problem, the type is going to be `Parsley[(Int, Int) => Binding]` instead of the desired `Parsley[Binding]`. If that wasn't already enough, there is the issue of having not dealt with the position anywhere either: what a mess!

We'll keep pretending that the `Binding.lift` notation works for a second, and consider how to get that position information in and get it "working" again. The combinators for extracting position information are:

```
import parsley.position._
val line: Parsley[Int]
val col: Parsley[Int]
val pos: Parsley[(Int, Int)] = line.zip(col)
```

So in this case, `pos` is what we are after, our first instinct might be to just add it as an extra parameter to the lift: `Binding.lift(identifier, "=" ~> letExpr, pos)`, but `Binding` is curried, and lift takes an uncurried function. Instead, we can use `<*>` to apply a parser returning a function to its next argument:

```
val binding: Parsley[Binding] = Binding.lift(identifier, "=" ~> letExpr)
 <*> pos
```

Again, assuming that `Binding.lift` compiles with this snippet, this would compile fine. However, it's faulty, because the position of the binding will point *after* the binding itself is finished! Instead we need to swap it round so that the position is read *before* we start reading anything to do with the binding. This is a great reason why we always read trailing whitespace and not leading whitespace, as it keeps the position as close to the token as possible. To do the position first and binding second, we can use `<**>`:

```
val binding: Parsley[Binding] = pos <**> Binding.lift(identifier, "="
 *> letExpr)
```

Now, to get it properly compiling again, we'll need to lean on the `zipped` notation instead, to help Scala's type inference figure out what we want.

```
import parsley.syntax.zipped.Zipped2
val binding: Parsley[Binding] =
    pos <**> (identifier, "=" ~> letExpr).zipped(Binding(_, _) _)
```

This *finally* compiles and works as intended. The `Binding(_, _) _` is desugared as follows:

```
Binding(_, _) _ = ((x, y) => Binding(x, y)(_))
                = ((x, y) => z => Binding(x, y)(z))
```

This isn't particularly intuitive, but it might help to recognise that the similar `Binding(_, _)(_)` is actually equivalent to `(x, y, z) => Binding(x, y)(z)`, which is not what we want. At this point though, we can see what a pain this would be if we put this into the parser in all the places, especially in a bigger parser, it's very noisy and the `.zipped` notation is (in my opinion) slightly harder to read than the `.lift` notation: it is, however, required to get Scala to correctly annotate the types of our anonymous function for us, which would otherwise make the size of the code even *worse*.

## 1 The *Parser Bridge* Pattern

The work we've done is unavoidable, but that doesn't mean we can't move it somewhere more sensible and, at the same time, get a nice new syntax to abstract the way that AST nodes are constructed. I call this technique the *Parser Bridge* pattern, and it takes many shapes depending on how the AST nodes are made.

The *Bridge* pattern is one of the classic "Gang of Four" structural design patterns. Its description is as follows:

> *Decouple an abstraction from its implementation so that the two can vary independently.*

This is roughly the intent of the *Parser Bridge* pattern, which is defined as:

> *Separate the construction of an AST node and metadata collection by using bridge constructors in the parser.*

In practice though, this can be used for more general decoupling of the AST from the parser, which we will also see examples of (especially in the Haskell interlude!). We'll start exploring this pattern, and the associated terminology, with the let binding, `Num` and `Var` cases to get a feel for it, before figuring out how to adapt it for the operators.

The general idea behind the pattern is to leverage Scala's syntactic sugar for `apply` methods. If you're unaware, `apply` methods get sugared into "function call" syntax. It is, in fact, how `case classes` don't require you to write a `new` keyword to build them: instead, the compiler has generated an `apply` method on each class' *companion object* (more on this later!). Basically, we are going to follow suit, but tailor our `apply` method to work on parsers instead of values! These `apply` methods are referred to as ***bridge constructors***. Let's get working within the `ast` object:

```
object ast {
    import parsley.position.pos
    import parsley.syntax.zipped.Zipped2

    sealed trait LetExpr
    case class Let(bindings: List[Binding], x: Expr)(val pos:
 (Int, Int)) extends LetExpr
    case class Binding(v: String, x: LetExpr)(val pos: (Int, Int))

    // New code here!
    object Let {

  def apply(bindings: Parsley[List[Binding]], x: Parsley[Expr]): Parsley[Let] =
            pos <**> (bindings, x).zipped(Let(_, _) _)
    }
    object Binding {
        def apply(v: Parsley[String], x: Parsley[LetExpr]): Parsley[Binding] =
            pos <**> (v, x).zipped(Binding(_, _) _)
    }

    sealed trait Expr extends LetExpr
    case class Add(x: Expr, y: Expr) extends Expr
    case class Mul(x: Expr, y: Expr) extends Expr
    case class Sub(x: Expr, y: Expr) extends Expr
    case class Neg(x: Expr) extends Expr
    case class Num(x: BigInt)(val pos: (Int, Int)) extends Expr
    case class Var(x: String)(val pos: (Int, Int)) extends Expr

    // New code here!
    object Num {
        def apply(x: Parsley[BigInt]): Parsley[Num] = pos <**> x.map(Num(_) _)
    }
    object Var {
        def apply(x: Parsley[String]): Parsley[Var] = pos <**> x.map(Var(_) _)
    }
}
```

Notice how the structure of the new `apply` bridge constructors mirror the shape and type of the companion class' constructor: where `Binding` requires a `String`, a `LetExpr` and a position, `Binding.apply` requires a `Parsley[String]` and a `Parsley[LetExpr]`. Notice that the position is absent

from the builder: this is the entire point! If we need to remove the position (or add a position to an existing node), we only need to make the change in the bridge constructor:

```
pos <**> (v, x).zipped(Binding(_, _) _) ===> (v, x).zipped(Binding(_, _))
```

This makes it really easy to change!

Now we can just use the bridge constructors in the main parser, and leave the work of building the data to the `apply` itself. The advantage, as I alluded to above, is that whether or not a position is required for a given node is not *at all* visible to the parser that uses its bridge: the bridge is the only place where this needs to be handled. The main parser itself now looks like this:

```
object expressions {
    import parsley.expr.{precedence, Ops, InfixL, Prefix}
    import parsley.combinator.sepEndBy1

    import lexer.implicits.implicitSymbol
    import lexer.{number, fully, identifier}
    import ast._

    private lazy val atom: Parsley[Expr] =
        "(" ~> expr <~ ")" | Num(number) | Var(identifier)
    private lazy val expr = precedence[Expr](atom)(
        Ops(Prefix)("negate" as Neg),
        Ops(InfixL)("*" as Mul),
        Ops(InfixL)("+" as Add, "-" as Sub))

    private lazy val binding = Binding(identifier, "=" ~> letExpr)
    private lazy val bindings = sepEndBy1(binding, ";")
    private lazy val letExpr: Parsley[LetExpr] =
      Let("let" ~> bindings, "in" ~> expr) | expr

    val parser = fully(letExpr)
}
```

As you can see, *very* little has changed. In fact, it's actually gotten slightly *nicer*. We no longer need to worry about `map` or `lift` inside this parser, and can focus more on the structure itself. Just to make it **very** clear: if we change our requirements for which nodes do and do not require positions, this parser will **not change** in the slightest. There is still some work left to do however: first it would be nice if the boilerplate introduced by each bridge could be reduced; and position information needs to be added to `Add`, `Mul`, `Sub`, and `Neg`.

## 2 Reducing Boilerplate with *Generic Bridge Traits*

So far, we've constructed four bridge constructors:

```
object Let {
```

```
    def apply(bindings: Parsley[List[Binding]], x: Parsley[Expr]): Parsley[Let]
 =
        pos <**> (bindings, x).zipped(Let(_, _) _)
}
object Binding {
    def apply(v: Parsley[String], x: Parsley[LetExpr]): Parsley[Binding] =
        pos <**> (v, x).zipped(Binding(_, _) _)
}
object Num {
    def apply(x: Parsley[BigInt]): Parsley[Num] = pos <**> x.map(Num(_) _)
}
object Var {
    def apply(x: Parsley[String]): Parsley[Var] = pos <**> x.map(Var(_) _)
}
```

As the number of AST nodes increase, it becomes more tedious to continue to define bridge constructors and functions by hand. This can be improved by so-called ***generic bridge traits***. This idea leverages the common structure between each of the bridge constructors and tries to build a recipe for eliminating the boilerplate. This leverages another classic OOP design pattern, called the *Template Method* pattern:

> *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclassses redefine certain steps of an algorithm (called hooks) without changing the algorithm's structure.*

Let's first desuguar these four objects a little to make the shared structure between `Let` and `Binding` as well as between `Num` and `Var` more apparent:

```
object Let {
    def apply(bindings: Parsley[List[Binding]], x: Parsley[Expr]): Parsley[Let]
 =
        pos <**> (bindings, x).zipped(Let.apply(_, _) _)
}
object Binding {
    def apply(v: Parsley[String], x: Parsley[LetExpr]): Parsley[Binding] =
        pos <**> (v, x).zipped(Binding.apply(_, _) _)
}
object Num {
    def apply(x: Parsley[BigInt]): Parsley[Num] = pos
 <**> x.map(Num.apply(_) _)
}
object Var {
    def apply(x: Parsley[String]): Parsley[Var] = pos
 <**> x.map(Var.apply(_) _)
}
```

This exposes the fact that `Let()` is just sugar for `Let.apply()`, which is automatically generated by the compiler into companion objects. Now simplify the scoping of these `apply` calls:

```
object Let {
    def apply(bindings: Parsley[List[Binding]], x: Parsley[Expr]): Parsley[Let]
 =
        pos <**> (bindings, x).zipped(this.apply(_, _) _)
}
object Binding {
    def apply(v: Parsley[String], x: Parsley[LetExpr]): Parsley[Binding] =
        pos <**> (v, x).zipped(this.apply(_, _) _)
}
object Num {
    def apply(x: Parsley[BigInt]): Parsley[Num] = pos
 <**> x.map(this.apply(_) _)
}
object Var {
    def apply(x: Parsley[String]): Parsley[Var] = pos <**> x.map(this
 .apply(_) _)
}
```

Now, the shared structure of each of these bridge constructors should hopefully be much clearer. That doesn't mean they are all identical, indeed, the types vary, as do the arities of the constructors themselves. But there is enough structure here to extract some shiny new bridge template traits:

```
trait ParserBridgePos1[-A, +B] {
    // this is called the "hook": it's the hole in the template that must be
 implemented
    def apply(x: A)(pos: (Int, Int)): B
    // this is the template method, in this case the template for the bridge
 constructor
    def apply(x: Parsley[A]): Parsley[B] = pos <**> x.map(this.apply(_) _)
}

trait ParserBridgePos2[-A, -B, +C] {
    def apply(x: A, y: B)(pos: (Int, Int)): C
    def apply(x: Parsley[A], y: Parsley[B]): Parsley[C] =
        pos <**> (x, y).zipped(this.apply(_, _) _)
}
```

These are the two generic bridge traits that provide the implementations of our bridge constructors. Obviously, there are many many more possible such traits. At the very least, it is also useful to have "plain" versions that do not interact with positions at all also (these are provided by `parsley` within `parsley.generic`):

```
trait ParserBridge1[-A, +B] {
    def apply(x: A): B
    def apply(x: Parsley[A]): Parsley[B] = x.map(this.apply(_))
}

trait ParserBridge2[-A, -B, +C] {
    def apply(x: A, y: B): C
    def apply(x: Parsley[A], y: Parsley[B]): Parsley[C] =
        (x, y).zipped(this.apply(_, _))
}
```

So, how are these used to help remove the boilerplate? Well, the companion objects for each of the AST nodes will simply extend one of the generic bridge traits as appropriate:

```
object Let extends ParserBridgePos2[List[Binding], Expr, LetExpr]
object Binding extends ParserBridgePos2[String, LetExpr, Binding]
object Num extends ParserBridgePos1[BigInt, Num]
object Var extends ParserBridgePos1[String, Var]
```

Ahhhhh, much better! If position information was removed from say `Num`, then it would just have to extend `ParserBridge1` instead, and no more changes need to be made!

## 3 *Singleton Bridge* for Precedence Ops

With the basics of bridge constructors (as well as generic bridge traits) under our belt, let's explore how we might add position information to the arithmetic operators, which are used within the `precedence` combinator. The problem with these is that the arguments to the bridge constructors are not "immediately" available when we use them: it's the `precedence` combinator that provides the arguments internally. This means we can't use the same shape of bridge here. That said, there are a couple of different ways we can implement a bridge constructor for the operators:

1) Treat them just the same as `Num`, `Var`, `Let`, and `Binding` and create a bridge constructor that looks like `Neg("negate")`, `Mul("*")`, etc. This is the easiest, and they'll differ because of the type they return (they need to be parsers that return functions). 2) Build a special combinator `from` (or `<#`) that can transform the bridges for these operators into just looking like they do now. It would look like: `Neg from "negate"`, `Mul from "*"`, etc. This is slightly more effort to do, however I think it is more faithful to how these operators usually behave. The *Parser Bridge* pattern treats arguments to the builder as arguments to the data-type, but the argument to `Neg` isn't the `()` returned by `"negate": Parsley[Unit]`, so *personally* I think it's a bit jarring to use (1).

So that you can make the choice about which style you prefer, we'll go ahead and implement both, with `Mul` using style (1) and the rest using style (2).

```
case class Mul(x: Expr, y: Expr)(val pos: (Int, Int)) extends Expr

object Mul {
    def apply(op: Parsley[Unit]): Parsley[(Expr, Expr) => Mul] =
        pos.map[(Expr, Expr) => Mul](p => Mul(_, _)(p)) <~ op
}

// or, alternatively, we can explicitly provide a new hook for our generic
 bridge trait:
object Mul extends ParserBridgePos1[Unit, (Expr, Expr) => Mul] {
    def apply(x: Unit)(pos: (Int, Int)): (Expr, Expr) => Mul = Mul(_, _)(pos)
}
```

Firstly, we can see this is a bit more effort than the previous bridge constructors. This is because there is nothing for scala's inference to "latch" onto, since we are returning a function and not providing the arguments here. While this is annoying, there isn't much we can do about it for this style. Interestingly, here we can also see an example of where the `apply` hook method can be overriden explicitly to adapt our existing bridge behaviour to a type that is **not** consistent with the AST nodes type itself: this can be useful!

