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

Chapter 1
Design and Implementation

Writing domain-specific lint rules unlocks the potential for more powerful and interesting transformations utilising specialised domain knowledge. Desirable: * inspectability for analysis (that's what we're here for!) and optimisation The purpose of this chapter is to describe the intermediate representations of parsers (§1.1) and functions (§1.2). Show that terms must be simplified to a normal form Demonstrate equivalence to dsl optimisations in staged metaprogramming

1.1 Parser Representation

This is an INTERMEDIATE SYMBOLIC REPRESENTATION (?) more specialised than general-purpose scala ast This section is about simplifying in our semantic domain (parsers)

Scalafix runs at the meta-level, outside of the phase distinction of compile- and run-time. Staged metaprogramming applies optimisations at compile-time, whereas these "optimisations" at applied post-compilation

For example, given two AST nodes `Term.Name("p")` and `Term.Name("q")` corresponding to named parsers `p` and `q`, suppose a transformation involves combining them with the *ap* combinator `<*>`. One may consider using quasiquotes to achieve this: `q"p <*> q"` would automatically expand to `Term.ApplyInfix(Term.Name("p"), Term.Name("<*>"), Type.ArgClause(Nil), Term.ArgClause(List(Term.Name("q")), None))`. However, this loses the static inspectability of the individual parsers `p` and `q` – although quasiquotes can be used as extractor patterns to recover the original AST nodes, their usage as such is discouraged as they can easily result in unintended match errors. The recommended approach is to pattern match on the AST nodes directly, which is obviously unergonomic even for this small example: to extract the RHS term `q`, one would have to perform a nested pattern match on the `Term.ApplyInfix` term and its `Term.ArgClause` node representing the arguments of the infix function application.

It is hopefully obvious that this would a very painful process for the rule author. It would be desirable to abstract away from the low-level syntactic AST representation, and instead treat these AST nodes as what they semantically represent – parsers.

Instead, fig. 1.1 shows how parser terms can be represented as an algebraic data type ADT, in the same way `parsley` itself uses a deep embedding to represent parsers as pure data objects. The reasoning behind this approach is the same as that for PARSLEY – this representation allows parsers to be easily inspected and analysed via pattern matching on constructors.

```
trait Parser
case class NonTerminal(ref: Symbol) extends Parser
case class Pure(f: Function) extends Parser
case object Empty extends Parser
case class Choice(p: Parser, q: Parser) extends Parser
case class Ap(p: Parser, q: Parser) extends Parser
...
```

Fig. 1.1: A subset of the core combinators in the `Parser` ADT.

This section explores the motivation behind this and the design choices made in the implementation. Use the left-recursion factoring (§2.6) rule as a basis/context to demonstrate the utility of this representation.

Running example The left-recursion factoring rule (§2.6) performs the most complex analyses and transformations on parsers in `parsley-garnish`. Thus, it is a good example to motivate the design requirements for the

parser representation. The following left-recursive parser and its transformation into its `postfix` form will serve as a running example for this section:

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

1.1.1 Detecting Named Parsers

Before any analysis on parsers can be performed, it is first necessary to identify which AST nodes correspond to parsers. `parsley-garnish` builds a map of all parsers defined within a source file, indexed by the unique symbol of its name.

Identifying these AST nodes of interest involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be some `Parsley[_]` – this information is accessed by querying the Scalafix semantic API for the node's symbol information. Consider the labelled `ast` structure of the `expr` parser:

```
Defn.Val(
  mods = List(Mod.Lazy()),
  pats = List(Pat.Var(Term.Name("expr"))),
  decltpe = Some(
    Type.Apply(Type.Name("Parsley"), Type.ArgClause(List(Type.Name("String"))))
  ),
  rhs = Term.ApplyInfix(...)
)
```

The qualified symbol `expr` is used as the key in the map, and the `rhs` term is lifted the intermediate parser representation for analysis. A reference to the original AST node is also kept so any lint diagnostics or code rewrites can be applied to the correct location in the source file. Thus, a full traversal through the source file builds a map of all named parsers, representing all non-terminals in the grammar defined within that file.

