Misc. things to write (might move to other sections): * Top-down parsers – intro to left-recursion and the types of left recursion in PEG grammars

Not sure how to order things: lint rules first, or implementation?

Chapter 1

Lint Rules in parsley-garnish

Catalogue of lint rules implemented.

Categorise these – but also somehow split into the ”simple” rules and the ”complex” rules. Simple rules can consist of a single heading, containing: * Explanation of the rule * Simple example to show a diagnostic, and a before and after if it’s fixable * How it’s implemented in the code * Proof (if applicable) * Limitations

Simple rule ideas: * Overly complex parser definitions * Manually calling implicitSymbol instead of using the implicit

Not sure how to lay out the complex rules yet – so far this is just the left-recursion removal rule. The other complex rule(s) will likely share implementation details with the Parser/Func representation, so work from there.

1.1 Avoid Redefining Existing Parsers

* Catch cases when user manually writes out a parser that is already defined in the library

1.2 Simplify Complex Parsers

* Apply parser laws, re-using Parser and Func representations to do cool things <- should this be a separate rule?

1.3 Ambiguous Implicit Conversions

> Heroin is just one letter away from heroine, and implicit conversions are the heroine we don’t deserve.
>
> Jamie Willis, 2024

Problem

Implicit conversions are a powerful feature in Scala, allowing users to supply an argument of one type when another is expected, to reduce boilerplate. As noted by Willis and Wu [2022], implicit conversions are particularly useful for designing DSLs. In the context of parser combinators, they introduce the usage of implicit conversions to automatically lift string and character literals into parsers in the *Implicit Conversions* design pattern. This eliminates the need to explicitly wrap these elements in combinators: `string("parsley") | string("garnish")` can now be expressed as just `"parsley" | "garnish"`, more closely resembling the style of a BNF grammar.

The *Implicit Lexer* pattern is a further specialisation of this approach, hiding the boilerplate of whitespace handling entirely within a `lexer` object. This design pattern allows whitespace handling to be encapsulated as private combinators within the `lexer` object, which are then made available only through implicit conversions automatically applied by the Scala compiler.

However, due to their utility, implicit conversions are also an easily abused feature of Scala. They can obscure the flow of the program, making it difficult to understand what the code is doing and potentially hiding side effects or costly operations. A downside particularly relevant to Parsley is that implicit conversions often lead to confusing error diagnostics when the compiler is unable to resolve them.

One common issue arises from ambiguous implicits when there are multiple implicit conversions in scope. Parsley provides `stringLift` and `charLift` combinators in the `parsley.syntax.character` package for the *Implicit Conversions* pattern, and exposes an `implicitSymbol` combinator for lexers to use in the *Implicit Lexer* pattern. For novice users, it is easy to accidentally import both sets of these implicits, when it is likely that they only intended to use the `implicitSymbol` implicit. For example, consider the following code snippet:

```
val p = 'g' ~> "arnish"
p.parse("garnish")
// [error] type mismatch;
//   found   : String("arnish")
//   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("arnish") to parsley.Parsley[?]
//    val p = 'g' ~> "arnish"
//                   ^^^^^^^^
```

Here, the compiler provides a detailed error message indicating the ambiguity between two possible implicit conversions. However, the compiler is not always able to report such issues clearly. For instance, switching the position of the intended implicit conversion results in a less helpful message:

```
val p = "garnis" <~ 'h'
p.parse("garnish")
// [error] value <~ is not a member of String
//    val p = "garnis" <~ 'h'
//            ^^^^^^^^^^^
```

Solution

Ideally, this issue would be addressed by implementing a lint-on-compile rule, which could annotate the compiler error message at the exact location of the issue. If this were implemented as a compiler plugin, partial information available from the compiler stages before the error could potentially provide enough detail to identify the exact clashing implicits. This approach would allow leveraging domain knowledge to update the error message with more useful Parsley-specific diagnostics.

Incidentally, WartRemover has a related lint rule for implicit conversions[1], although it only targets the locations where implicit conversions are *defined*, not where they are *applied*. Despite this limitation, it serves as a proof of concept demonstrating the feasibility of such an approach.

Unfortunately, Scalafix restricts usage to only syntactic rules on the bare AST or semantic rules that operate fully post-compilation. Since the ambiguous implicit conversions will cause compilation failures, this lint must be

[1] http://www.wartremover.org/doc/warts.html#implicitconversion

implemented as a syntactic rule. Consequently, the solution takes a different approach: estimating the presence of clashing implicits by examining their import statements within each scope.