Let's see how style (2) compares. To accomplish this, we can think of the new `from` combinator as being another template method provided by our generic bridge traits:

```
trait ParserBridgePos1[-A, +B] {
    def apply(x: A)(pos: (Int, Int)): B
    def apply(x: Parsley[A]): Parsley[B] = pos <**> x.map(this.apply(_) _)
    def from(op: Parsley[_]): Parsley[A => B] =
        pos.map[A => B](p => this.apply(_)(p)) <~ op
    final def <#(op: Parsley[_]): Parsley[A => B] = this from op
}

trait ParserBridgePos2[-A, -B, +C] {
    def apply(x: A, y: B)(pos: (Int, Int)): C
    def apply(x: Parsley[A], y: Parsley[B]): Parsley[C] =
        pos <**> (x, y).zipped(this.apply(_, _) _)
    def from(op: Parsley[_]): Parsley[(A, B) => C] =
        pos.map[(A, B) => C](p => this.apply(_, _)(p)) <* op
    final def <#(op: Parsley[_]): Parsley[(A, B) => C] = this from op
}
```

Now, by mixing in one of the generic bridge traits, we get two ways of using bridge constructors: the first, `apply`, allows for fully-saturated application of a constructor to its parser arguments; and the second, `from`, allows for fully-unsaturated application of a constructor to its arguments, whilst still handling the position tracking. You can imagine that partially-saturated bridge constructors can also be templated in a similar way, perhaps to fit some unconventional use-cases. In this case, here are the definitions of the companion objects for `Add`, `Sub` and `Neg` now:

```
case class Add(x: Expr, y: Expr)(val pos: (Int, Int)) extends Expr
case class Sub(x: Expr, y: Expr)(val pos: (Int, Int)) extends Expr
case class Neg(x: Expr)(val pos: (Int, Int)) extends Expr

object Add extends ParserBridgePos2[Expr, Expr, Add]
object Sub extends ParserBridgePos2[Expr, Expr, Sub]
object Neg extends ParserBridgePos1[Expr, Neg]
```

To make it clear, this automatically gives us the option to use `Add(p, q)` *or* `Add from "+"`, and its the latter that we'll want to use inside the `precedence` combinator:

```
object expressions {
    import parsley.expr.{precedence, Ops, InfixL, Prefix}
    import parsley.combinator.sepEndBy1

    import lexer.implicits.implicitSymbol
    import lexer.{number, fully, identifier}
    import ast._

    private lazy val atom: Parsley[Expr] =
        "(" ~> expr <~ ")" | Num(number) | Var(identifier)
    private lazy val expr = precedence[Expr](atom)(
        Ops(Prefix)(Neg from "negate"),
        Ops(InfixL)(Mul("*")),
        Ops(InfixL)(Add from "+", Sub from "-"))

    private lazy val binding = Binding(identifier, "=" ~> letExpr)
    private lazy val bindings = sepEndBy1(binding, ";")
    private lazy val letExpr: Parsley[LetExpr] =
      Let("let" ~> bindings, "in" ~> expr) | expr

    val parser = fully(letExpr)
}
```

## 3.1 Abstracting One More Time

In the refined definition of our generic bridge traits we supported the *Singleton Bridge* parsing design pattern by allowing the companion object itself to "appear" like a parser itself. However, if we peer in closely we can even spot some common structure between the two different `from` implementations from above:

```
trait ParserBridgePos1[-A, +B] {
    def apply(x: A)(pos: (Int, Int)): B
    def from(op: Parsley[_]): Parsley[A => B] =
        pos.map[A => B](p => this.apply(_)(p)) <* op
    final def <#(op: Parsley[_]): Parsley[A => B] = this from op
}

trait ParserBridgePos2[-A, -B, +C] {
    def apply(x: A, y: B)(pos: (Int, Int)): C
```

```
    def from(op: Parsley[_]): Parsley[(A, B) => C] =
        pos.map[(A, B) => C](p => this.apply(_, _)(p)) <* op
    final def <#(op: Parsley[_]): Parsley[(A, B) => C] = this from op
}
```

They are *almost* identical, except for the arity of the `apply` method found within the `map`. It's possible to abstract one more layer and introduce another couple of traits to help factor the common code:

```
trait ParserSingletonBridgePos[+A] {
    def con(pos: (Int, Int)): A
    def from(op: Parsley[_]): Parsley[A] = pos.map(this.con(_)) <* op
    final def <#(op: Parsley[_]): Parsley[A] = this from op
}

trait ParserBridgePos1[-A, +B] extends ParserSingletonBridgePos[A => B] {
    def apply(x: A)(pos: (Int, Int)): B
    def apply(x: Parsley[A]): Parsley[B] = pos <**> x.map(this.apply(_) _)
    override final def con(pos: (Int, Int)): A => B = this.apply(_)(pos)
}

trait ParserBridgePos2[-A, -B, +C] extends ParserSingletonBridgePos[(A, B)
 => C] {
    def apply(x: A, y: B)(pos: (Int, Int)): C
    def apply(x: Parsley[A], y: Parsley[B]): Parsley[C] =
        pos <**> (x, y).zipped(this.apply(_, _) _)
    override final def con(pos: (Int, Int)): (A, B) => C = this.apply(_, _)
(pos)
}
```

This provides a *modest* improvement over the original versions.

## 4 The Final Parser

This is our final parser which compiles fine, and tracks positions correctly. As we can see, all of the work we needed to handle the position tracking, AST node construction, whitespace handling and lexing has all been abstracted elsewhere, leaving a clean core. For completeness, here's the entire file:

```
object lexer {
    import parsley.token.{Lexer, predicate}
    import parsley.token.descriptions.{LexicalDesc, NameDesc, SymbolDesc}

    private val desc = LexicalDesc.plain.copy(
        nameDesc = NameDesc.plain.copy(
            identifierStart = predicate.Basic(_.isLetter),
            identifierLetter = predicate.Basic(_.isLetter),
        ),
        symbolDesc = SymbolDesc.plain.copy(
            hardKeywords = Set("negate", "let", "in"),
            hardOperators = Set("*", "+", "-"),
```

```
        ),
    )

    private val lexer = new Lexer(desc)

    val identifier = lexer.lexeme.names.identifier
    val number = lexer.lexeme.natural.decimal

    def fully[A](p: Parsley[A]) = lexer.fully(p)
    val implicits = lexer.lexeme.symbol.implicits
}

object ast {
    sealed trait LetExpr
    case class Let(bindings: List[Binding], x: Expr)(val pos:
 (Int, Int)) extends LetExpr
    case class Binding(v: String, x: LetExpr)(val pos: (Int, Int))

    sealed trait Expr extends LetExpr
    case class Add(x: Expr, y: Expr)(val pos: (Int, Int)) extends Expr
    case class Mul(x: Expr, y: Expr)(val pos: (Int, Int)) extends Expr
    case class Sub(x: Expr, y: Expr)(val pos: (Int, Int)) extends Expr
    case class Neg(x: Expr)(val pos: (Int, Int)) extends Expr
    case class Num(x: BigInt)(val pos: (Int, Int)) extends Expr
    case class Var(x: String)(val pos: (Int, Int)) extends Expr

    object Let extends ParserBridgePos2[List[Binding], Expr, LetExpr]
    object Binding extends ParserBridgePos2[String, LetExpr, Binding]
    object Add extends ParserBridgePos2[Expr, Expr, Add]
    object Mul extends ParserBridgePos2[Expr, Expr, Mul]
    object Sub extends ParserBridgePos2[Expr, Expr, Sub]
    object Neg extends ParserBridgePos1[Expr, Neg]
    object Num extends ParserBridgePos1[BigInt, Num]
    object Var extends ParserBridgePos1[String, Var]
}

object expressions {
    import parsley.expr.{precedence, Ops, InfixL, Prefix}
    import parsley.combinator.sepEndBy1

    import lexer.implicits.implicitSymbol
    import lexer.{number, fully, identifier}
    import ast._

    private lazy val atom: Parsley[Expr] =
        "(" ~> expr <~ ")" | Num(number) | Var(identifier)
    private lazy val expr = precedence[Expr](atom)(
        Ops(Prefix)(Neg from "negate"),
        Ops(InfixL)(Mul from "*"),
        Ops(InfixL)(Add from "+", Sub from "-"))

    private lazy val binding = Binding(identifier, "=" ~> letExpr)
    private lazy val bindings = sepEndBy1(binding, ";")
```

```
    private lazy val letExpr: Parsley[LetExpr] =
      Let("let" ~> bindings, "in" ~> expr) | expr

    val parser = fully(letExpr)
}
```

As a last thought, it's worth reinforcing that the parser bridge pattern is just a guideline: it's free to take any shape you need it to, so experiment with what works well for your own structures. The value in it really in how easy you can separate the concerns of building a structure from the parser for the grammar. Of course, there is nothing to say you *have* to use it either. If you are fine with writing the bridge constructors inline in the parser, then do it! It might be that you find the extra lines of code in the file that defines your AST to be too grating. Really this is just another exercise in how leveraging Scala's functionality when we make our parsers can help us abstract and manage our code, once again showcasing the limitless flexibility combinators provide.

# Interlude 1: Building a Parser for Haskell

We've covered a lot of ground in the series so far! I think it's time to take a medium-length break and implement a parser for a (somewhat) simplified Haskell from scratch. Haskell is a deceptively simple looking language, but is actually a minefield of ambiguous parses and the whitespace sensitivity in the language makes it even trickier. For now, I'm going to make a couple of simplifications: firstly, I'm disallowing multi-line *anything*. We are going to make a `\n` a significant character and not include it in the whitespace parsing (at least until part 3!). Furthermore, I'm banning **`where`** clauses, more than one definition in a **`let`**, guards on a new line, and **`case`** statements *must* use `{}` (these are optional in Haskell, so long as the code is well-indented). I'm also not going to deal with user defined operators, because the introduction of these during the parser does create a context-sensitive grammar, which I'd prefer to avoid for now. With that being said, here's the grammar. I wouldn't recommend spending too much time looking at the grammar, and instead we'll get started with the fairly easy task of lexing.

```
<program> ::= ((<data> | <declaration> | <clause>) NEWLINE)*

<data>         ::= 'data' <con-id> <var-id>* '=' <constructors>
<constructors> ::= <constructor> ['|' <constructors>]
<constructor>  ::= <con-id> <type-atom>*

<declaration> ::= <var-id> '::' <type>

<clause>     ::= <var-id> <pat-naked>* [<guard>] '=' <expr>
<pat-naked>  ::= <var-id> | <pat-con> | '()' | '[]' | <literal> | '_'
               | '(' <pat> ')'
               | '(' <pat> [',' <pat>]+ ')'
               | '[' <pat> [',' <pat>]* ']'
<pat>        ::= <pat-paren> [':' <pat>]
<pat-paren>  ::= <pat-app> | <pat-naked>
<pat-app>    ::= <pat-con> <pat>+
<pat-con>    ::= '(' ','+ ')' | <con-id> | '(' ':' ')'

<guard>     ::= '|' <expr>

<type>      ::= <type-app> ['->' <type>]
<type-app>  ::= <type-atom>+
<type-atom> ::= <type-con> | <var-id> | '()'  | '[' <type> ']'
              | '(' <type> (',' <type> ')')+ | '(' <type> ')'
<type-con>  ::= <con-id> | '[]' | '(' '->' ')' | '(' ','+ ')'

<expr>    ::= [<expr> '$'] <expr-1>
<expr-1>  ::= <expr-2> ['||' <expr-1>]
<expr-2>  ::= <expr-3> ['&&' <expr-2>]
<expr-3>  ::= <expr-4> [('<' | '<=' | '>' | '>=' | '==' | '/=') <expr-4>]
<expr-4>  ::= <expr-5> [(':' | '++') <expr-4>]
<expr-5>  ::= [<expr-5> ('+' | '-')] <expr-6>
<expr-6>  ::= '-' <expr-6> | <expr-7>
<expr-7>  ::= [<expr-7> ('*' | '/')] <expr-8>
```

```
<expr-8>  ::= <expr-9> ['^' <expr-8>]
<expr-9>  ::= <expr-10> ['.' <expr-9>]
<expr-10> ::= '\' <pat-naked>+ '->' <expr>
            | 'let' <clause> 'in' <expr>
            | 'if' <expr> 'then' <expr> 'else' <expr>
            | 'case' <expr> 'of' '{' <alt> [(';'|NEWLINE) <alt>]* '}'
            | <func-app>
<alt> ::= <pat> '->' <expr>
<func-app> ::= <term>+
<term> ::= <var-id> | <con-id> | '()' | '(' ','+ ')'
         | '(' <expr> ')'
         | '(' <expr> (',' <expr>)+) ')'
         | '[' [<expr> (',' <expr>)*] ']'
         | <literal>
<literal> ::= FLOAT | INTEGER | STRING | CHAR
<var-id> ::= VAR-ID
<con-id> ::= CON-ID
```

## 1 Lexing

As it turns out, lexing Haskell in Parsley is particularly easy: the `Lexer` class is compliant with the Haskell specification!

```
import parsley.Parsley

object lexer {
    import parsley.token.Lexer
    import parsley.token.descriptions.
{LexicalDesc, NameDesc, SymbolDesc, SpaceDesc,
                                       numeric, text}
    import parsley.token.predicate.{Unicode, Basic}
    import parsley.character.newline
    private val haskellDesc = LexicalDesc(
        NameDesc.plain.copy(
            identifierStart = Unicode(c => Character.isLetter(c) || c == '_'),
            identifierLetter =
                Unicode(c => Character.isLetterOrDigit(c) || c == '_' || c
 == '\''),
        ),
        SymbolDesc.plain.copy(
            hardKeywords = Set("if", "then", "else", "data", "where",
                               "let", "in", "case", "of"),
            hardOperators = Set("$", "||", "&&", "<", "<=", ">", ">=", "==", "/
=", ":",
                               "++", "+", "-", "*", "/", "^", "."),
        ),
        numeric.NumericDesc.plain.copy(
            octalExponentDesc = numeric.ExponentDesc.NoExponents,
            binaryExponentDesc = numeric.ExponentDesc.NoExponents,
        ),
        text.TextDesc.plain.copy(
```

```
            escapeSequences = text.EscapeDesc.haskell,
        ),
        SpaceDesc.plain.copy(
            commentStart = "{-",
            commentEnd = "-}",
            commentLine = "--",
            nestedComments = true,
            space = Basic(c => c == ' ' || c == '\t'),
        )
    )

    private val lexer = new Lexer(haskellDesc)

    val CON_ID = lexer.lexeme.names.identifier(Basic(_.isUpper))
    val VAR_ID = lexer.lexeme.names.identifier(Basic(_.isLower))
    val INTEGER = lexer.lexeme.natural.number
    val FLOAT = lexer.lexeme.floating.number
    val INT_OR_FLOAT = lexer.lexeme.unsignedCombined.number
    // Strictly speaking, Haskell files are probably encoded as UTF-8, but this
    // is not supported by Parsley _yet_
    val STRING = lexer.lexeme.string.fullUtf16
    val CHAR = lexer.lexeme.character.fullUtf16

    val NEWLINE = lexer.lexeme(newline).void

    def fully[A](p: Parsley[A]) = lexer.fully(p)

    val implicits = lexer.lexeme.symbol.implicits
}
```

The only tricky bit here is identifiers. Ideally, we can make a distinction between so-called constructor ids and variable ids: this is done by using the `identifier` combinator, which refines what the first letter of the identifier is allowed to be. I've exposed `INT_OR_FLOAT` to our interface here, since it prevents any backtracking required by `INTEGER | FLOAT`. Also seen here, is the `NEWLINE` token, which we will use in the parser to deliberate delimit newlines, but still ensure it consumes whitespace! The reason I have picked the shouty-case names is to mimic how tokens sometimes look in grammars. This is purely stylistic, but will help us distinguish between parts of our parser and the primitives of our lexer.