1.1.2 Converting Scalameta Terms to the Parser ADT

Having identified the AST nodes which represent parsers, they need to be transformed into the appropriate `Parser` representation. This involves pattern matching on the `scala.meta.Term` to determine which parser combinator it represents, and then constructing the appropriate `Parser` instance.

Each `Parser` defines a partial function, `fromTerm`, which creates an instance of that parser from the appropriate `scala.meta.Term`. These `fromTerm` methods are combined to define a `toParser` extension method on `scala.meta.Term` – this is where AST nodes are lifted to their corresponding `Parser` representation.

The top-level combinator that makes up `expr`'s definition is the choice combinator, `|`. Scalameta represents this infix application of the `|` operator as so:

```
Term.ApplyInfix(
  lhs = Term.Apply(...), // AST node for (expr, string("a")).zipped(_ + _)
  op = Term.Name("|"),
  targClause = Type.ArgClause(List()),
  argClause = Term.ArgClause(
    List(
      Term.Apply(
        Term.Name("string"),
        Term.ArgClause(List(Lit.String("b")), None)
      )
    ),
    None
  )
)
```

This structure therefore guides the implementation of the pattern match in `Choice.fromTerm`:

```
object Choice {
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`|`", "parsley.Parsley.`<|>`")

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

The definition of this method is fairly self-explanatory: it matches on a `ApplyInfix` term where the operator is the `|` combinator, and recursively applies `toParser` to its LHS and RHS nodes. Finishing off, the `expr` parser is therefore converted to the following `Parser` instance:

```
Choice(
  Zipped(Function(_ + _), List(NonTerminal(expr), Str(a))),
  Str(b)
)
```

The exact representation of the `Function` is not important at this momenet – this is covered in the next section. For brevity, the remaining code snippets in this section will simplify the function representations and continue to grey them out.

1.1.3 Building New Parsers From Existing Parsers

Now that raw AST terms can be lifted to the higher-level parser representation, it is easy to build new parsers from existing parsers. This is crucial for left-recursion factoring, 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.

Smart constructors are used to make manipulating parser terms resemble writing `parsley` code itself. These are defined as infix operators, which are written as extension methods on the `Parser` trait:

```
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)
}
```

Parser terms can now be manipulated in a manner that looks almost indistinguishable from writing `parsley` code. For example, the `unfold` method on the `Ap` parser contains this snippet, where `pl`, `ql`, and `q` are parsers (`pe` is not a parser, but rather an `Option` value):

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

Other than the capitalised `Empty` constructor, this would be perfectly valid `parsley` code.

1.1.4 Simplifying Parsers Using Parser Laws

Recombining unfolded parsers during left-recursion factoring introduces many necessary, but extraneous "glue" combinators. Even though the transformed parser is semantically correct, it ends up very noisy syntactically. Consider the resulting parser from factoring out the left-recursion in `expr`:

```
lazy val expr: Parsley[String] = chain.postfix(
  empty | (empty.map(a => b => a + b) | empty <*> expr) <*> string("a")
    | string("b") | empty
)(
  (empty.map(Flip) <*> expr | pure(Identity).map(Compose(a => b => a + b)))
    .map(Flip) <*> string("a")
    | empty | empty
)
```

The intent of this parser is completely obfuscated – it would be unacceptable for the output of the transformation to be left in this form. For human readability, this parser term must be simplified as much as possible, using domain-specific knowledge about parser combinators. This is where the deep embedding approach comes to shine; simplifications are easily expressed by pattern matching on `Parser` constructors.

Willis, Wu, and Pickering [2020] note that parser combinators are subject to *parser laws*, which often form a natural simplification in one direction. In Haskell `parsley`, Willis [2023] uses these parser laws as the basis for high-level optimisations to simplify the structure of the combinator tree. `parsley-garnish` uses the same principles to simplify the parser term to become more human-readable. The two only differ in the purpose of the simplification: whereas Haskell `parsley` does this to produce an optimised AST to be compiled as code, `parsley-garnish` simplifies the parser AST to be pretty-printed as text.

Fig. 1.2 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of the laws in fig. 1.2 have already been shown to hold for Parsley by Willis and Wu [2018]; an additional proof for eq. (1.8) can be found in **??**.

$$\texttt{p.map(f).map(g)} = \texttt{p.map(g compose f)} \tag{1.1}$$
$$\texttt{pure(f) <*> pure(x)} = \texttt{pure(f(x))} \tag{1.2}$$
$$\texttt{pure(f) <*> x} = \texttt{x.map(f)} \tag{1.3}$$
$$\texttt{empty | u} = \texttt{u} \tag{1.4}$$
$$\texttt{u | empty} = \texttt{u} \tag{1.5}$$
$$\texttt{pure(x) | u} = \texttt{pure(x)} \tag{1.6}$$
$$\texttt{empty <*> u} = \texttt{empty} \tag{1.7}$$
$$\texttt{empty.map(f)} = \texttt{empty} \tag{1.8}$$

Fig. 1.2: Functor (1.1), Applicative (1.2, 1.3), and Alternative (1.4–1.8) laws.

In the previous example, it is evident that the most noise results from the `empty` combinators. These can be eliminated using eqs. (1.4), (1.5), (1.7), and (1.8):

```
lazy val expr: Parsley[String] = chain.postfix(string("b"))(
  (pure(Identity).map(Compose(a => b => a + b))).map(Flip) <*> string("a")
)
```

The complicated term in the postfix operator can then be simplified as follows:

```
    (pure(Identity).map(Compose(a => b => a + b))).map(Flip) <*> string("a")
=       { eqs. (1.2) and (1.3) }
    pure(Compose(a => b => a + b)(Identity)).map(Flip) <*> string("a")
=       { eqs. (1.2) and (1.3) }
    pure(Flip(Compose(a => b => a + b)(Identity))) <*> string("a")
```

$=$ { eq. (1.3) }

```
string("a").map(Flip(Compose(a => b => a + b)(Identity)))
```

This results in the most simplified form of the parser:

```
val f: Function = Flip(Compose(a => b => a + b)(Identity))
lazy val expr: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

Encapsulating boilerplate Lawful simplifications are applied akin to peephole optimisations on the recursively defined `Parser` ADT. There are many instances of parsers, which inevitably leads to repetitive and error-prone boilerplate code which exists to simply recurse through each case. To avoid this, the recursive traversal itself is decoupled from the application of the transformation function. Although the traversal is still hand-written, the implementation is inspired by the generic traversal patterns offered by Haskell's `uniplate` library [Mitchell and Runciman 2007].

This is realised as a `transform` method on the `Parser` trait, which takes a partial function and applies it to nodes where it is defined. The transformation is applied via a bottom-up traversal:

```
def transform(pf: PartialFunction[Parser, Parser]): Parser = {
  val p = this match {
    case Ap(p, q)      => Ap(p.transform(pf), q.transform(pf))
    case Zipped(f, ps) => Zipped(f, ps.map(_.transform(pf)))
    case Pure(f)       => Pure(f)
    ...
  }
  if (pf.isDefinedAt(p)) pf(p) else p
}
```

A `rewrite` method can then be defined in terms of `transform`, applying the partial function everywhere and re-applying it until it no longer makes a change. This has the effect of applying a transformation exhaustively until a normal form is reached.

```
def rewrite(pf: PartialFunction[Parser, Parser]): Parser = {
  def pf0(p: Parser) = if (pf.isDefinedAt(p)) pf(p).rewrite(pf) else p
  this.transform(pf0)
}
```

Therefore, any transformation on parsers can be defined without having to worry about recursion boilerplate: the act of traversal itself is fully abstracted away and encapsulated within the `transform` method. Using `rewrite`, parser simplification can then be expressed in a clean and maintainable manner:

```
def simplify: Parser = this.rewrite {
  // p.map(f).map(g) == p.map(g compose f)
  case FMap(FMap(p, f), g) => FMap(p, composeH(g, f))
  // u <|> empty == u
  case Choice(u, Empty) => u
  // pure(f) <|> u == pure(f)
  case Choice(Pure(f), _) => Pure(f)
  ...
}
```

Further design considerations are made to ensure the extensibility and safety of this approach: the `Parser` trait is sealed, which enables compiler warnings if a new `Parser` case is added and the `transform` method is not updated. Since the traversal is still written by hand rather than generically derived, it is still more prone to error The

traversal could be generically derived rather than written by hand, but this would require the use of an external dependency such as `shapeless`[1], which is overkill for the complexity of the `Parser` ADT.