Example

Fig. 1.1 extends the previous example to a full Scala source file following the *Implicit Lexer* pattern, but where the user has erroneously additionally imported the `stringLift` implicit from the *Implicit Conversions* pattern. This results in the Scala compiler throwing an error on line 6 due to ambiguous implicits. When run on this file, `parsley-garnish` will report a warning at line 3 similar to that shown in fig. 1.2.

```
1  object parser {
2    import parsley.syntax.character.stringLift
3    import lexer.implicits._
4
5    val p = "garnis" <~ 'h'
6  }
7
8  object lexer {
9    import parsley.token.Lexer, parsley.token.descriptions.LexicalDesc
10
11   private val lexer = new Lexer(LexicalDesc.plain)
12   val implicits = lexer.lexeme.symbol.implicits
13 }
```

Fig. 1.1: A minimal Parsley program which fails to compile due to ambiguous implicits in the `parser` object.

```
warning: [AmbiguousImplicitConversions] This import may cause clashing implicit conversions:
* import parsley.syntax.character.stringLift at line 2
* import lexer.implicits._ at line 3
If this is the case, you may encounter confusing errors like 'method is not a member of String'.
To fix this, ensure that you only import a single implicit conversion.

  import lexer.implicits._
  ^^^^^^^^^^^^^^^^^^^^^^^^
```

Fig. 1.2: The warning message produced by the `AmbiguousImplicitConversions` lint rule.

Implementation

Unlike Java, Scala offers more flexibility with import statements, allowing them to appear anywhere in source files rather than just at the top. Scala's import statements are lexically scoped, allowing their visibility to be limited to a single class, object, or function. Additionally, Scala processes import statements in a top-down order within the file, further restricting their visibility, as scopes above an import cannot see the imports defined below them.

* Scalafix (via scalameta) provides a generic traversal of the AST: filter to find all import statements in top-down order * This allows the scope to be lexically managed – traversal in the same order that the compiler reads imports * The ancestor AST node of an import statement is its enclosing scope * Use ancestor information to determine which of the visited imports are in scope at that point

* to find stringLift: Pattern match to find if import is of form ‘import parsley.syntax.character._‘ * to find implicit lexer: pattern match to find if there is an importee called ‘implicitSymbol‘ or if an import contains keywords ‘lexer‘ and ‘implicit(s)‘

* if at any point in the traversal, both types of imports are in scope, report a warning

1.4 Remove Explicit Usage of Implicit Conversions

1.5 Refactor to use Parser Bridges

* This would be cool, idk if I have time though, but this should also piggyback off of Func * the pos bridges don’t actually exist, so we can ignore that case and just say its too much code synthesis * shouldn’t be too bad? idk * indicate limitations that this will only work if the ADT is defined in the same file, in order to extend it

1.6 Left Recursion Factoring

Chapter 2

Implementation

Non-terminal detection. This may get reworked/renamed since it's pretty specialised for leftrec rn, and in reality it's just trying to grab all the parsers.

Other util things? ACTUALLY NEED TO DO: import combinators if they aren't already imported

2.1 Parser Representation

Several of the more complex lint rules, most notably §1.6, require manipulating parser combinators in a high-level manner. It would be possible, but extremely cumbersome, to work directly with the generic Scala AST nodes provided by Scalameta. Instead, it is very useful to represent parsers as an algebraic data type ADT to simplify the manipulation of combinators. This section explores the motivation behind this and the design choices made in the implementation. Use the §1.6 rule as a basis/context to demonstrate the utility of this representation.

Why do we want a parser representation? Because the left-recursion factoring is based on lawful transformations and rearrangements of parsers, which are much easier to achieve and reason about when parsers are represented as a high-level ADT rather than as raw Scala AST nodes. This representation also then gives us for free the implementation for lint rules such as *Simplify Complex Parsers* rule, which applies parser laws to simplify parser definitions.

Running example to motivate all requirements for the parser representation – removing left recursion from the following simple parser:

```
lazy val p: Parsley[String] = (p, string("a")).zipped(_ + _) | string("b")
```

2.1.1 Detecting Parsers

Finding AST nodes of interest which actually represent a Parser. Find val/var/def definitions which have a type that is inferred to have type `Parsley[_]`.