## 2 The AST + *Parser Bridge* Pattern

Now it's time to build the AST for our Haskell Parser to return. Since we'll be using the *Parser Bridge* pattern anyway, I get a choice about whether or not I want position tracking for each node in the tree. Just to keep the AST looking simple, I'll not track anything. Of course, if I did change my mind, I could do it here by changing which generic bridge trait is used. More interesting will be what bridge constructor shapes I pick for each of the AST nodes. Let's start by just outlining the datatypes themselves and why they are how they are:

```
object ast {
```

```
    import parsley.generic._

    case class HaskellProgram(lines: List[ProgramUnit])
    sealed trait ProgramUnit

  case class Data(id: ConId, tys: List[VarId], cons: List[Con]) extends ProgramUnit
    case class Con(id: ConId, tys: List[TyAtom])

    case class Decl(id: VarId, ty: Type) extends ProgramUnit


  case class Clause(id: VarId, pats: List[PatNaked], guard: Option[Expr], rhs: Expr)
        extends ProgramUnit
    sealed trait Pat
    case class PatCons(x: PatParen, xs: Pat) extends Pat
    sealed trait PatParen extends Pat
    case class PatApp(con: PatCon, args: List[PatNaked]) extends PatParen
    sealed trait PatNaked extends PatParen
    case object NilCon extends PatNaked with ParserBridge0[PatNaked]
    case object Wild extends PatNaked with ParserBridge0[PatNaked]
    case class NestedPat(pat: Pat) extends PatNaked
    case class PatTuple(xs: List[Pat]) extends PatNaked
    case class PatList(xs: List[Pat]) extends PatNaked
    sealed trait PatCon extends PatNaked
    case object ConsCon extends PatCon with ParserBridge0[PatCon]

    sealed trait Type
    case class FunTy(argTy: Type_, resTy: Type) extends Type
    sealed trait Type_ extends Type
    case class TyApp(tyF: Type_, tyX: TyAtom) extends Type_
    sealed trait TyAtom extends Type_
    case object UnitTy extends TyAtom with ParserBridge0[TyAtom]
    case class ListTy(ty: Type) extends TyAtom
    case class TupleTy(tys: List[Type]) extends TyAtom
    // This is needed if we want to maximise the well-typedness of the parser
    // For a parser as big as this one, it's definitely desirable: we can
always
    // weaken the types later if we want to!
    case class ParenTy(ty: Type) extends TyAtom
    case object ListConTy extends TyAtom with ParserBridge0[TyAtom]
    case object FunConTy extends TyAtom with ParserBridge0[TyAtom]
    case class TupleConTy(arity: Int) extends TyAtom

    // We'll model this layer by layer, to maximise the flexiblity whilst
maintaining
    // The type safety: by using subtyping, we can avoid useless wrapper
constructors
    sealed trait Expr
    case class WeakApp(f: Expr, arg: Expr1) extends Expr
    sealed trait Expr1 extends Expr
    case class Or(x: Expr2, y: Expr1) extends Expr1
    sealed trait Expr2 extends Expr1
    case class And(x: Expr3, y: Expr2) extends Expr2
```

```
    sealed trait Expr3 extends Expr2
    // We could certainly compress this by factoring out the op!
    // Notice that these operators have Expr4 on both sides: this implies they
are
    // not left _or_ right associative! x < y < z is not legal in Haskell
    case class Less(x: Expr4, y: Expr4) extends Expr3
    case class LessEqual(x: Expr4, y: Expr4) extends Expr3
    case class Greater(x: Expr4, y: Expr4) extends Expr3
    case class GreaterEqual(x: Expr4, y: Expr4) extends Expr3
    case class Equal(x: Expr4, y: Expr4) extends Expr3
    case class NotEqual(x: Expr4, y: Expr4) extends Expr3
    sealed trait Expr4 extends Expr3
    case class Cons(x: Expr5, xs: Expr4) extends Expr4
    case class Append(xs: Expr5, ys: Expr4) extends Expr4
    sealed trait Expr5 extends Expr4
    case class Add(x: Expr5, y: Expr6) extends Expr5
    case class Sub(x: Expr5, y: Expr6) extends Expr5
    sealed trait Expr6 extends Expr5
    case class Negate(x: Expr6) extends Expr6
    sealed trait Expr7 extends Expr6
    case class Mul(x: Expr7, y: Expr8) extends Expr7
    case class Div(x: Expr7, y: Expr8) extends Expr7
    sealed trait Expr8 extends Expr7
    case class Exp(x: Expr9, y: Expr8) extends Expr8
    sealed trait Expr9 extends Expr8
    case class Comp(f: Expr10, g: Expr9) extends Expr9
    sealed trait Expr10 extends Expr9
    case class Lam(args: List[Pat], body: Expr) extends Expr10
    case class Let(binding: Clause, in: Expr) extends Expr10
    case class If(cond: Expr, thenExpr: Expr, elseExpr: Expr) extends Expr10
    case class Case(scrutinee: Expr, cases: List[Alt]) extends Expr10
    case class Alt(pat: Pat, body: Expr)
    sealed trait Expr10_ extends Expr10
    case class StrongApp(f: Expr10_, arg: Term) extends Expr10_

    sealed trait Term extends Expr10_
    case class ConId(v: String) extends Term with PatCon with TyAtom
    case class VarId(v: String) extends Term with PatNaked with TyAtom

  case object UnitCon extends Term with PatNaked with ParserBridge0[Term with PatNaked]
    case class TupleCon(arity: Int) extends Term with PatCon
    case class ParensVal(x: Expr) extends Term
    case class TupleLit(xs: List[Expr]) extends Term
    case class ListLit(xs: List[Expr]) extends Term

    trait Literal extends Term with PatNaked
    case class HsInt(x: BigInt) extends Literal
    case class HsString(s: String) extends Literal
    case class HsChar(c: Int) extends Literal
    case class HsDouble(x: BigDecimal) extends Literal

    object WeakApp extends ParserBridge2[Expr, Expr1, Expr]
    object Or extends ParserBridge2[Expr2, Expr1, Expr1]
```

```
    object And extends ParserBridge2[Expr3, Expr2, Expr2]
    object Less extends ParserBridge2[Expr4, Expr4, Expr3]
    object LessEqual extends ParserBridge2[Expr4, Expr4, Expr3]
    object Greater extends ParserBridge2[Expr4, Expr4, Expr3]
    object GreaterEqual extends ParserBridge2[Expr4, Expr4, Expr3]
    object Equal extends ParserBridge2[Expr4, Expr4, Expr3]
    object NotEqual extends ParserBridge2[Expr4, Expr4, Expr3]
    object Cons extends ParserBridge2[Expr5, Expr4, Expr4]
    object Append extends ParserBridge2[Expr5, Expr4, Expr4]
    object Add extends ParserBridge2[Expr5, Expr6, Expr5]
    object Sub extends ParserBridge2[Expr5, Expr6, Expr5]
    object Negate extends ParserBridge1[Expr6, Expr6]
    object Mul extends ParserBridge2[Expr7, Expr8, Expr7]
    object Div extends ParserBridge2[Expr7, Expr8, Expr7]
    object Exp extends ParserBridge2[Expr9, Expr8, Expr8]
    object Comp extends ParserBridge2[Expr10, Expr9, Expr9]
    object FunTy extends ParserBridge2[Type_, Type, Type]
    object Lam extends ParserBridge2[List[Pat], Expr, Lam]
    object Let extends ParserBridge2[Clause, Expr, Let]
    object If extends ParserBridge3[Expr, Expr, Expr, If]
    object Case extends ParserBridge2[Expr, List[Alt], Case]
    object Alt extends ParserBridge2[Pat, Expr, Alt]
    object ConId extends ParserBridge1[String, ConId]
    object VarId extends ParserBridge1[String, VarId]
    object TupleCon extends ParserBridge1[Int, TupleCon]
    object ParensVal extends ParserBridge1[Expr, ParensVal]
    object TupleLit extends ParserBridge1[List[Expr], TupleLit]
    object ListLit extends ParserBridge1[List[Expr], ListLit]
    object HsInt extends ParserBridge1[BigInt, HsInt]
    object HsString extends ParserBridge1[String, HsString]
    object HsChar extends ParserBridge1[Int, HsChar]
    object HsDouble extends ParserBridge1[BigDecimal, HsDouble]
    object Data extends ParserBridge3[ConId, List[VarId], List[Con], Data]
    object Con extends ParserBridge2[ConId, List[TyAtom], Con]
    object Decl extends ParserBridge2[VarId, Type, Decl]

  object Clause extends ParserBridge4[VarId, List[PatNaked], Option[Expr], Expr, Clause]
    object PatCons extends ParserBridge2[PatParen, Pat, Pat]
    object PatApp extends ParserBridge2[PatCon, List[PatNaked], PatApp]
    object NestedPat extends ParserBridge1[Pat, NestedPat]
    object PatTuple extends ParserBridge1[List[Pat], PatTuple]
    object PatList extends ParserBridge1[List[Pat], PatList]
    object TupleConTy extends ParserBridge1[Int, TupleConTy]
    object ParenTy extends ParserBridge1[Type, ParenTy]
    object TupleTy extends ParserBridge1[List[Type], TupleTy]
    object ListTy extends ParserBridge1[Type, ListTy]
}
```

There is a lot of constructors here, but that's because the grammar is quite big. Notice that the shape of the AST roughly follows the shape of the grammar (down to the naming). Subtyping has been used where the same rule can appear in multiple places (`ConId` or `VarId` for instance). This helps keep the parser simple

whilst still providing a level of type safety. We know, for instance, the associativity of the operators purely based on their types alone. By generalising, we can see that left-associative types are shaped like `(B, A) => B`, right associative ones as `(A, B) => B` and non-associative ones as `(A, A) => B`. This is a helpful guide for us, and in fact it will also ensure that we can't get the precedence table "wrong". A consequence, as we'll see later, is we will be forced to use `SOps` instead of `Ops` in the precedence tables.

The bridges have all also been defined above as well, including those marked with `ParserBridge0`, which is done on the object itself.

Notably though, there are two AST nodes I'm *not* going to give bridge constructors to: `StrongApp`, `TyApp`. If you look at the grammar, you'll see that the two relevant rules are both just `many`-like. Another option for these datatypes would have been to have taken a `List` of the sub-parses. But, morally, Haskell function applications are done one at a time so I've not flattened the structure, and as we'll see, we'll use a `reduce` to handle these cases. In reality, providing a position to either of these two nodes is quite difficult, because they are actually both delimited by `' '`, so there really isn't a sensible position to latch on to.

# 3 Parsing

Now it's finally time to tackle the parser itself. Remember, our lexer handles all whitespace (except newlines) for us, and our bridge constructors would handle position information for us. The type of the AST is going to help us make sure that the parsers are constructed in the right way. One concern we need to be aware of is notice areas of the grammar where ambiguity lies, and make sure we resolve it properly with `atomic` (and only when needed!). Let's start at the bottom and work our way up: this means tackling, in order, atoms, expressions, patterns, clauses, types, declarations, and then finally data.

## 3.1 Atoms of the Expression

Let's make a start with `<term>`. There is nothing particular special about this, but we will need to be careful to handle the difference between tupled expressions and parenthesised expressions. There are a couple of ways we can try and tackle this: the first is to write both cases out and be sure to `atomic` one of them, so we can back out if required. The second is to parse them as one case together and then disambiguate which of the constructors should be used in a bridge! I'm going to take the first approach for now, and then we can revisit later. Now for the `<term>` parser:

```
import parsley.Parsley.atomic
import parsley.combinator.{sepBy, sepBy1, countSome}

import lexer._
import implicits.implicitSymbol
import ast._

val `<literal>` = atomic(HsDouble(FLOAT)) | HsInt(INTEGER) | HsString(STRING)
 | HsChar(CHAR)
val `<var-id>` = VarId(VAR_ID)
```

```
val `<con-id>` = ConId(CON_ID)

val `<expr>`: Parsley[Expr] = /???/

val `<term>` = ( `<var-id>` | `<con-id>` | (UnitCon from "()")
               | atomic(TupleCon("(" ~> countSome(",") <~ ")"))
               | atomic(ParensVal("(" ~> `<expr>` <~ ")"))
               | TupleLit("(" ~> sepBy1(`<expr>`, ",") <~ ")")
               | ListLit("[" ~> sepBy(`<expr>`, ",") <~ "]")
               | `<literal>`
               )
```

Here, I've followed the structure of the grammar quite closely. I'm even making sure to follow the same order that the grammar uses: this means that I use `lazy val` for any parser which forward references another grammar rule (this is why I didn't need the laziness in the bridges). I'm making use of the `sepBy` and `sepBy1` combinators to handle the comma separated values and parse them into a list. Notice that there are three instances of backtracking *alone* in this parser. Tuple constructions (like `(,,)`), parenthesised values, and tuple literals all share the `(` token. That being said, so does `()`, but there it's been treated as a single atomic token by our `implicitSymbol`, so no backtracking required at the branching level. As I said, this could get expensive, so we will re-visit it later. The same crops up with `FLOAT` and `INTEGER`, which may also overlap with each other: again, we will re-visit this later and use the `FLOAT_OR_INT` token instead. To deal with the number of `,` representing the arity of a tuple constructor operator, I've also drafted up a `countSome` combinator, which will parse its given parser one or more times, counting the number of successes. You'll also notice that, in Scala, anything goes between backticks! For this parser, I'll go with this notation to make it match a little more closely with the grammar itself (and for variety): it's up to you whether or not you like this notation. One thing that's nice about it is that it clearly distinguishes between our combinators and the grammar rules themselves.

## 3.2 Expressions

Next we'll tackle up to `<expr>`. By now, we should know that the correct tool to reach for is `precedence`. While `<expr-10>` can be considered to have operators in it, they do not fit with any associativity in Haskell so I will split them out for digestibility. Without further ado, let's get going:

```
import parsley.expr.{precedence, SOps, InfixL, InfixR, InfixN, Prefix, Atoms}

lazy val `<expr>`: Parsley[Expr] = precedence {
    SOps(InfixL)(WeakApp from "$") +:
    SOps(InfixR)(Or      from "||") +:
    SOps(InfixR)(And     from "&&") +:
    SOps(InfixN)(Less    from "<",  LessEqual    from "<=",
                 Greater from ">",  GreaterEqual from ">=",
                 Equal   from "==", NotEqual     from "/=") +:
    SOps(InfixR)(Cons    from ":",  Append       from "++") +:
```

```
    SOps(InfixL)(Add     from "+",  Sub          from "-") +:
    SOps(Prefix)(Negate  from "-") +:
    SOps(InfixL)(Mul     from "*",  Div          from "/") +:
    SOps(InfixR)(Exp     from "^") +:
    SOps(InfixR)(Comp    from ".") +:
    `<expr-10>`
}
lazy val `<expr-10>` = Atoms(/???/)
```

Here we can see a whole bunch of interesting things! Firstly, up to this point we've been used to seeing `Ops` in our precedence, where here we are using `SOps` and the `Levels` list. This is important, because our AST is far more strongly typed. If we made each layer of the tree the same `(Expr, Expr) => Expr` shape, then we could use `Ops` as we've been used to in other pages. However, sine I opted to make a more strongly typed tree using subtyping, we have to use the more complex and general `SOps` precedence architecture. This has some really nice consequences:

1) if, say, I removed `SOps(InfixR)(Exp <# "^")` from the list, it would no longer compile 2) if, say, I accidentally said `SOps(InfixL)(Exp <# "^")`, then it would no longer compile 3) if, say, I reordered the lines in the list, then it would no longer compile 4) by using subtyping, we don't need to provide any explicit wrapper constructors

Next up is the remaining three rules: `<expr-10>`, `<alt>`, and `<func-app>`.

```
import parsley.Parsley.{some, many}

val `<clause>`: Parsley[Clause] = /???/
val `<pat-naked>`: Parsley[PatNaked] = /???/
val `<pat>`: Parsley[Pat] = /???/

val `<alt>` = Alt(`<pat>`, "->" ~> `<expr>`)
lazy val `<func-app>` = `<term>`.reduceLeft(StrongApp)
lazy val `<expr-10>` = Atoms(
    Lam("\\" ~> some(`<pat-naked>`), "->" ~> `<expr>`),
    Let("let" ~> `<clause>`, "in" ~> `<expr>`),
    If("if" ~> `<expr>`, "then" ~> `<expr>`, "else" ~> `<expr>`),
    Case("case" ~> `<expr>`,
         "of" ~> "{" ~> sepBy1(`<alt>`, (";" | NEWLINE) <~ many(NEWLINE))
 <~ "}"),
    `<func-app>`)
```

This section of the parser is much more straightforward: we are using the regular shape of the grammar in conjunction with our bridge constructors. Notice here we are explicitly making use of `NEWLINE`, so that we can make multi-line `case` statements. This is ok because we have explicit curly braces to delimit the start and end of the `case`. Note here that the `"\\"` is just parsing a `\`, but the backslash much be escaped to fit within the Scala string! The `<func-app>` rule is interesting, because it is the

same as `some(`<term>`).map(_.reduceLeft(StrongApp))`, but is more efficient, not having to have to construct an intermediate list. It's always a good idea to check out `parsley`'s API to see if you can find any hidden gems like this one!

Now we have tackled everything from `<expr>` down, we are now in a position to deal with `<clause>` and its sub grammars.

## 3.3 Clauses

These set of parsers are similar to what we saw with `<term>`. There are going to be elements of ambiguity in the grammar that need to be resolved with `atomic`, specifically those involving parentheses. But other than that there isn't anything really new here.

```
import parsley.Parsley.many
import parsley.combinator.option
import parsley.expr.infix

lazy val `<clause>` =
    Clause(`<var-id>`, many(`<pat-naked>`), option(`<guard>`), "=" ~> `<expr>`)
lazy val `<pat-naked>`: Parsley[PatNaked] =
    ( `<var-id>` | atomic(`<pat-con>`)
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | atomic(NestedPat("(" ~> `<pat>` <~ ")"))
    | PatTuple("(" ~> sepBy1(`<pat>`, ",") <~ ")")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )
lazy val `<pat>` = infix.right1(`<pat-paren>`, PatCons from ":")
lazy val `<pat-paren>` = atomic(`<pat-app>`) | `<pat-naked>`
lazy val `<pat-app>` = PatApp(`<pat-con>`, some(`<pat-naked>`))
lazy val `<pat-con>` = ( atomic("(" ~> (ConsCon from ":") <~ ")")
                       | TupleCon("(" ~> countSome(",") <~ ")")
                       | `<con-id>`
                       )

lazy val `<guard>` = "|" ~> `<expr>`
```

Like I said, nothing too interesting here. Notice, however, that for `<pat>` I have used a `infix.right1`: we've been used to using `precedence`, but in cases like these the `chain`s are just so much more simple. Don't be afraid to make use of them! There are plenty of `atomic`s here that we will come back and eliminate later on: notably, the `atomic` in `<pat-paren>` is used to guard against a `<pat-con>` being parsed without any `<pat-naked>` - this needs to be parsed again in `<pat-naked>` itself.