1.1.5 Converting Parsers Back to Scalameta Terms

After parsers have been transformed and simplified, the last step is to convert them back to a textual representation to be applied as a Scalafix patch. Parsers can be lowered back to `scala.meta.Term` nodes by the inverse of the original `fromTerm` transformation. The `Parser` trait defines this transformation as the method `term`, using quasiquotes to simplify the construction of the `scala.meta.Term` nodes.

```
case class Zipped(func: Function, parsers: List[Parser]) extends Parser {
  val term: Term = q"(..${parsers.map(_.term)}).zipped(${func.term})"
}
```

This term can then be pretty-printed into a string, and applied as a Scalafix patch.

Summary

1.2 Function Representation

This section is about simplifying in the general domain, so really Squid can do all of this? Still an interesting approach ig And shows a shortcoming of scalameta quasiquotes

Section 1.1 showed that it is useful to lift AST nodes corresponding to parser terms into an intermediate `Parser` representation. Crucially, §1.1.4 demonstrates how this representation lends itself to reducing parsers to a simpler form. This section extends this idea to representing functions in a similar, statically-inspectable manner, and explores why this machinery is not just desirable but necessary for `parsley-garnish`.

Running example Utilising parser laws as a basis for term simplification for the `Parser` ADT has managed to tame the jumble of parser combinators resulting from the left-recursing factoring transformation §2.6. However, the functions passed to lifting combinators such as `map` have so far been treated as opaque, and have not been subject to the same simplifications as parsers. We continue from the running example from the previous section, where the function passed to the `map` combinator has ended up in an unnecessarily complex form:

```
val f: Function = Flip(Compose(a => b => a + b)(Identity))
```

This is because the transformations on parsers during left-recursion factoring involve mapping over parsers with higher-order functions such as `flip` to reverse the order of arguments, and `compose` to compose functions. These are used to ensure that unfolded parsers are recombined in a lawful manner, but also result in a lot of syntactic noise. Yet again, leaving the transformed terms in this form would be unacceptable for the linter. It is vital that these functions are partially evaluated and *normalised* into a semantically equivalent but visually simpler form.

1.2.1 Representing Functions

Partial evaluation and normalisation are related concepts: it is useful to view normalisation as statically evaluating as many terms as possible, but since not all terms have known values, the expression cannot be fully evaluated to a result value. Normalisation can thus be viewed simply as a process of evaluation, but in the presence of unknown terms.

The functions passed to parsers such as `map` are generally relatively simple and assumed to be pure, so these are modelled as a variant of the lambda calculus. In the lambda calculus, each function only takes one argument, and multi-argument functions are represented as a chain of single-argument functions: this is known as *currying*. In Scala, functions are allowed to have multiple parameter lists to make them curried, although uncurried functions

[1] https://github.com/milessabin/shapeless

are preferred for performance reasons. In order to maintain a high-level equivalence between this representation and Scala code, the lambda calculus is extended to be able to represent multi-argument functions using n-ary abstraction and application.

$$M, N ::= x \quad \text{(variable)} \mid (\lambda x.M) \quad \text{(abstraction)} \mid (MN) \quad \text{(application)}$$