Take our example:

```
lazy val p: Parsley[String] = (p, string("a")).zipped(_ + _) | string("b")
^^^^^^^^ ^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
val      sym p: Parsley[_]    rhs of definition: convert this to Parser
```

Convert rhs of definition to Parser. In each document, build a map of all parsers defined in that document, indexed by the symbol of the definition. This gets us all the non-terminals in the grammar defined within that file.

2.1.2 Converting Scalameta Trees to the Parser ADT

Pattern match on AST nodes of interest to build the appropriate Parser. Each Parser defines a partial function `fromTerm` which creates an instance of that parser from the appropriate Scalameta `Term`. Use Scalafix's `SymbolMatcher` to match tree nodes that resolve to a specific set of symbols. This makes use of semantic information from SemanticDB, so we are sure that a `<*>` is actually within the `parsley.Parsley` package, rather than

some other function with the same name. This is a drawback of HLint, which is prone to false positives due to its reliance on syntactic information only.

For example:

```
object Ap {
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

  def fromTerm(implicit doc: SemanticDocument): PartialFunction[Term, Ap] = {
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
      Ap(p.toParser, q.toParser)
  }
}
```

So this looks for an infix application `p <*> q`, where we can then build an `Ap` parser while recursively converting `p` and `q` to parsers.

Our running example would then be converted as so (cleaned up to be less verbose). For now we are only concerned with the parser representation; we will touch upon how `Function`s are represented in the next section.

```
// (p, string("a")).zipped(_ + _) | string("b")
Choice(
  Zipped(Function(_ + _), List(NonTerminal(p), Str(a))),
  Str(b)
)
```

2.1.3 Building New Parsers From Existing Parsers

Now that we have parsers represented as an ADT, we can easily build new parsers from existing parsers. This is crucial for the left-recursion factoring rule, which "unfolds" parsers into separate parsers representing the left-recursive and non-left-recursive parts. These are then recombined to form parsers which are free from left recursion.

Make this even easier by utilising Scala's ability to define infix operators, define them as extension methods on the `Parser` trait. For example:

```
implicit class ParserOps(private val p: Parser) extends AnyVal {
  def <*>(q: Parser): Parser = Ap(p, q)
  def <|>(q: Parser): Parser = Choice(p, q)
  def map(f: Function): Parser = FMap(p, f)
}
```

This makes it more ergonomic to manipulate parsers, it's like we're writing Parsley code itself. A small example snippet from the `unfold` method on the `Ap` parser:

```
val lefts = {
  val llr = pl.map(flip) <*> q
  val rlr = pe.map(f => ql.map(composeH(f))).getOrElse(Empty)
  llr <|> rlr
}
```

Notice how the code closely resembles the high-level description of the transformation, using `<*>`, `<|>`, `map`, operators.

2.1.4 Simplifying Parsers Using Parser Laws

Once all the unfolded parsers have been recombined, the raw output is very noisy and difficult to read. Again, ignore the functions, these will be covered in §2.2.

```
lazy val p: Parsley[String] = chain.postfix(
  empty | (empty.map(a => b => a + b) | empty <*> p) <*> string("a") | string("b") | empty
)(
  (empty.map(FLIP) <*> p | pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
    | empty | empty
)
```

For human readability of the transformed output, it is therefore important to simplify the parser as much as possible. This is another key motivation for the parser representation, bringing static inspectability. Similar to the high-level optimisations done in [Willis, Wu, and Pickering 2020] using parser laws based on applicative, alternative, selective. Willis [2023] provides the most comprehensive list of parser laws which Parsley adheres to.

```
p.map(f).map(g) = p.map(g compose f)                    (2.1)
pure(f) <*> pure(x) = pure(f(x))                        (2.2)
pure(f) <*> x = x.map(f)                                (2.3)
empty | u = u                                           (2.4)
u | empty = u                                           (2.5)
pure(x) | u = pure(x)                                   (2.6)
empty <*> u = empty                                     (2.7)
empty.map(f) = empty                                    (2.8)
```

Fig. 2.1: Functor (2.1), Applicative (2.2, 2.3), and Alternative (2.4–2.8) laws.