## 3.4 Types

Again, here we can just see more examples of the same concepts we've already been working with. There is a sense in which we've really reached the limit of stuff we need for our practical common cases: there isn't much more to say until we try and deal with much more complex grammatical features.

```
lazy val `<type>`: Parsley[Type] = infix.right1(`<type-app>`, FunTy from "->")
lazy val `<type-app>` = `<type-atom>`.reduceLeft(TyApp)
lazy val `<type-atom>` = ( `<type-con>` | `<var-id>` | (UnitTy from "()")
                         | ListTy("[" ~> `<type>` <~ "]")
                         | atomic(ParenTy("(" ~> `<type>` <~ ")"))
                         | TupleTy("(" ~> sepBy1(`<type>`, ",") <~ ")")
                         )
lazy val `<type-con>` = ( `<con-id>`
                        | (ListConTy from "[]")
                        | atomic("(" ~> (FunConTy from "->") <~ ")")
                        | atomic(TupleConTy("(" ~> countSome(",") <~ ")"))
                        )
```

We can see another instance of `infix.right1` out in the wild here, as well as the other `reduceLeft` used for type application. Yet again, there is some ambiguity evidenced by the `atomic` here, and just like in the other instances, it's to do with parentheses. The second `atomic` in `<type-con>` is interesting: it's not clear from the rule itself why it is there. In fact, it's there because we need to be able to backtrack out of `<type-con>` during `<type-atom>`; however, since the ambiguity only arises from the tuple constructor (the function constructor doesn't cause ambiguity in this case, because of the existing `atomic`), we don't need to enable backtracking on the *entire* rule. Finally we can move onto the top level parsers and start tackling the performance gotchas: this is the more challenging aspect of managing a parser of this size and complexity.

## 3.5 Declarations and Data

Here is a nice easy finish. These last rules are really just book-keeping. I'm also going to introduce a way of running the parser directly.

```
import parsley.Parsley.some
import parsley.combinator.sepEndBy
import parsley.errors.ErrorBuilder

def parse[Err: ErrorBuilder](input: String) = `<program>`.parse(input)

lazy val `<program>` =
    fully(sepEndBy(`<data>` | atomic(`<declaration>`)
 | `<clause>`, some(NEWLINE)))

lazy val `<data>` = Data("data" ~> `<con-id>`, many(`<var-id>`), "="
 ~> `<constructors>`)
lazy val `<constructors>` = sepBy1(`<constructor>`, "|")
lazy val `<constructor>` = Con(`<con-id>`, many(`<type-atom>`))
```

```
lazy val `<declaration>` = Decl(`<var-id>`, "::" ~> `<type>`)
```

We do have to be careful here, there is some overlap between `<declaration>` and `<clause>`. This, unfortunately, is unavoidable without more involved work, but the overlap is just a single variable name, so it's fairly minor. That being said, really, the scope on that `atomic` is too big: once we've seen the `"::"`, we *know* for sure that everything that follows (good or bad) is a type. At the moment, if something bad happens in the type, it can backtrack all the way out and try reading a clause instead. This isn't too much of a problem for us here, since the clause will also fail and the declaration's error message got further (and so takes precedence), but suppose the backtracking allowed the parser to succeed somehow (perhaps it was `many(`<declaration>`)`) then the error will for sure be in the wrong place!

The `parse` method is our hook to the outside world. I've written it "properly" here so that it is parameterised by an instance of the `ErrorBuilder` typeclass. We could omit this, but then we'd always have to run it with the `ErrorBuilder` in scope for whatever concrete `Err` we choose (in most cases, `String`). It's useful to do this in case your unit tests want to test for the correctness of error messages!

# 4 Optimising the Parser

## 4.1 Fixing Parentheses Backtracking

Before we finish up this page, and move onto learning about error messages, let's deal with the last remaining warts in this version of our parser. There are several instances of backtracking: some are very avoidable, and others are potentially expensive. We will progress through them, in some cases making incremental changes. As I've said before, dealing with backtracking in an effective way is in most cases the trickiest part of actually writing a parser. Make sure you understand how the parser has been built up so far and convince yourself about why each instance of `atomic` in the previous parsers has been necessary before continuing.

### 4.1.1 `<pat-con>`, `<type-con>`, and `<term>`

These three grammar rules are very straightforward to factor out and remove the backtracking for, so we will start with them first. Let's remind ourselves of the three rules in question and identify the parts we can handle:

```
lazy val `<pat-con>` = ( atomic("(" ~> (ConsCon from ":") <~ ")")
                       | TupleCon("(" ~> countSome(",") <~ ")")
                       | `<con-id>`
                       )

val `<type-con>` = ( `<con-id>`
                   | (ListConTy from "[]")
                   | atomic("(" ~> (FunConTy from "->") <~ ")")
                   | atomic(TupleConTy("(" ~> countSome(",") <~ ")"))
                   )
```

```
val `<term>` = ( `<var-id>` | `<con-id>` | (UnitCon from "()")
               | atomic(TupleCon("(" ~> countSome(",") <~ ")"))
               | atomic(ParensVal("(" ~> `<expr>` <~ ")"))
               | TupleLit("(" ~> sepBy1(`<expr>`, ",") <~ ")")
               | ListLit("[" ~> sepBy(`<expr>`, ",") <~ "]")
               | `<literal>`
               )
```

Now, with `<pat-con>` and `<type-con>`, they both contain backtracking because there are two portions of the parser which lie within parentheses. You can see the same thing in the `<term>` parser, however, as we'll see, `<term>` will require a bit more extra work to fix the second instance of backtracking. Thankfully, these are all relatively easy to fix: we just need to distribute the parentheses through the rules that contain them on the *inside*. This is a nice warm-up exercise:

```
lazy val `<pat-con>` = ( atomic("(" ~> (ConsCon from ":") <~ ")")
                       | "(" ~> TupleCon(countSome(",")) <~ ")"
                       | `<con-id>`
                       )

val `<type-con>` = ( `<con-id>`
                   | (ListConTy from "[]")
                   | atomic("(" ~> (FunConTy from "->") <~ ")")
                   | atomic("(" ~> TupleConTy(countSome(",")) <~ ")")
                   )

val `<term>` = ( `<var-id>` | `<con-id>` | (UnitCon from "()")
               | atomic("(" ~> TupleCon(countSome(",")) <~ ")")
               | atomic("(" ~> ParensVal(`<expr>`) <~ ")")
               | "(" ~> TupleLit(sepBy1(`<expr>`, ",")) <~ ")"
               | ListLit("[" ~> sepBy(`<expr>`, ",") <~ "]")
               | `<literal>`
               )
```

With the parentheses distributed, we can see that they are easily factored out (on both the left- and the right-hand sides):

```
lazy val `<pat-con>` = ( "(" ~> ((ConsCon from ":") | TupleCon(countSome(",")))
 <~ ")"
                       | `<con-id>`
                       )

val `<type-con>` = ( `<con-id>`
                   | (ListConTy from "[]")
                   | atomic("(" ~> ((FunConTy from "->")
 | TupleConTy(countSome(","))) <~ ")")
                   )

val `<term>` = ( `<var-id>` | `<con-id>` | (UnitCon from "()")
```

```
                | "(" ~> ( TupleCon(countSome(","))
                        | atomic(ParensVal(`<expr>`))
                        | TupleLit(sepBy1(`<expr>`, ","))
                        ) <~ ")"
                | ListLit("[" ~> sepBy(`<expr>`, ",") <~ "]")
                | `<literal>`
                )
```

This has immediately eliminated three of the `atomics`, but one persists inside `<term>`: this is because `ParensVal` and `TupleLit` both share an `<expr>`. This is a bit trickier to eliminate, but let's move on to tackling these. Note that the other `atomic` in `<type-con>` is due to a conflict in the wider grammar, we can look at that in part 4.

### 4.1.2 `<pat-naked>`, `<type-atom>`, and `<term>` (again)

The next three grammar rules contain similar patterns to the last three, but solving the backtracking is less obvious. Let's start by recapping what the three rules are:

```
lazy val `<pat-naked>`: Parsley[PatNaked] =
    ( `<var-id>` | atomic(`<pat-con>`)
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | atomic(NestedPat("(" ~> `<pat>` <~ ")"))
    | PatTuple("(" ~> sepBy1(`<pat>`, ",") <~ ")")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )

lazy val `<type-atom>` = ( `<type-con>` | `<var-id>` | (UnitTy from "()")
                         | ListTy("[" ~> `<type>` <~ "]")
                         | atomic(ParenTy("(" ~> `<type>` <~ ")"))
                         | TupleTy("(" ~> sepBy1(`<type>`, ",") <~ ")")
                         )

val `<term>` = ( `<var-id>` | `<con-id>` | (UnitCon from "()")
               | "(" ~> ( TupleCon(countSome(","))
                        | atomic(ParensVal(`<expr>`))
                        | TupleLit(sepBy1(`<expr>`, ","))
                        ) <~ ")"
               | ListLit("[" ~> sepBy(`<expr>`, ",") <~ "]")
               | `<literal>`
               )
```

First, let's do what we did to `<term>` to `<pat-naked>` and `<type-atom>`:

```
lazy val `<pat-naked>`: Parsley[PatNaked] =
    ( `<var-id>` | atomic(`<pat-con>`)
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | "(" ~> ( atomic(NestedPat(`<pat>`))
             | PatTuple(sepBy1(`<pat>`, ","))
```

```
            ) <~ ")"
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )

lazy val `<type-atom>` = ( `<type-con>` | `<var-id>` | (UnitTy from "()")
                         | ListTy("[" ~> `<type>` <~ "]")
                         | "(" ~> ( atomic(ParenTy(`<type>`))
                                  | TupleTy(sepBy1(`<type>`, ","))
                                  ) <~ ")"
                         )

val `<term>` = ( `<var-id>` | `<con-id>` | (UnitCon from "()")
               | "(" ~> ( TupleCon(countSome(","))
                        | atomic(ParensVal(`<expr>`))
                        | TupleLit(sepBy1(`<expr>`, ","))
                        ) <~ ")"
               | ListLit("[" ~> sepBy(`<expr>`, ",") <~ "]")
               | `<literal>`
               )
```

Hopefully you can see that all three rules are similar to each other: they all have an `atomic` *inside* the factored parentheses. The problem is that they use different bridge constructors and the `sepBy1` does not allow for easy factoring. That being said, we could deal with this by creating... a new ***disambiguator bridge***! Let's take a look at them:

```
object NestedPatOrPatTuple extends ParserBridge1[List[Pat], PatNaked] {
    def apply(ps: List[Pat]): PatNaked = ps match {
        case List(p) => NestedPat(p)
        case ps => PatTuple(ps)
    }
}

object ParenTyOrTupleTy extends ParserBridge1[List[Type], TyAtom] {
    def apply(tys: List[Type]): TyAtom = tys match {
        case List(ty) => ParenTy(ty)
        case tys => TupleTy(tys)
    }
}

object TupleLitOrParensVal extends ParserBridge1[List[Expr], Term] {
    def apply(xs: List[Expr]): Term = xs match {
        case List(x) => ParensVal(x)
        case xs => TupleLit(xs)
    }
}
```

Compared with ***bridge constructors***, ***disambiguator bridges*** are used to arbitrate between several possible instantiations dependending on the data that is fed to them. These bridges each encapsulate both of the

overlapping cases. If the list of results only has one element, then it indicates we should not be creating a tuple. With these new bridges replacing the old ones, we can adjust our parsers:

```
lazy val `<pat-naked>`: Parsley[PatNaked] =
    ( `<var-id>` | atomic(`<pat-con>`)
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | NestedPatOrPatTuple("(" ~> sepBy1(`<pat>`, ",") <~ ")")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )

lazy val `<type-atom>` = ( `<type-con>` | `<var-id>` | (UnitTy from "()")
                         | ListTy("[" ~> `<type>` <~ "]")
                         | ParenTyOrTupleTy("(" ~> sepBy1(`<type>`, ",")
 <~ ")")
                         )

val `<term>` = ( `<var-id>` | `<con-id>` | (UnitCon from "()")
               | "(" ~> ( TupleCon(countSome(","))
                        | TupleLitOrParensVal(sepBy1(`<expr>`, ","))
                        ) <~ ")"
               | ListLit("[" ~> sepBy(`<expr>`, ",") <~ "]")
               | `<literal>`
               )
```

By making use of our special disambiguators bridge, we've eliminated the pesky `atomic`s. I've put the parentheses back inside the bridge call for the `<pat-naked>` and `<type-atom>` rules, because, in my opinion, it looks a bit cleaner.

### 4.1.3 Numbers in `<literal>`

With the tools we've already developed so far, this one is easy. Let's remind ourselves of the rule first:

```
/*
val INTEGER = lexer.lexeme.numeric.natural.number
val FLOAT = lexer.lexeme.numeric.floating.number
val INT_OR_FLOAT = lexer.lexeme.numeric.unsignedCombined.number
*/
val `<literal>` = atomic(HsDouble(FLOAT)) | HsInt(INTEGER) | HsString(STRING)
 | HsChar(CHAR)
```

The problem here is that floats and ints share a common leading prefix: the whole number part. When we defined the `lexer`, I mentioned that it supports a `INT_OR_FLOAT` token. Now it's time to make use of it to remove this `atomic`. Our first thought might be to make a disambiguator bridge that can accommodate either of them, and that would be a fine idea:

```
object HsIntOrDouble extends ParserBridge1[Either[BigInt, BigDecimal], Literal]
 {
    def apply(x: Either[BigInt, BigDecimal]): Literal
 = x.fold(HsInt(_), HsDouble(_))
}

val `<literal>` = HsIntOrDouble(INT_OR_FLOAT) | HsString(STRING) | HsChar(CHAR)
```

And this works fine!

### 4.1.4 Loose ends

At this point, there are four **atomic**s left in the entire parser: one in **<type-con>**, one in **<pat-paren>**, one in **<pat-naked>**, and, finally, one in **<program>**. At this point it gets much harder to remove them without altering the grammar substantially. Let's see how much more we can do to remove them and explore some of the more dramatic changes it entails to the parser. We'll start with the **atomic** in **<type-con>**.