Assume functions passed to parsers are pure, because we assume sane users. So it is reasonable to model functions as a lambda calculus, and this can also be achieved with an ADT. In the simple lambda calculus all functions are curried i.e. each function only takes one argument, and multi-argument functions are represented as a chain of single-argument functions. In Scala, uncurried functions are preferred for performance reasons, but it allows currying as functions can have multiple parameter lists. We could desugar all this and curry the functions, but we want to be able to easily transform our output back into Scala code. So we extend the lambda calculus to be able to represent multi-argument functions using n-ary abstraction and application.

```
trait Function

trait Lambda extends Function
case class Abs(xs: List[Var], f: Function) extends Lambda
case class App(f: Function, xs: List[Function]) extends Lambda
case class Var(name: VarName) extends Lambda
case class Translucent(t: Term, env: Map[VarName, Function]) extends Lambda

case object Identity extends Function
case object Flip extends Function
case object Compose extends Function
```

Normalisation by Rewriting

Essentially term-rewriting, similar to how Parsers are simplified. Strong reduction – allow beta reduction to occur deep inside lambda terms, in all redexes of a term, until no more reductions can be made. To beta normal form, not just beta-WHNF (weak head normal form)

Aside: Untyped lambda calculus is not strongly normalising, but these terms are converted from valid typed Scala programs. Although not formally proven, it is reasonable to assume that these terms are a subset of the simple lambda calculus that are strongly normalising – they won't be particularly complex.

`Function` has one binding construct – the n-ary abstraction. This introduces variables and the need for capture-avoiding substitution. Capture-avoiding substitution is hard, many approaches e.g. de Bruijn indices (but inefficient to open/close terms so much, so the correct choice is levels). * There is a dizzying array of existing literature and techniques on to represent binders without names Attempted a barendregt's but the optimised form (not freshening on every substitution) might be subtly incorrect, not sure tbh.

Normalisation by Evaluation

Fortunately, there is a cheat to completely circumvent the nightmare of capture-avoiding substitution. Unlike traditional normalisation techniques, NbE bypasses rewriting entirely, instead appealing to their denotational semantics. (fuck yeah bitches) Interpret the term into a denotational model which can then be evaluated, and reify the result back into a term. Use HOAS to leverage the meta-language (i.e. Scala) as our semantic model – blazingly fast! (theoretically lol) No rewriting = no worrying about avoiding capture. Shift that burden onto the scala compiler.

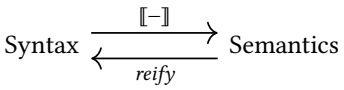

Fig. 1.3: Normalisation by evaluation in a semantic model.

1.2.2 Converting Scalameta Terms to the Function ADT

Three cases:

1. `Term.Function` nodes, representing lambda expressions
2. `Term.AnonymousFunction` nodes, representing lambda expressions using placeholder syntax
3. Any other term, which will be *eta*-expanded if possible

Lambda Expressions

Often found as relatively simple functions to glue together parsers, or transform the result of a parser:

```
val asciiCode: Parsley[Int] = item.map(char => char.toInt)
```

Choose a particularly convoluted example:

```
a => (a, b) => a + b
```

In this example, the a in the function body refers to the a in the second parameter list, as it shadows the a in the first parameter list.

Conversion "flattens" the lambda expression into a chain of n-ary abstractions, with the final term being the body of the lambda. Because we lose scoping information this way, perform α-conversion on variables. The body becomes a `Translucent` term.

```
λ(_l1). λ(_l2, _l3). Translucent(_l2 + _l3, env = {_l1 → _l1, _l2 → _l2, _l3 → _l3})
```

Values in bold are Scalameta tree nodes, so the body term's environment maps `Term.Name` nodes to their corresponding variable terms.

Placeholder Syntax

Scala supports placeholder syntax for terseness, so the earlier parser can be rewritten as:

```
val asciiCode: Parsley[Int] = item.map(_.toInt)
```

Eta-Expansion

```
case class AsciiCode(code: Int)
object AsciiCode extends ParserBridge1[Char, AsciiCode] {
  def apply(char: Char): AsciiCode = AsciiCode(char.toInt)
}
val asciiCode = AsciiCode(item) // desugars to item.map(AsciiCode.apply)
```