Example by hand. In the example we can see that the most noise results from the `empty` combinators. Tidying those up first:

```
lazy val p: Parsley[String] = chain.postfix(
  empty | empty | string("b") | empty
)(
  (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
    | empty | empty
)
```

`empty | empty | string("b") | empty` simplifies to just `string("b")` via eqs. (2.4) and (2.5).

```
    (pure(ID).map(COMPOSE(a => b => a + b))).map(FLIP) <*> string("a")
=       { eqs. (2.2) and (2.3) }
    pure(COMPOSE(a => b => a + b)(ID)).map(FLIP) <*> string("a")
=       { eqs. (2.2) and (2.3) }
    pure(FLIP(COMPOSE(a => b => a + b)(ID))) <*> string("a")
=       { eq. (2.3) }
    string("a").map(FLIP(COMPOSE(a => b => a + b)(ID)))
```

So final result:

```
val f: Function = FLIP(COMPOSE(a => b => a + b)(ID))
lazy val p: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

Implementation

Bottom-up traversal to apply peephole optimisations based on parser laws. For maintainability, code readability, and extensibility: decouple the application of transformation function from recursive traversal through the datatype. Hand-written traversal, but inspired by the generic traversal techniques in [Mitchell and Runciman 2007]. Make ADT sealed so the compiler will warn if we miss on adding a new case: this is still more error prone than generically deriving, but we don't want to introduce a new dependency on Shapeless library (it's not complex enough to warrant bringing in a dependency for generic derivation).

Representation of Parsley combinators in parsley-garnish. Compare with approach in Scala Parsley, take cues from the 2018 paper. * Approach to composites? Need to think about this. * For LeftRec: Parse ASTs into a small group of core combinators, but we also need to represent composite combinators as their own case classes – recombine/"simplify" after analysis is concluded, it doesn't really matter if we completely change what combinators are used as long as semantic meaning is preserved. * For others: probably need to parse directly into composite combinators, since we don't want to destructively modify what combinators have been used. * Optimisations: for us, the goal is human readability, so this is interesting to compare to the paper. Lots of similar stuff actually, like top-down peephole optimisations utilising parser laws (I think I do it this way? Need to double check). * For cleanliness to isolate boilerplate: https://blog.sumtypeofway.com/posts/introduction-to-recursion-schemes.html – we don't have a generic traversal, but we can decouple the recursive application of a given partial function from the actual pf itself (I've called it .transform for the Parser class)

2.2 Function Representation

From the running example it is hopefully becoming clear that functions also need to be simplified, for the same reasons that parsers need to be simplified.

Abstraction built over scalafix/meta ASTs to represent functions. Allows us to statically evaluate function composition/flipping etc, so it doesn't turn into one big mess – again, human readability of the transformed output is the goal. Abstraction is again an ADT as a lambda calculus, but with parameter lists so not everything is curried. îdk, this is still a work-in-progress. Seems that there might not be enough time to uncurry the leftrec analysis so this design decision might not be super important. Representation as a lambda calc has allocation overhead, but greatly simplifies function evaluation via beta reduction, instead of having to deal with high-level representations of compose/id (not too bad tbh) and flip (annoying). Also attempted to make it typed but that didn't go so well with Scala's limitations on type inference.

* Extracting method arguments (alongside their types) is very painful * Need to unify information from signature (within symbolinformation) and synthetics * synthetics exist in certain cases: .apply methods, showing the concrete type of a generic argument, implicit conversions * from https://scalacenter.github.io/scalafix/docs/developers/semantic-tree.html: SemanticTree is a sealed data structure that encodes tree nodes that are generated by the compiler from inferred type parameters, implicit arguments, implicit conversions, inferred .apply and for-comprehensions.

* map, lift (implicit and explicit), zipped, (.as perhaps?) – these should surely boil down into two cases: (x, y).xxx(f) and xxx(f, x, y) * named function literals (val) * named method literals (def) * anonymous functions i.e. lambdas * functions with placeholder syntax * apply methods of case classes - symbol will tell its a class signature so we use this as a clue to look at synthetics??? * generic bridges – I reckon the information will probably show up in synthetics again

* Don't have full access to type information - can do more work here theoretically, but its difficult and error-prone * So we don't model a typed lambda calculus, just have it untyped

Approaches - AVOIDING capture via substitution * Substitution approaches * De Bruijn indices - inefficient to open/close terms so much - De Bruijn levels as an alternative * HOAS * Normalisation by evaluation