The reason for this **atomic** is more obvious when we compare it with **<type-atom>**:

```
val `<type-con>` = ( `<con-id>`
                   | (ListConTy from "[]")
                   | atomic("(" ~> ((FunConTy from "->")
 | TupleConTy(countSome(","))) <~ ")")
                   )
lazy val `<type-atom>` = ( `<type-con>` | `<var-id>` | (UnitTy from "()")
                         | ListTy("[" ~> `<type>` <~ "]")
                         | ParenTyOrTupleTy("(" ~> sepBy1(`<type>`, ",")
 <~ ")")
                         )
```

The **atomic** in **<type-con>** is used to backtrack out of the parentheses, since **ParenTyOrTupleTy** will also consume them if we didn't see a **->** or a **,**. Thankfully, **<type-con>** is used in one place in the parser, and we are looking at it! To fix this **atomic** we just need to be destructive and stop following the grammar as rigidly as we have been. Let's inline **<type-con>** into **<type-atom>** to start with:

```
lazy val `<type-atom>` = ( `<con-id>`
                         | (ListConTy from "[]")
                         | atomic("(" ~> ( (FunConTy from "->")
                                         | TupleConTy(countSome(","))
                                         ) <~ ")")
                         | `<var-id>` | (UnitTy from "()")
                         | ListTy("[" ~> `<type>` <~ "]")
                         | ParenTyOrTupleTy("(" ~> sepBy1(`<type>`, ",")
 <~ ")")
                         )
```

Right, now that they have been put together, we can see the problem more clearly. Let's now reorganise the parser so that the problematic parentheses appear next to each other: it's worth mentioning that, for parsers without backtracking, we can always reorder the branches without consequence; the restriction is that backtracking parsers cannot move ahead of their paired up branch.

```
lazy val `<type-atom>` = ( `<con-id>`
                         | (ListConTy from "[]")
                         | `<var-id>` | (UnitTy from "()")
                         | ListTy("[" ~> `<type>` <~ "]")
                         | atomic("(" ~> ( (FunConTy from "->")
                                         | TupleConTy(countSome(","))
                                         ) <~ ")")
                         | ParenTyOrTupleTy("(" ~> sepBy1(`<type>`, ",")
 <~ ")")
                         )
```

This parser is a bit neater, and now we can apply our favourite tricks from part 1 to resolve this `atomic`:

```
lazy val `<type-atom>` = ( `<con-id>`
                         | (ListConTy from "[]")
                         | `<var-id>` | (UnitTy from "()")
                         | ListTy("[" ~> `<type>` <~ "]")
                         | "(" ~> ( (FunConTy from "->")
                                  | TupleConTy(countSome(","))
                                  | ParenTyOrTupleTy(sepBy1(`<type>`, ","))
                                  ) <~ ")"
                         )
```

Nice! One down, three to go. Let's have a look at the two involving patterns together:

```
lazy val `<pat-naked>`: Parsley[PatNaked] =
    ( `<var-id>` | atomic(`<pat-con>`)
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | NestedPatOrPatTuple("(" ~> sepBy1(`<pat>`, ",") <~ ")")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )
lazy val `<pat-paren>` = atomic(`<pat-app>`) | `<pat-naked>`
lazy val `<pat-app>` = PatApp(`<pat-con>`, some(`<pat-naked>`))
lazy val `<pat-con>` = ( "(" ~> ((ConsCon from ":") | TupleCon(countSome(",")))
 <~ ")"
                       | `<con-id>`
                       )
```

I've omitted the `<pat>` rule here, since it's not relevant. Right, so the interaction of these rules is quite intricate! The `atomic` in `<pat-paren>` is there because `<pat-app>` reads a `<pat-con>` and `<pat-naked>` can also read one of those. Additionally, within `<pat-naked>`, the `atomic` is guarding against yet more parentheses ambiguity caused by `ConsCon`, `TupleCon`, and `NestedPatOrPatTuple`. Now, our first thought

might be that we can eliminate the `atomic` within `<pat-naked>` with the same strategy we used for `<type-con>`; we certainly could, but `<pat-con>` is used in *two* places, so doing so will cause duplication in the parser. This is a trade-off: inlining `<pat-con>` will eliminate backtracking and make the parser more efficient, but that comes at the cost of increased code size; in this case, in fact, the backtracking is limited to a single character, which is relatively cheap, *and* the size of the rule is small, so inlining it will not increase the code size significantly. It doesn't really matter either way what we do, so let's reinforce our factoring skills and duplicate the code to eliminate the `atomic`!

```
lazy val `<pat-naked>`: Parsley[PatNaked] =
    ( `<var-id>` | `<con-id>`
    | atomic("(" ~> ((ConsCon from ":") | TupleCon(countSome(","))) <~ ")")
    | NestedPatOrPatTuple("(" ~> sepBy1(`<pat>`, ",") <~ ")")
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )
lazy val `<pat-paren>` = atomic(`<pat-app>`) | `<pat-naked>`
lazy val `<pat-app>` = PatApp(`<pat-con>`, some(`<pat-naked>`))
lazy val `<pat-con>` = ( "(" ~> ((ConsCon from ":") | TupleCon(countSome(",")))
 <~ ")"
                       | `<con-id>`
                       )
```

In the above parser, I inlined the parser and reorganised it to bring the offending sub-rules together. We know the drill by this point, let's factor that out:

```
lazy val `<pat-naked>`: Parsley[PatNaked] =
    ( `<var-id>` | `<con-id>`
    | "(" ~> ( (ConsCon from ":")
             | TupleCon(countSome(","))
             | NestedPatOrPatTuple(sepBy1(`<pat>`, ","))
             ) <~ ")"
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )
lazy val `<pat-paren>` = atomic(`<pat-app>`) | `<pat-naked>`
lazy val `<pat-app>` = PatApp(`<pat-con>`, some(`<pat-naked>`))
lazy val `<pat-con>` = ( "(" ~> ((ConsCon from ":") | TupleCon(countSome(",")))
 <~ ")"
                       | `<con-id>`
                       )
```

Nice, another `atomic` down! Now, what about the `atomic` in `<pat-paren>`? Well, it turns out that, by eliminating the first `atomic`, we've stopped ourselves from being able to deal with this one! The reason is that we've cannibalised the common `<pat-con>` structure into our factored parentheses in `<pat-naked>`: oops! This `atomic` is actually worse than the other one, since it can backtrack out of an entire `<pat-con>`

as opposed to just a single (. So, could we undo what we've just done and fix this one instead? We certainly could; but this transformation is very violent since <pat-naked> appears in other places in the parser. That being said, lets do it!

The first step is to return to our old parser:

```
lazy val `<pat-naked>`: Parsley[PatNaked] =
    ( `<var-id>` | atomic(`<pat-con>`)
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | NestedPatOrPatTuple("(" ~> sepBy1(`<pat>`, ",") <~ ")")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )
lazy val `<pat-paren>` = atomic(`<pat-app>`) | `<pat-naked>`
lazy val `<pat-app>` = PatApp(`<pat-con>`, some(`<pat-naked>`))
lazy val `<pat-con>` = ( "(" ~> ((ConsCon from ":") | TupleCon(countSome(",")))
 <~ ")"
                       | `<con-id>`
                       )
```

Now, we know that the <pat-con> is the problematic bit here, so let's break the <pat-naked> into two rules:

```
lazy val `<pat-naked>` = atomic(`<pat-con>`) | `<pat-naked'>`
lazy val `<pat-naked'>`: Parsley[PatNaked] =
    ( `<var-id>`
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | NestedPatOrPatTuple("(" ~> sepBy1(`<pat>`, ",") <~ ")")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )
lazy val `<pat-paren>` = atomic(`<pat-app>`) | atomic(`<pat-con>`) | `<pat-
naked'>`
lazy val `<pat-app>` = PatApp(`<pat-con>`, some(`<pat-naked>`))
lazy val `<pat-con>` = ( "(" ~> ((ConsCon from ":") | TupleCon(countSome(",")))
 <~ ")"
                       | `<con-id>`
                       )
```

Now, notice that I've inlined <pat-naked> into <pat-paren>. The reason I did this is to make it clear that the part we are trying to factor is the <pat-con>. In fact, let's do a little bit of shuffling and move it into <pat-app>:

```
lazy val `<pat-paren>` = atomic(`<pat-app>`) | `<pat-naked'>`
lazy val `<pat-app>` = atomic(PatApp(`<pat-con>`, some(`<pat-naked>`)))
 | `<pat-con>`
```

Now, the aim here is to smash those `<pat-con>`s together! We can introduce a new disambiguator bridge to handle this, and switch `some` for `many`:

```
object PatAppIfNonEmpty extends ParserBridge2[PatCon, List[PatNaked], PatParen]
 {
    def apply(con: PatCon, args: List[PatNaked]): PatParen = args match {
        case Nil => con
        case args => PatApp(con, args)
    }
}
```

Now, compared to the original `PatApp` bridge constructor, this one returns a `PatParen` instead of a `PatApp`. This is because that is the common supertype of `PatApp` and `PatCon`. Let's see what the parser looks like now:

```
lazy val `<pat-paren>` = atomic(`<pat-app>`) | `<pat-naked'>`
lazy val `<pat-app>` = PatAppIfNonEmpty(`<pat-con>`, many(`<pat-naked>`))
```

Now, since we've switched to a `many`, we can actually push both of our `atomic`s down into the `<pat-con>` and leave it at that:

```
lazy val `<pat-naked>` = `<pat-con>` | `<pat-naked'>`
lazy val `<pat-naked'>`: Parsley[PatNaked] =
    ( `<var-id>`
    | (UnitCon from "()") | (NilCon from "[]") | `<literal>` | (Wild from "_")
    | NestedPatOrPatTuple("(" ~> sepBy1(`<pat>`, ",") <~ ")")
    | PatList("[" ~> sepBy(`<pat>`, ",") <~ "]")
    )
lazy val `<pat-paren>` = `<pat-app>` | `<pat-naked'>`
lazy val `<pat-app>` = PatAppIfNonEmpty(`<pat-con>`, many(`<pat-naked>`))
lazy val `<pat-con>` = ( atomic("(" ~> ((ConsCon from ":")
 | TupleCon(countSome(","))) <~ ")")
                       | `<con-id>`
                       )
```

So, can we remove that last `atomic`? No. At least not without a collosal amount of very destructive refactoring of the grammar. What we can do, however, is make its scope ever so slightly smaller:

```
lazy val `<pat-con>` = ( atomic("(" ~> ((ConsCon from ":")
 | TupleCon(countSome(",")))) <~ ")"
                       | `<con-id>`
                       )
```

All I've done there is move it one parser to the left, that way, we commit to the branch as soon as we've seen either a `,` or a `:` and can't backtrack out of the closing bracket. That's as good as we're going to get. A little unsatisfying, perhaps, but it's really such a minor point.

So, now what? Well, we have one final `atomic` we can look at. And it's trickier than it looks.

```
val `<declaration>` = Decl(`<var-id>`, "::" ~> `<type>`)

val `<clause>` =
    Clause(`<var-id>`, many(`<pat-naked>`), option(`<guard>`), "=" ~> `<expr>`)

val `<program>` =
    fully(sepEndBy(`<data>` | atomic(`<declaration>`)
 | `<clause>`, some(NEWLINE)))
```

I've skipped out the irrelevant `<data>` parser here. So, from the outset, this `atomic` doesn't look so bad: both `<declaration>` and `<clause>` share a `<var-id>`. This, in theory, should be easy to factor out. The problem is the bridges: when we factor out `<var-id>` we no longer have the right shape to use their bridges. You might think that the solution is to introduce an `Either`, like we did with the `INT_OR_FLOAT` lexeme. This would work out ok, but is a little clunky. The new bridge factory would be dealing with `Either[Type, (List[PatNaked], Option[Expr], Expr)])]`. Let's start with this approach first, and then see an alternative that keeps the two bridges separate and avoids the tuple.

```
// These make use of `ast.Clause` because they are defined outside of `ast` (in
 this .md file)
type PartialClause = (List[PatNaked], Option[Expr], Expr)
object DeclOrClause extends ParserBridge2[VarId, Either[Type, PartialClause], ProgramUnit]
 {

  def apply(id: VarId, declOrClause: Either[Type, PartialClause]): ProgramUnit
 =
        declOrClause match {
            case Left(ty) => Decl(id, ty)
            case Right((args, guard, body))
 => ast.Clause(id, args, guard, body)
        }
}

object Clause extends ParserBridge2[VarId, PartialClause, Clause] {
    def apply(id: VarId, partialClause: PartialClause): Clause = {
        val (args, guard, body) = partialClause
        ast.Clause(id, args, guard, body)
    }
}
```

Now, to make this work nicely, I'm going to make use of the `<+>` combinator: pronounced "sum", this parser works like `|` except it returns its result into a co-product (`Either`). Using this, we can define the factored `<program>`:

```
import parsley.syntax.zipped.Zipped3

val `<declaration>` = "::" ~> `<type>`

val `<partial-clause>` = (many(`<pat-naked>`), option(`<guard>`), "="
 ~> `<expr>`).zipped
val `<clause>` = Clause(`<var-id>`, `<partial-clause>`)

val `<decl-or-clause>` = DeclOrClause(`<var-id>`, `<declaration>` <
+> `<partial-clause>`)
val `<program>` =
    fully(sepEndBy(`<data>` | `<decl-or-clause>`, some(NEWLINE)))
```

Why have we got two `<clause>`s? Well, we also need a `<clause>` for the `let`-expressions further down the parser. Now, there isn't anything wrong with this parser, and its a perfectly reasonable approach.

Let's also take a look at the other way we could have done this. This time, we'll change the two original bridge constructors, but won't introduce a third:

```
// These make use of `ast.Clause` because they are defined outside of `ast` (in
 this .md file)
object Decl extends ParserBridge1[Type, VarId => ast.Decl] {
    def apply(ty: Type): VarId => ast.Decl = ast.Decl(_, ty)
}
object Clause extends ParserBridge3[List[PatNaked], Option[Expr], Expr, VarId
 => ast.Clause] {
    def apply(pats: List[PatNaked], guard: Option[Expr], rhs: Expr): VarId
 => ast.Clause =
        ast.Clause(_, pats, guard, rhs)
}
```

This time, the two bridge constructors return functions that take the factored `VarId`. How does this change the parser?

```
val `<declaration>` = Decl("::" ~> `<type>`)

val `<partial-clause>` = Clause(many(`<pat-naked>`), option(`<guard>`), "="
 ~> `<expr>`)
val `<clause>` = `<var-id>` <**> `<partial-clause>`

val `<decl-or-clause>` = `<var-id>` <**> (`<declaration>` | `<partial-clause>`)
val `<program>` =
    fully(sepEndBy(`<data>` | `<decl-or-clause>`, some(NEWLINE)))
```

This also works completely fine. This time, we use `<**>` to apply the `<var-id>` to the partially built AST nodes. The advantage of this style is that it's a little more concise, and involves less unnecessary object construction. *However*, this version has a subtle flaw: suppose we wanted to add position information to the AST, then where would the `pos` go? The position information would be in the `var-id>`, and we'd have to extract it out in our bridges, for example:

```
object Decl extends ParserBridge1[Type, VarId => Decl] {
    def apply(ty: Type): VarId => Decl = v => Decl(v, ty)(v.pos)
}
object Clause extends ParserBridge3[List[PatNaked], Option[Expr], Expr, VarId
 => Clause] {
    def apply(pats: List[PatNaked], guard: Option[Expr], rhs: Expr): VarId
 => Clause =
        v => Clause(v, pats, guard, rhs)(v.pos)
}
```

This is really brittle: it is relying on the `VarId` type having *and exposing* its position information. In contrast, here's how our other bridge factory would be transformed:

```
object DeclOrClause
    extends ParserBridgePos2[VarId, Either[Type, PartialClause], ProgramUnit] {
    def apply(id: VarId, declOrClause: Either[Type, PartialClause])
             (pos: (Int, Int)): ProgramUnit = declOrClause match {
        case Left(ty) => Decl(id, ty)(pos)
        case Right((args, guard, body)) => Clause(id, args, guard, body)(pos)
    }
}
```

Much more straightforward, with no effect on any other bridge! This is worth considering if you do find yourself in this sort of position (no pun intended). That being said, the `<**>`-based version is slightly cleaner and a *tiny* bit more efficient. Regardless of which method we pick, however, that pesky `atomic` is gone! That leaves us in a final state with a *single* `atomic` which has a maximum backtrack of one character. In other words, this parser runs in linear-time. Excellent!

## 5 Concluding Thoughts

In this (rather long) page, we've explored the implementation of an entire parser for a subset of Haskell from *scratch*. We've also seen a few techniques for factoring out common branches of the parser and applied everything we've learnt so far to a real example. We are going to come back to this parser later in the series: we've got to add better error messages, and deal with Haskell's indentation-sensitive off-side rule!

# Customising Error Messages

Previously, in **Effective Lexing** we saw how we could extend our parsers to handle whitespace and lexing. In this wiki post we'll finally address error messages. Thoughout all the other entries in this series I have neglected to talk about error messages at all, but they are a very important part of a parser.

## 1 Adjusting Error Content

I'm going to start with the parser from last time, but before we introduced the `Lexer` class. The reason for this is that the `Lexer` functionality has error messages baked into it, which means this post would be even shorter! It's not perfect, however, but it does make some good error messages for your basic lexemes. There is nothing stopping you from using the techniques here to change those messages if you wish though. Simply put: the original grammar has more room for exploration for us.

```
import parsley.Parsley, Parsley.atomic

object lexer {
    import parsley.Parsley.{eof, many}
    import parsley.character.{digit, whitespace, string, item, endOfLine}
    import parsley.combinator.manyTill

    private def symbol(str: String) = atomic(string(str)).void
    private implicit def implicitSymbol(tok: String) = symbol(tok)

    private val lineComment = "//" ~> manyTill(item, endOfLine).void
    private val multiComment = "/*" ~> manyTill(item, "*/").void
    private val comment = lineComment | multiComment
    private val skipWhitespace = many(whitespace.void | comment).void

    private def lexeme[A](p: =>Parsley[A]) = p <~ skipWhitespace
    private def token[A](p: =>Parsley[A]) = lexeme(atomic(p))
    def fully[A](p: =>Parsley[A]): Parsley[A] = skipWhitespace ~> p <~ eof

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

    object implicits {
        implicit def implicitSymbol(s: String): Parsley[Unit]
 = lexeme(symbol(s))
    }
}

object expressions {
    import parsley.expr.{precedence, Ops, InfixL}

    import lexer.implicits.implicitSymbol
    import lexer.{number, fully}
```

```
    private lazy val atom: Parsley[BigInt] = "(" ~> expr <~ ")" | number
    private lazy val expr = precedence[BigInt](atom)(
        Ops(InfixL)("*" as (_ * _)),
        Ops(InfixL)("+" as (_ + _), "-" as (_ - _)))

    val parser = fully(expr)
    def parse(input: String) = parser.parse(input)
}
```

So, before, we saw how this ran on succesful cases. Let's now start to see how it works on *bad* input.

```
expressions.parse("5d")
// res0: parsley.Result[String, BigInt] = Failure((line 1, column 2):
//   unexpected "d"
//   expected "*", "+", "-", "/*", "//", digit, end of input, or whitespace
//   >5d
//     ^)
```

Let's start by breaking this error down first and understanding what the components of it are and why this information has appeared. The first line of the error reports the line and column number of the message (in Parsley hard tabs are treated as aligning to the nearest 4th column). If you are using `parseFromFile` then this will also display the filename. The last two lines always show the location at which the error occured. This is going to be *the point* at which the error that eventually ended up being reported occured, *not* necessarily where the parser ended up. This can be improved in the future. Next you can see the *unexpected* and *expected* clauses. The unexpected "d" here is telling us roughly what we already knew. The expected clause on the other hand tells us all the things we could have used to fix our problem. There is definitely a lot of noise here though.

First let's just make sure we understand where each of these alternatives came from. Firstly, it's clear that since the last thing we read was a 5, a good way of carrying on would be reading another digit to make the number bigger. We could also read a space or start a comment as a way of making more progress too. Of course, another way we could make progress would have been using one of the operators and in the process continued our expression. Finally we could simply remove the d and it would run perfectly fine. Notice how `(` and `)` are not suggested as alternatives despite appearing in the parser: `5(` or `5)` makes no sense either. As another small example, let's see what happens to the error if we add a space between the 5 and the d.

```
expressions.parse("5 d")
// res1: parsley.Result[String, BigInt] = Failure((line 1, column 3):
//   unexpected "d"
//   expected "*", "+", "-", "/*", "//", end of input, or whitespace
//   >5 d
//      ^)
```

Neat, so this time round `digit` is no longer a valid alternative: clearly the number has come to an end because we wrote a space. But the other possibilities from before are still valid. So, how can we start making improvements? There are seven core combinators available to us for this purpose:

- `.label` or `?` is the most common combinator you'll be using. It influences the way an *expected* behaves for the parser it annotates. Importantly, if the parser it is annotating failed *and* consumed input in the process, then the label will not be applied. We'll see an example of why this is useful later.
- `fail` is useful, but a bit of a sledgehammer. When `fail` (or any of its derivative combinators like `guardAgainst`) is used, it removes the unexpected and expected information and just replaces it with a given message. If there are multiple `fail`s that appear in the same message, they are each written on a newline.
- `unexpected` is the least commonly used combinator in practice. When it is used, it will, like `fail`, immediately fails except it reports a custom unexpected message. Currently, only one unexpected message can be present in the error at once, so this is not very useful unless you really know what you are doing.
- `.hide` is a method that removes the output of a parser from the error.
- `.explain` is a method that can provide *reasons* for a parser's failure. If the parser can recover and move onto other alternatives, the reasons may be lost. But they can still be quite nice when used in the correct place!
- `amend` and `entrench` are a pair of combinators that work together to correct the position of some error messages. These are quite rare in practice.

All of these can be found in the `parsley.errors.combinator` module.

## 1.1 Using `label`

From this section, we are only going to be using `label` and `hide`, as they are by far the most useful and effective of the five methods. That being said, `explain` can be *very* useful, but we'll find there are no compelling use-cases for it in this example. Let's start off by giving a label to `comment` and see what happens:

```
import parsley.errors.combinator._

val comment = (lineComment | multiComment).label("comment")
```

Now let's run our parser from before:

```
expressions.parse("5d")
// res3: parsley.Result[String, BigInt] = Failure((line 1, column 2):
//   unexpected "d"
//   expected "*", "+", "-", comment, digit, end of input, or whitespace
//   >5d
//     ^)
```

Nice! So, if you compare the two, you'll notice that `"/*"` and `"//"` both disappeared from the message, but `comment` was added. You can tell when `label` is being used because there are not quotes surrounding the items. Knowing this, you can probably guess that `digit`, `eof`, and `whitespace` all have error labels of their own.

### 1.1.1 Using `hide` to trim away junk

This is a good start, but normally we might say that whitespace suggestions in an error message are normally just noise: of course we expect to be able to write whitespace in places, it's not *usually* the solution to someone's problem. This makes it a good candidate for the `hide` combinator:

```
import parsley.errors.combinator._

val skipWhitespace = many(whitespace.void | comment).void.hide
```

Now let's check again:

```
expressions.parse("5d")
// res4: parsley.Result[String, BigInt] = Failure((line 1, column 2):
//   unexpected "d"
//   expected "*", "+", "-", digit, or end of input
//   >5d
//     ^)
```

Great! The `hide` combinator has removed the information from the error message, and now it's looking a lot cleaner. But what if we started writing a comment, what would happen then?

```
expressions.parse("5/*")
// res5: parsley.Result[String, BigInt] = Failure((line 1, column 4):
//   unexpected end of input
//   expected "*/" or any character
//   >5/*
//       ^)
```

So, as I mentioned earlier, `hide` is just a `label`, and `label` will not relabel something if it fails and consumes input. That means, by opening our comment but not *finishing* it, we can see some different suggestions. In

this case, end of input is not allowed, and any character will work to extend the comment, but clearly `*/` is a way to properly end it. Let's add a label to that, however, to make it a bit friendlier:

```
val lineComment = "//" *> manyTill(item, endOfLine.label("end of
 comment")).void
val multiComment = "/*" *> manyTill(item, "*/".label("end of comment")).void
```

Now we get a more informative error message of:

```
expressions.parse("5/*")
// res6: parsley.Result[String, BigInt] = Failure((line 1, column 4):
//   unexpected end of input
//   expected any character or end of comment
//   >5/*
//       ^)
```

Great! Now let's turn our attention back to expressions and not whitespace.

### 1.1.2 Labelling our numbers

Let's take a look at a very simple bad input and see how we can improve on it:

```
expressions.parse("d")
// res7: parsley.Result[String, BigInt] = Failure((line 1, column 1):
//   unexpected "d"
//   expected "(" or digit
//   >d
//    ^)
```

So this time, we can see two possible ways of resolving this error are opening brackets, or a **`digit`**. Now **`digit`** is really a poor name here, what we really mean is **`integer`** or **`number`**:

```
val number =
    token(digit.foldLeft1[BigInt](0)((n, d) => n * 10
 + d.asDigit)).label("number")
```

Now we get, the following, nicer error message:

```
expressions.parse("d")
// res8: parsley.Result[String, BigInt] = Failure((line 1, column 1):
//   unexpected "d"
//   expected "(" or number
//   >d
//    ^)

expressions.parse("5x")
```

```
// res9: parsley.Result[String, BigInt] = Failure((line 1, column 2):
//   unexpected "x"
//   expected "*", "+", "-", digit, or end of input
//   >5x
//     ^)
```

But notice in the second error message, again we have been given `digit` and not `number` as our alternative. This is good, once we've started reading a number by reading `5` it would be inappropriate to suggest a number as a good next step. But `digit` here is not particularly descriptive and we can do better still:

```
val number =
    token(
        digit.label("end of number").foldLeft1[BigInt](0)((n, d) => n * 10
 + d.asDigit)
    ).label("number")
```

This gives us, again, a much nicer message:

```
expressions.parse("5x")
// res10: parsley.Result[String, BigInt] = Failure((line 1, column 2):
//   unexpected "x"
//   expected "*", "+", "-", end of input, or end of number
//   >5x
//     ^)
```

### 1.1.3 Merging multiple labels

With an example grammar as small as this, I think we are almost done here! The last thing we could improve is the repetition of `"*"`, `"+"`, and `"-"`. Really, we know that there is nothing special about any of them individually, so we could more concisely replace this them with `arithmetic operator`, or since we only have arithmetic operators here `operator` will do. we don't need to do anything special here, when multiple labels are encountered with the same name, they will only appear once!

```
lazy val expr = precedence[BigInt](atom)(
  Ops(InfixL)("*".label("operator") as (_ * _)),
  Ops(InfixL)("+".label("operator") as (_ + _), "-".label("operator") as (_
 - _)))
```

Now we arrive at our final form:

```
expressions.parse("5x")
// res11: parsley.Result[String, BigInt] = Failure((line 1, column 2):
//   unexpected "x"
//   expected end of input, end of number, or operator
//   >5x
```

```
//     ^)

expressions.parse(" 67 + ")
// res12: parsley.Result[String, BigInt] = Failure((line 1, column 7):
//   unexpected end of input
//   expected "(" or number
//   > 67 +
//          ^)
```

Great! Now obviously you could take this even further and make `"("` become `opening parenthesis` or something, but I don't really feel that adds much.

## 1.2 Wrapping up the Expression Example

Hopefully, you get a sense of how much of an art form and subjective writing good error messages is, but Parsley provides decent error messages out of the box (now based on `megaparsec`'s error messages from Haskell). It doesn't have to be hard though, so just play around and see what feels right. I would say, however, there is an interesting phenomenon in the programming languages and compilers community: **compiler writers write error messages that are tailored for compiler writers**. It's an interesting problem when you think about it: the person who writes error messages is a compiler expert, and so they often rely on the concepts they understand. That means they are more prone to including the names of stuff in the grammar to describe syntax problems, and so on. While this is great for experts and compiler writers, it seemingly forgets people who are new to programming or this "grammar" in particular. That can make error messages needlessly intimidating for the average Joe. The take home from this is to try and avoid labelling `expr` with `.label("expression")`, because that just ends up making something that is no longer useful or informative:

```
expressions.parse("")
// res13: parsley.Result[String, BigInt] = Failure((line 1, column 1):
//   unexpected end of input
//   expected expression
//   >
//    ^)
```

What use is that to anybody? The same idea applies to statements, and various other abstract grammatical notions. Something like `"expected if statement, while loop, for loop, variable declaration, or assignment"` is so much more meaningful than `"expected statement"`. I would ask that you keep that in mind #. To conclude our work with this parser, here is the full code of the finished product. Obviously, with the `Lexer`, some of this work is already done, but you can still apply the lessons learnt here to the wider parser!

```
import parsley.Parsley, Parsley.{atomic, eof, many}
import parsley.errors.combinator._

object lexer {
```

```
    import parsley.character.{digit, whitespace, string, item, endOfLine}
    import parsley.combinator.manyTill

    private def symbol(str: String) = atomic(string(str)).void
    private implicit def implicitSymbol(tok: String) = symbol(tok)

    private val lineComment = "//" ~> manyTill(item, endOfLine).void.label("end
 of comment")
    private val multiComment = "/*" ~> manyTill(item, "*/").void.label("end of
 comment")
    private val comment = (lineComment | multiComment).label("comment")
    private val skipWhitespace = many(whitespace.void | comment).void.hide

    private def lexeme[A](p: =>Parsley[A]) = p <~ skipWhitespace
    private def token[A](p: =>Parsley[A]) = lexeme(atomic(p))
    def fully[A](p: =>Parsley[A]): Parsley[A] = skipWhitespace ~> p <~ eof

    val number: Parsley[BigInt] = token {
        digit.label("end of number").foldLeft1[BigInt](0)((n, d) => n * 10
 + d.asDigit)
    }.label("number")

    object implicits {
        implicit def implicitSymbol(s: String): Parsley[Unit]
 = lexeme(symbol(s))
    }
}

object expressions {
    import parsley.expr.{precedence, Ops, InfixL}

    import lexer.implicits.implicitSymbol
    import lexer.{number, fully}

    private lazy val atom: Parsley[BigInt] = "(" ~> expr <~ ")" | number
    private lazy val expr = precedence[BigInt](atom)(
        Ops(InfixL)("*".label("operator") as (_ * _)),
        Ops(InfixL)("+".label("operator") as (_ + _), "-".label("operator") as
 (_ - _)))

    val parser = fully(expr)
    def parse(input: String) = parser.parse(input)
}
```

## 1.3 Using `explain`

So far, we've seen how **`label`** can be used to clean up error messages and make them much more presentable and informative. Another way of achieving this is by using the **`explain`** combinator. Unlike **`label`** this is much more freeform and when used properly can be *incredibly* effective. Essentially, with

`explain` you are leveraging your own knowledge about the context you are in to provide a much more tailored and hand-crafted message to the user. It can be used to both provide an additional hint in an otherwise poor message or to enrich the error with suggestions for how the error might be fixed.

Using it is just as easy as using `label` and you can't really go wrong with it: other than being a bit... too descriptive. Again, the `Lexer` class already makes use of this technique to improve its own error messages, but let's suppose we wanted to write some of its functionality ourselves. Let's cook up a string literal parser, supporting some (limited) escape sequences.

```
import parsley.Parsley
import parsley.syntax.character.charLift
import parsley.combinator.choice
import parsley.character._
import parsley.errors.combinator._

val escapeChar =
    choice('n' as '\n', 't' as '\t', '\"', '\\')
val stringLetter =
    noneOf('\"', '\\').label("string character") |
    ('\\' ~> escapeChar).label("escape character")

val stringLiteral =
    ('\"' ~> stringOfMany(stringLetter) <~ '\"'.label("end of
 string")).label("string")
```

Let's start with something like this. If we run a couple of examples, we can see where it performs well and where it performs less well:

```
stringLiteral.parse("")
// res16: parsley.Result[String, String] = Failure((line 1, column 1):
//   unexpected end of input
//   expected string
//   >
//    ^)

stringLiteral.parse("\"")
// res17: parsley.Result[String, String] = Failure((line 1, column 2):
//   unexpected end of input
//   expected end of string, escape character, or string character
//   >"
//     ^)

stringLiteral.parse("\"\\a")
// res18: parsley.Result[String, String] = Failure((line 1, column 3):
//   unexpected "a"
//   expected """, "\", "n", or "t"
//   >"\a
//      ^)
```

So, for the first two cases, the error message performs quite well. But the last message is a bit noisy. One possible approach to improve this could be to label each alternative to give them a slightly clearer name, which would result in something like:

```
stringLiteral.parse("\"\\a")
// res19: parsley.Result[String, String] = Failure((line 1, column 3):
//   unexpected "a"
//   expected \", \\, \n, or \t
//   >"\a
//      ^)
```

This is *better*, but a bit misleading, we don't expect a \! Now, you could instead opt to remove the backslashes, but then that doesn't give much information about why these things are expected. Another option would be to label all alternatives with some common name:

```
val escapeChar =
    choice('n' as '\n', 't' as '\t', '\"', '\\')
        .label("end of escape sequence")
```

Which would yield

```
stringLiteral.parse("\"\\a")
// res20: parsley.Result[String, String] = Failure((line 1, column 3):
//   unexpected "a"
//   expected end of escape sequence
//   >"\a
//      ^)
```

This is a bit more helpful, in that it does provide a good name to what we expected. But at the same time it doesn't help the user to understand how to fix their problem: "what is an escape sequence". This is similar to the "statement" problem I described above. In this case, (and indeed in the "statement" case), we can add an `explain` to help the user understand what we mean:

```
val escapeChar =
    choice('n' as '\n', 't' as '\t', '\"', '\\')
        .label("end of escape sequence")
        .explain("valid escape sequences include \\n, \\t, \\\", or \\\\")
```

The `explain` combinator annotates failure messages with an additional reason. These can stack, and are displayed each on their own line in the error message. With this in place, let's see what the new error message is:

```
stringLiteral.parse("\"\\a")
// res21: parsley.Result[String, String] = Failure((line 1, column 3):
//   unexpected "a"
//   expected end of escape sequence
//   valid escape sequences include \n, \t, \", or \\
//   >"\a
//      ^)
```

This time, we keep the name of the expected token clear and concise, but we *also* help the user to understand what this actually means. The error isn't misleading in the sense that we aren't suggesting that a `\n` would fix the parse error *after* the `\` we already wrote, but have have said that we expect the end of the escape as well as demonstrated what that would look like. This is great!

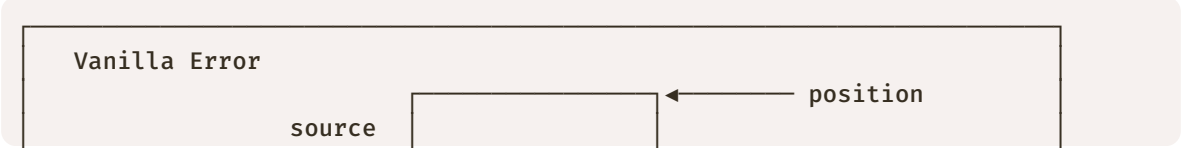
There isn't much more to say about the `explain` combinator than that really. Hopefully this already gives you a sense of how useful it can be. Like I mentioned before, the poor error problem that compiler writers often suffer from can be nicely solved using `explain`. For instance, a message like "... expected statement ... valid statements include 'if statements', 'loops', or 'assignments'" is subjectively better than both of the alternatives (namely "expected statement" or the one that lists out every single alternative). This has the benefits of both worlds: for an experienced user, the error message gets straight to the point, and for the newcomer, the error message provides a bit more information that can help them learn the terminology.

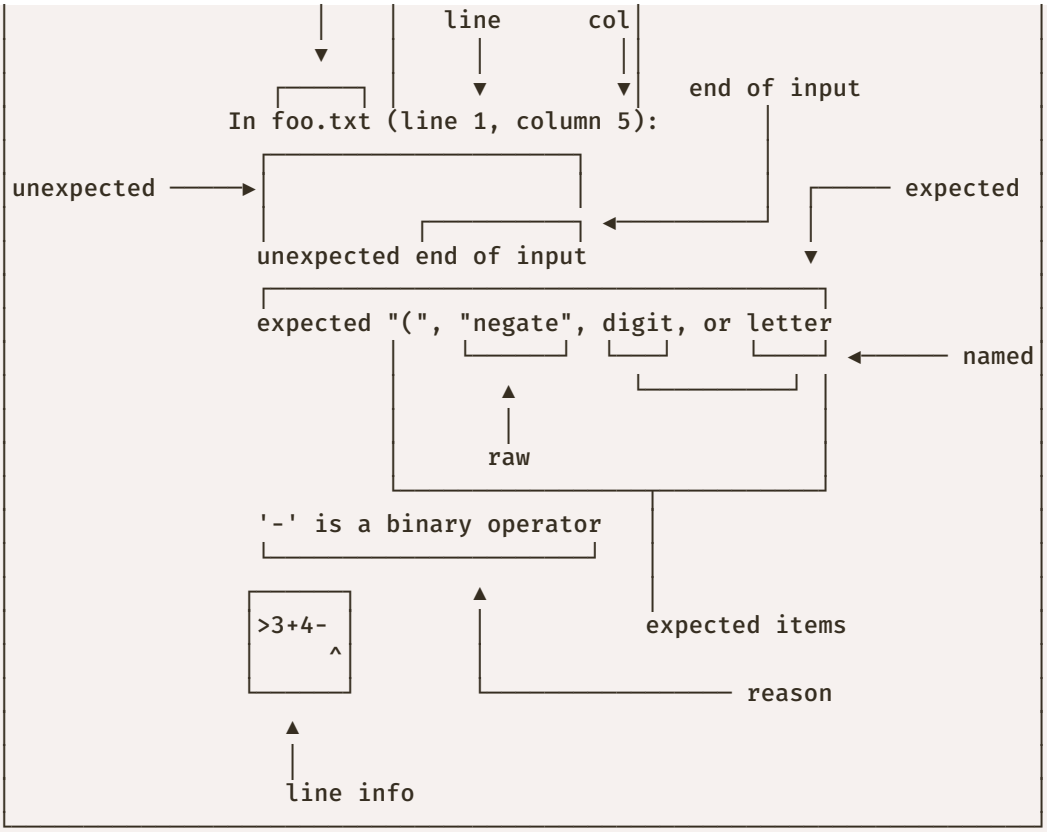
## 2 Adjusting Error Formatting

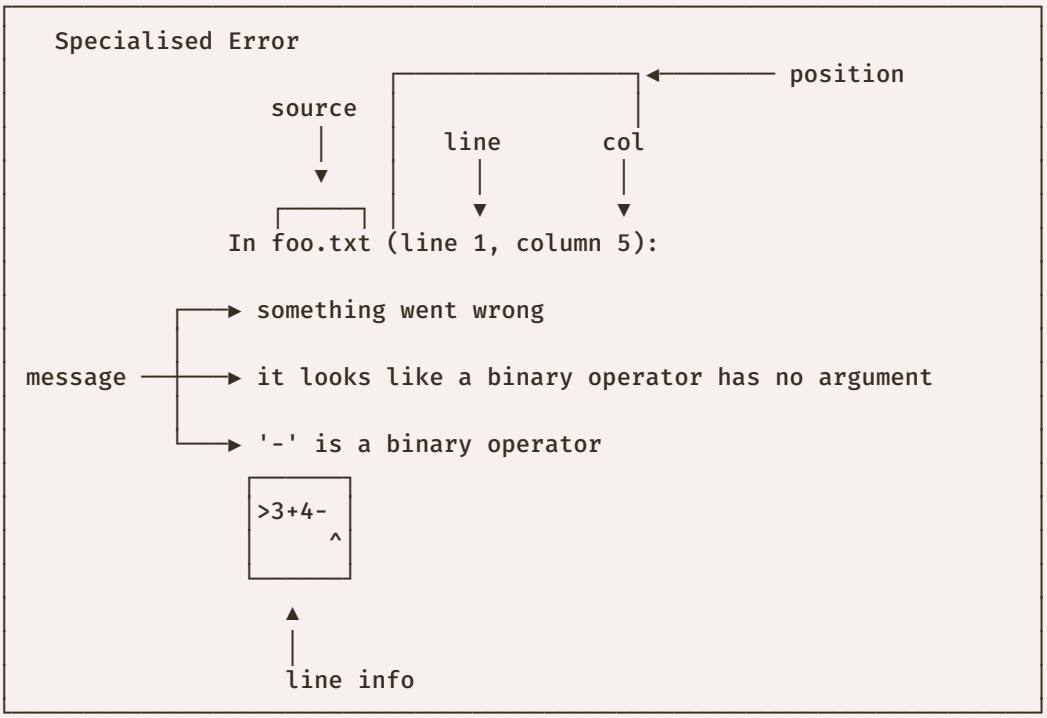
As we've seen in this post, the error messages produced by parsley are fairly readable. They are broken into two kinds: "vanilla" errors built up of "expected", "unexpected", and "reason" clauses; and "specialised" errors built up solely of "message" clauses. So far, we have only seen examples of the "vanilla" errors, and we will see the "specialised" errors in the next post. These have been so far formatted using Parsley's default mechanism, which creates an error as a string. This is ok for basic use, but in projects where there is some pre-existing error format, then maintaining consistency across error messages is much harder without *parsing* the resulting `String` errors to extract their content: this is, frankly, ridiculous to expect! Moreover, suppose you wanted to unit test your parser in both successful and failing cases, then performing raw string comparision is really brittle, especially if Parsley adjusts the format slightly!

Luckily, Parsley 3.0.0 introduced an abstraction layer between the error messages that the parsers work with and the final resulting error message. This means that actually, the error message format is not only configurable, but doesn't *have* to be a `String`! The final part of this post is dedicated to understanding how to work with this mechanism, using Parsley's own unit test formatter as an example.

Firstly, I want to give examples of both types of format, and annotate the names given to each part of them:






As you can see, the content for a specialised error is (ironically) plainer than a vanilla message. This means that the errors are much more customisable from the parser side, but it is less rich in parser generated

information than the vanilla is. Hopefully you can see that both error messages still have a *very* similar shape other than the error info lines themselves. In both cases, and not shown by the diagrams, the main contents of the error -- either unexpected, expected, reasons, and line info; or messages and line info -- are called "error info lines".

For vanilla errors, notice that the unexpected and expected lines make references to raw, named, and end of input: these are collectively known as *items*. The `.label` combinator produces named items, the `eof` combinator produces the end of input item, and unlabelled combinators produce raw items.

Together, all these components are referenced (by these names!) by the `ErrorBuilder` trait. The way it works is that a concrete `ErrorBuilder` has to be provided to the `.parse` method of a parser, and when the parser has finished failing, the builder is used to construct the final error message, converting the internal representation that Parsley uses into the desired output specified by the builder: you can think of it like a conversation. The internals of Parsley take a portion of the information it has, and talks to the builder how to format it into another intermediate form; it then will feed this new information into another method of the builder after possibly more collection. To allow all of this plumbing to be fed together and maintain maximum flexiblity to the user, the builder makes use of "associated types". Let's take a look at the definition of `ErrorBuilder` without all the sub-formatters to understand what I mean:

```
trait ErrorBuilder[Err] {
    // This is the top level function which takes all the sub-parts
    // and combines them into the final `Err`
    def format(pos: Position, source: Source, lines: ErrorInfoLines): Err

    type Position
    type Source
    type ErrorInfoLines
    type ExpectedItems
    type Messages
    type UnexpectedLine
    type ExpectedLine
    type Message
    type LineInfo
    type Item
    type Raw <: Item
    type Named <: Item
    type EndOfInput <: Item

    ...
}
```

Wow, that's a lot of types! Essentially, each concrete implementation of this trait must specify what each of those types are. This means that the representation of the error is as flexible as possible. In the `format` method, you can see that the types `Position`, `Source`, and `ErrorInfoLines` are all referenced. Indeed, you can also see these marked on *both* diagrams: in other words, `format` is responsible for the general shape of *both* types of error message.

To understand how these might come about, let's take a step "into" the formatter to find the sources of values for `Position`, `Source`, and `ErrorInfoLines`:

```
trait ErrorBuilder[Err] {
    ...

    def pos(line: Int, col: Int): Position
    def source(sourceName: Option[String]): Source

    def vanillaError(unexpected: UnexpectedLine, expected: ExpectedLine,
                     reasons: Messages, line: LineInfo): ErrorInfoLines
    def specialisedError(msgs: Messages, line: LineInfo): ErrorInfoLines

    ...
}
```

Hopefully, you can start to see how this might be structured:

- To get a `Position` value for the final error message, the line and column information is fed *straight* from the parser into the `pos` method, which can then hand back the "formatted" position.
- To get the `Source` name, the filename (if it exists!) is fed into the `source` method, which can then be fed into `format` by the internals of Parsley.
- To collect up all the `ErrorInfoLines` there are two possible approaches depending on whether the error is vanilla or specialised. In both cases, the relevant information is passed in and can be "formatted" into whatever `ErrorInfoLines` is: for instance, the default in Parsley has `type ErrorInfoLines = Seq[String]`. Neither of these two methods take raw information from the parser, they have clearly been fed through another part of the formatter, given their types.

I won't continue traversing deeper and deeper into the system, because it's just going to be the same idea over and over again. But I will note all the "terminal" methods that do take information directly from the parser:

```
trait ErrorBuilder[Err] {
    ...

    def pos(line: Int, col: Int): Position
    def source(sourceName: Option[String]): Source

    def reason(reason: String): Message
    def message(msg: String): Message
    def lineInfo(line: String, linesBefore: Seq[String],
                 linesAfter: Seq[String], errorPointsAt: Int): LineInfo
    val numLinesBefore: Int
    val numLinesAfter: Int
```

```
    def raw(item: String): Raw
    def named(item: String): Named
    val endOfInput: EndOfInput

    def unexpectedToken(cs: Iterable[Char], amountOfInputParserWanted: Int,
                        lexicalError: Boolean): Token
}
```

The two attributes `numLinesBefore` and `numLinesAfter` are used by the Parsley internals to decide how many raw lines of input both before and after the problematic line to provide to `lineInfo`. In a pinch, overriding these values from `DefaultErrorBuilder` is a quick way of changing how specific your errors are to other lines in the input. The `unexpectedToken` method is special, but I'll leave a discussion of this **another page**. All of the other methods in the `ErrorBuilder` will make use of the refined results from the methods above.

I hope that, by this point, you have a reasonable idea of how this system all ties together. But, if you don't, or you want an example, take a look at how `parsley`'s own unit tests format error messages to be easier to pattern match on and test against, the implementation can be found **here**.

# Advanced Error Messages

Previously, we saw the most basic approach to improving error messages: `.label` and `.explain`. However, the other tools I listed in the post are valuable in their own right, but can be slightly less common. However, most of the time, their use is abstracted by other higher-level combinators, which will be the focus of this page. The API Guide page on **Error Message Combinators** covers the core combinators more thoroughly.

## 1 A Statement Language

In this section, we'll set the stage for the discussion of the error messages. We'll start by defining an evaluator for a langauge (as opposed to an AST), and then write a parser that generates partially evaluated expressions and collapse it to form an interpreter. The language supports conditionals, assignment, arithmetic, and boolean expressions; variables must be integers.

### 1.1 A Stateful Evaluator

The language will be evaluated by producing a value of type `Eval[Unit]`, a monad that threads a variable environment through the program, handling any out-of-scope variables. Just for fun, I'll use the `cats` functional programming library to do this, as it makes it very easy to express. The environment will be carried around in a `StateT` state monad, and errors contains within an `Either[String, _]` type. Don't worry if you don't understand how it works; our main focus is obviously the **parser**:

```
import cats.syntax.all._
object eval {
    import cats.data.StateT
    import cats.Monad
    type Error[A] = Either[String, A]
    type Eval[A] = StateT[Error, Map[String, Int], A]

    def number(x: Int): Eval[Int] = Monad[Eval].pure(x)
    def bool(b: Boolean): Eval[Boolean] = Monad[Eval].pure(b)

    def negate(mx: Eval[Int]): Eval[Int] = mx.map(0 - _)
    def add(mx: Eval[Int], my: Eval[Int]): Eval[Int] = (mx, my).mapN(_ + _)
    def sub(mx: Eval[Int], my: Eval[Int]): Eval[Int] = (mx, my).mapN(_ - _)
    def mul(mx: Eval[Int], my: Eval[Int]): Eval[Int] = (mx, my).mapN(_ * _)

    def less(mx: Eval[Int], my: Eval[Int]): Eval[Boolean] = (mx, my).mapN(_
 < _)
    def equal(mx: Eval[Int], my: Eval[Int]): Eval[Boolean] = (mx, my).mapN(_
 == _)

    def and(mx: Eval[Boolean], my: Eval[Boolean]): Eval[Boolean]
 = cond(mx, my, bool(false))
    def or(mx: Eval[Boolean], my: Eval[Boolean]): Eval[Boolean]
 = cond(mx, bool(true), my)
```

```
    def not(mx: Eval[Boolean]): Eval[Boolean] = mx.map(!_)

    def ask(v: String): Eval[Int] =
        StateT.inspectF(_.get(v).toRight(s"variable $v out of scope"))
    def store(v: String, mx: Eval[Int]): Eval[Unit] =
        mx.flatMap(x => StateT.modify(_.updated(v, x)))

    def cond[A](b: Eval[Boolean], t: Eval[A], e: Eval[A]): Eval[A] =
        Monad[Eval].ifM(b)(t, e)
}
```

The `ask` and `store` operations above allow for interaction with the environment, and the other operations are just "lifting" the relevant operations into our monad. Just so you can see how this might be stitched together, here are some example programs:

```
import eval._
store("x", add(number(5), ask("v"))).runS(Map("v" -> 3))
// res0: Error[Map[String, Int]] = Right(Map(v -> 3, x -> 8))
store("x", add(number(5), ask("v"))).runS(Map.empty)
// res1: Error[Map[String, Int]] = Left(variable v out of scope)

List(
    store("v", number(3)),                 // v = 3;
    store("x", add(number(5), ask("v"))), // x = 5 + v;
    cond(less(ask("x"), ask("v")),         // if x < v {
        store("v", ask("x")),              //   v = x } else {
        store("v", number(6))),            //   v = 6 };
).sequence.runS(Map.empty)
// res2: Either[String, Map[String, Int]] = Right(Map(v -> 6, x -> 8))
```

If we start off with an initial variable assignment of `{v = 3}`, then the `Eval` value representing `x = 5 + v` will finish with a final environment of `{v = 3, x = 8}`. If we don't provide an initial value to `v`, the evaluation errors with an "out of scope" error. The `sequence` method can be used to compose multiple statements in the langauge.

## 1.2 A Parser

We'll be using the same `lexer` as we've been accustomed to recently (with some extra keywords and operators), so let's see what the parser is like:

```
import parsley.Parsley.atomic
import lexer.implicits.implicitSymbol
import lexer.{number, identifier, fully}
import parsley.combinator.sepEndBy
import parsley.syntax.zipped.{Zipped2, Zipped3}
import parsley.expr.{Prefix, InfixR, InfixL, precedence, Ops}
```

```
def infixN[A, B](p: Parsley[A])(op: Parsley[(A, A) => B]): Parsley[B] =
    (p, op, p).zipped((x, f, y) => f(x, y))

lazy val atom: Parsley[Eval[Int]] =
    "(" ~> expr <~ ")" | number.map(eval.number) | identifier.map(ask)
lazy val expr = precedence[Eval[Int]](atom)(
    Ops(Prefix)("negate" as negate),
    Ops(InfixL)("*" as mul),
    Ops(InfixL)("+" as add, "-" as sub))

lazy val blit: Parsley[Eval[Boolean]] = "true".as(bool(true))
 | "false".as(bool(false))
lazy val comp: Parsley[Eval[Boolean]] = infixN(expr)("<".as(less)
 | "==".as(equal))
lazy val btom: Parsley[Eval[Boolean]] = atomic("(" ~> pred) <~ ")" | blit
 | comp
lazy val pred = precedence[Eval[Boolean]](btom)(
    Ops(Prefix)("not" as not),
    Ops(InfixR)("&&" as and),
    Ops(InfixR)("||" as or))

def braces[A](p: =>Parsley[A]) = "{" ~> p <~ "}"

lazy val asgnStmt: Parsley[Eval[Unit]] = (identifier, "="
 ~> expr).zipped(store)
lazy val ifStmt: Parsley[Eval[Unit]] =
    ("if" ~> pred, braces(stmts), "else" ~> braces(stmts)).zipped(cond[Unit])
lazy val stmt: Parsley[Eval[Unit]] = asgnStmt | ifStmt
lazy val stmts: Parsley[Eval[Unit]] = sepEndBy(stmt, ";").map(_.sequence.void)

val parser = fully(stmts)
```

I'm not going to say too much about this, since all of the ideas have been covered in previous pages (and the Haskell example!). The evaluator and the parser can be stitched together to form an interpreter:

```
def interpret(input: String)(ctx: Map[String, Int]) = {
    parser.parse(input).toEither.flatMap(_.runS(ctx))
}
```

Let's run an example through it and get used to the syntax:

```
interpret(
    """x = 7;
      |if x < v && 5 < v {
      |  y = 1;
      |}
      |else {
      |  y = 0;
      |};
      |x = 0;
    """.stripMargin)(Map("v" -> 8))
// res3: Either[String, Map[String, Int]] = Right(Map(v -> 8, x -> 0, y -> 1))
```

The `|`s here are being used by Scala's `stripMargin` method to remove the leading whitespace on our multi-line strings. We can see that the syntax of this language makes use of semi-colons for delimiting, and `if` statements require a semi-colon after the `else`. In addition, no parentheses are required for the `if`, but braces are mandated. This syntax is a little unorthodox, which is great for us because it'll give us a lot of opportunities to test out our new tools! I've neglected to add any `.label`s or `.explain`s here, but obviously there are plenty of opportunities.

## 1.3 Motivation

Let's start by seeing what happens if we accidentally write a `;` before an `else`:

```
interpret("if true {}; else {};")(Map.empty)
// res4: Either[String, Map[String, Int]] = Left((line 1, column 11):
//   unexpected ";"
//   expected else
//   >if true {}; else {};
//              ^)
```

This is what we'd expect, since at this point we'd expect an `else`. We could detail this issue for the user with an `.explain`, and explain that semi-colons are not something that work in this position:

```
import parsley.errors.combinator._
lazy val ifStmt: Parsley[Eval[Unit]] =
    ( "if" ~> pred
    , braces(stmts)
    , "else".explain("semi-colons cannot be written after ifs")
 ~> braces(stmts)
    ).zipped(cond[Unit])
```

What effect will this have?

```
interpret("if true {}; else {};")(Map.empty)
// res5: Either[String, Map[String, Int]] = Left((line 1, column 11):
//   unexpected ";"
//   expected else
//   semi-colons cannot be written after ifs
//   >if true {}; else {};
//              ^)
```

Ok, this is better! But what if I wrote something else there instead?

```
interpret("if true {}a else {};")(Map.empty)
// res6: Either[String, Map[String, Int]] = Left((line 1, column 11):
//   unexpected "a"
//   expected else
//   semi-colons cannot be written after ifs
//   >if true {}a else {};
//              ^)
```

Ah, right, not so good anymore. We could go back and make the `explain` a bit more general, of course, but that means we've lost out on the helpful prompt to the user about our language's syntax. So, what can we do here? This is where the *Verified Errors* pattern comes into play.

## 2 *Verified Errors*

The problem with using `explain` to write about a specific instance of a problem is that there is no guarantee that the problem actually occurred. This is what we observed above, where an `a` was misreported as a semi-colon! The *Verified Errors* pattern tells us the following:

> *Parse bad input to verify that contextual obligations are met before raising an error.*

Basically, if we want to report an error about a spurious semi-colon, we better check that it is *actually* there first! There are a **few properties** we'd expect of these errors and there are a few ways of formulating them. One way is using `amend`, `hide`, `empty`, and so on, but most of the time this pattern follows a regular enough structure that it has been encoded into a family of combinators in `parsley.errors.patterns.VerifiedErrors`:

```
import parsley.character.char
import parsley.errors.patterns.VerifiedErrors

val _semiCheck =
    char(';').verifiedExplain("semi-colons cannot be written between `if` and
 `else`")

lazy val ifStmt: Parsley[Eval[Unit]] =
    ( "if" ~> pred
    , braces(stmts)
    , ("else" | _semiCheck) ~> braces(stmts)
    ).zipped(cond[Unit])
```

Here, if we can't read an `else`, we immediately try to parse `_semiCheck`, which reads a semi-colon (and is careful to hide it from the errors, otherwise we might see `expected else or ";"`). If it succeeds, then we fail generate a reason for the failure. If it didn't successfully parse, nothing happens, because the *contextual obligation* for the error was not met. A parser like `_semiCheck` is known as an "error widget": prefix them with _ to make them more immediately identifiable.

With this we have:

```
interpret("if true {}; else {};")(Map.empty)
// res7: Either[String, Map[String, Int]] = Left((line 1, column 11):
//   unexpected ";"
//   expected else
//   semi-colons cannot be written between `if` and `else`
//   >if true {}; else {};
//              ^)

interpret("if true {}a else {};")(Map.empty)
// res8: Either[String, Map[String, Int]] = Left((line 1, column 11):
//   unexpected "a"
//   expected else
//   >if true {}a else {};
//              ^)
```

This is exactly what we wanted! So, where else can we apply this technique? Let's see what happens if we miss out a closing brace:

```
interpret("if true {} else {")(Map.empty)
// res9: Either[String, Map[String, Int]] = Left((line 1, column 18):
//   unexpected end of input
//   expected "}", identifier, or if
//   >if true {} else {
//                    ^)
```

We could, again, give the user a helping hand here, and point out that they have an unclosed *something* that they need to close. Again, we could start by using the `.explain` combinator directly, on the `}`. Let's see what effect this will have:

```
def braces[A](p: =>Parsley[A]) = "{" ~> p <~ "}".explain("unclosed `if` or
 `else`")
```

This will give us the more helpful error:

```
interpret("if true {} else {")(Map.empty)
// res10: Either[String, Map[String, Int]] = Left((line 1, column 18):
//   unexpected end of input
//   expected "}", identifier, or if
//   unclosed `if` or `else`
//   >if true {} else {
//                    ^)
```

This time, adding extra input won't cause a problem, so is this fine? Well, what about this input:

```
interpret(
    """if true {}
      |else {
      | x = 7a
      |}""".stripMargin)(Map.empty)
// res11: Either[String, Map[String, Int]] = Left((line 3, column 7):
//   unexpected "a"
//   expected ";", "}", *, +, -, or digit
//   unclosed `if` or `else`
//   >else {
//   > x = 7a
//         ^
//   >})
```

Argh! The `else` *is* closed this time, but since `}` is a valid continuation character we've triggered our `explain` message. Again, we can fix this by using a verified error (on `eof`)

```
import parsley.Parsley.eof
val _eofCheck = eof.verifiedExplain("unclosed `if` or `else`")
def braces[A](p: =>Parsley[A]) = "{" ~> p <~ ("}" | _eofCheck)
```

This time we've latched onto whether or not there is any input left at all. This will work fine!

```
interpret(
    """if true {}
      |else {
      | x = 7a
      |}""".stripMargin)(Map.empty)
```

```
// res12: Either[String, Map[String, Int]] = Left((line 3, column 7):
//   unexpected "a"
//   expected "}"
//   >else {
//   > x = 7a
//         ^
//   >})

interpret(
    """if true {}
      |else {
      | x = 7
      |""".stripMargin)(Map.empty)
// res13: Either[String, Map[String, Int]] = Left((line 4, column 1):
//   unexpected end of input
//   expected ";", "}", *, +, or -
//   unclosed `if` or `else`
//   > x = 7
//   >
//    ^)
```

Perfect #. What now? Well, another area where the user might trip up is thinking that you can assign booleans to variables! Let's see what the errors are:

```
interpret("x = true")(Map.empty)
// res14: Either[String, Map[String, Int]] = Left((line 1, column 5):
//   unexpected keyword true
//   expected "(", identifier, negate, or number
//   >x = true
//        ^^^^)

interpret("x = 10 < 9")(Map.empty)
// res15: Either[String, Map[String, Int]] = Left((line 1, column 8):
//   unexpected "<"
//   expected ";", *, +, -, or end of input
//   >x = 10 < 9
//           ^)

interpret("x = not true")(Map.empty)
// res16: Either[String, Map[String, Int]] = Left((line 1, column 5):
//   unexpected keyword not
//   expected "(", identifier, negate, or number
//   >x = not true
//        ^^^)
```

Now, there is a cheap way of dealing with this and an expensive one. Let's start cheap and see what needs to be done and how effective it is. The first thing we can recognise is that we can special case `not`, `true`, and `false` using the same strategy as before. We can choose to attach a new widget to the `expr` inside the `asgn`, let's start by defining the widget (using some more generalised machinery):

```
import parsley.combinator.choice
val _boolCheck = choice(
        lexer.nonLexemeSymbol("true"),
        lexer.nonLexemeSymbol("false"),
        lexer.nonLexemeSymbol("not"),
    ).verifiedExplain("booleans cannot be assigned to variables")
```

The `lexer.nonLexemeSymbol` combinator here is allowing for the parsing of keywords *without* reading trailing whitespace, which would make our error messages wider. Now, we can add this to the `asgn`:

```
lazy val asgnStmt: Parsley[Eval[Unit]] =
    (identifier, "=" ~> (expr | _boolCheck)).zipped(store)
```

And now for the errors:

```
interpret("x = true")(Map.empty)
// res17: Either[String, Map[String, Int]] = Left((line 1, column 5):
//   unexpected keyword true
//   expected "(", identifier, negate, or number
//   booleans cannot be assigned to variables
//   >x = true
//        ^^^^)

interpret("x = 10 < 9")(Map.empty)
// res18: Either[String, Map[String, Int]] = Left((line 1, column 8):
//   unexpected "<"
//   expected ";", *, +, -, or end of input
//   >x = 10 < 9
//           ^)

interpret("x = not true")(Map.empty)
// res19: Either[String, Map[String, Int]] = Left((line 1, column 5):
//   unexpected keyword not
//   expected "(", identifier, negate, or number
//   booleans cannot be assigned to variables
//   >x = not true
//        ^^^)
```

So, we've cracked the *leading* edge of the booleans, but we are still no closer to managing to deal with `<`, `==`. These are significantly trickier to handle with our current approach, because they occur *after* some input has already been read. We would need to insert them as alternatives at every point where they could be "valid" predictions. This is far from ideal. Note that we don't really need to worry about `&&` and `||`, since we are going to have to have already found one of `<`, `==`, `not`, `true`, or `false` before we reach it *anyway*. Instead, we need to look towards a different technique, the *Preventative Error*.

## 3 *Preventative Errors*

The problem with *verified errors* is that they have to follow the valid alternatives of a parse, which may be disparate and scattered. Not to mention that moving the widget to a shared parser may actually invalidate its context anyway! Instead, a *preventative error* allows us to proactively rule out bad input. This means trying to parse it *before* trying the "working" alternatives. Compared with *verified errors*, which are a last resort, *preventative errors* are by nature more costly. Again, there are a **few properties** we'd expect of these errors and there are a few ways of formulating them and `parsley.errors.patterns.PreventativeErrors` contains a family of combinators for the most common formulation:

```
import parsley.errors.patterns.PreventativeErrors

val _noCompCheck =

  choice(lexer.nonLexemeSymbol("<"), lexer.nonLexemeSymbol("==")).preventativeExplain(
        reason = "booleans cannot be assigned to variables",
        labels = "end of arithmetic expression"
    )
```

The idea here is to try and parse one of < or == and fail immediately if this is possible, otherwise, the combinator succeeds, and we can start parsing something else. Because raising the error happens more eagerly, it is important to provide error labels to describe what valid things can come next since these parsers won't actually get executed and contribute themselves! It should be placed at a point where it is suspected the < and == could be read, namely after an expression in the `asgn`. So, let's see what the effect of the error will be:

```
lazy val asgnStmt: Parsley[Eval[Unit]] =
    (identifier, "=" ~> ((expr <~ _noCompCheck) | _boolCheck)).zipped(store)
```

```
interpret("x = 10 < 9")(Map.empty)
// res20: Either[String, Map[String, Int]] = Left((line 1, column 8):
//   unexpected "<"
//   expected end of arithmetic expression
//   booleans cannot be assigned to variables
//   >x = 10 < 9
//           ^)
```

This looks pretty good! However, it's now looking a little messier, and could do with some abstraction:

```
def _noBool[A](p: Parsley[A]): Parsley[A] = (p <~ _noCompCheck) | _boolCheck

lazy val asgnStmt: Parsley[Eval[Unit]] = (identifier, "="
 ~> _noBool(expr)).zipped(store)
```

This is a bit tidier! When I said that there was a cheap way and an expensive way of implementing these improved errors earlier, the expensive way would have been using a preventative error from the outset, trying to parse an *entire* boolean expression up-front. This may be much more expensive than the reasonably small <7 character checks we've been doing. However, it is instructive to see what this might have looked like, now just by changing `_noBool`:

```
import parsley.errors.VanillaGen

val _noBoolCheck = pred.void.preventWith(
    err = new VanillaGen[Unit] {
        override def reason(x: Unit) = Some("booleans cannot be assigned to
 variables")
        override def unexpected(x: Unit) = VanillaGen.NamedItem("boolean
 expression")
    },
    labels = "arithmetic expression"
)


def _noBool[A](p: Parsley[A]): Parsley[A] = _noBoolCheck ~> p
```

This version makes use of the `VanillaGen`, which can allow for a bit more fine-grained configuration, including changing the unexpected message within the error. By parsing a full `pred`, we are certain to rule out all problematic inputs immediately, but this could be much more costly - there is a trade-off to be had! The errors are all as follows:

```
interpret("x = true")(Map.empty)
// res21: Either[String, Map[String, Int]] = Left((line 1, column 5):
//   unexpected boolean expression
//   expected arithmetic expression
//   booleans cannot be assigned to variables
//   >x = true
//        ^^^^)

interpret("x = 10 < 9")(Map.empty)
// res22: Either[String, Map[String, Int]] = Left((line 1, column 5):
//   unexpected boolean expression
//   expected arithmetic expression
//   booleans cannot be assigned to variables
//   >x = 10 < 9
//        ^^^^^^)

interpret("x = not true")(Map.empty)
// res23: Either[String, Map[String, Int]] = Left((line 1, column 5):
//   unexpected boolean expression
//   expected arithmetic expression
//   booleans cannot be assigned to variables
//   >x = not true
//        ^^^^^^^^)
```

The flip side is that these errors are just about as good as it gets for this particular problem: notice that the error caret covers the *entire* bad expression, and reports it under a concise naming. It can also catch errors with bracketed expressions, which may be missed by the previous implementations (though it is still possible to catch these with some work).

## 4 Conclusion

The main point of this page is to demonstrate how much of an important technique "parsing bad inputs" can be to generating informative and precise error messages. Of course, there are other formulations of these patterns, and they can address some of the shortcomings of the above implementations. In future, I may add a discussion of *how* these patterns are implemented in practice, to show how more bespoke ones can be useful.