1.2.3 Normalising Function Terms

```
trait HOAS
case class HAbs(n: Int, f: List[HOAS] => HOAS) extends HOAS
case class HApp(f: HOAS, xs: List[HOAS]) extends HOAS
case class HVar(name: VarName) extends HOAS
case class HTranslucent(t: Term, env: Map[VarName, HOAS]) extends HOAS
```

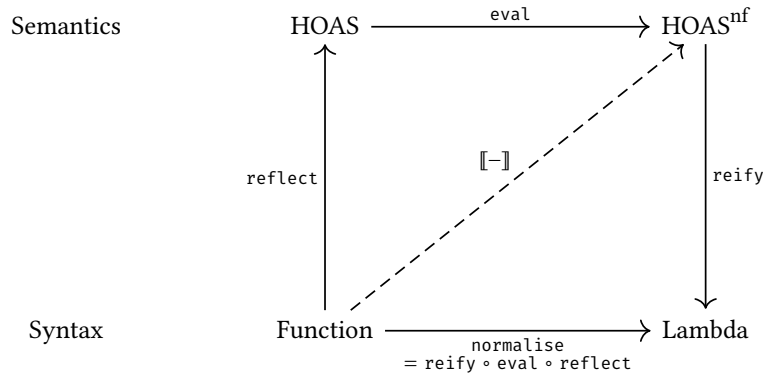

Fig. 1.4: Normalisation by evaluation for the `Function` datatype.

Abs represents n-ary λ-abstraction; App is n-ary function application; Var represents opaque free variables. Translucent is treated as opaque during normalisation, but contains normalisable terms in an environment, where they are substituted back in after normalisation.

Reflection

```
trait Function {
  def reflect: HOAS = {
    def reflect0(func: Function, boundVars: Map[Var, HOAS]): HOAS = func match {
      case Abs(xs, f) =>
        HAbs(xs.size, vs => reflect0(f, boundVars ++ xs.zip(vs)))
      case App(f, xs) =>
        HApp(reflect0(f, boundVars), xs.map(reflect0(_, boundVars)))
      case v @ Var(name) =>
        boundVars.getOrElse(v, HVar(name))
      case Translucent(term, env) =>
        HTranslucent(term, env.mapValues(reflect0(_, boundVars)))

      case Id => HAbs(f => f)
      case Flip => HAbs(f => HAbs(x => HAbs(y => HApp(HApp(f, y), x))))
      case Compose => HAbs(f => HAbs(g => HAbs(x => HApp(f, HApp(g, x)))))
    }

    reflect0(this, Map.empty)
  }
}
```

Evaluation

```
trait HOAS {
  def eval: HOAS = this match {
    case HAbs(n, f) => HAbs(n, x => f(x).eval)
    case HApp(f, x) => f.whnf match {
      case HAbs(_, g) => g(x).eval
      case g          => HApp(g.eval, x.map(_.eval))
    }
    case HTranslucent(t, env) => HTranslucent(t, env.map { case (k, v) => k -> v.eval })
    case HTranslucent(t, env) => HTranslucent(t, env.mapValues(_.eval))
```

```
    case _ => this
  }

  private def whnf: HOAS = this match {
    case HApp(f, x) => f.whnf match {
      case HAbs(_, g) => g(x).whnf
      case g          => HApp(g, x)
    }
    case HTranslucent(t, env) => HTranslucent(t, env.mapValues(_.whnf))
    case _ => this
  }
}
```

Reification

```
trait HOAS {
  def reify: Function = this match {
    case HAbs(n, f) =>
      val params = (1 to n).map(_ => Var.fresh()).toList
      Abs(params, f(params.map(x => HVar(x.name))).reify)
    case HApp(f, xs) => App(f.reify, xs.map(_.reify))
    case HVar(name) => Var(name)
    case HTranslucent(t, env) => Translucent(t, env.mapValues(_.reify))
  }
}
```

1.2.4 Converting Functions back to Scalameta Terms

Surprise bitches same shit again, quasiquotes ftw

1.2.5 Further Work?

Eta reduction – this is more complicated than in Haskell since Scala has special syntax Partial evaluation, not just normalisation (if we reduce to fully closed terms 1+1 can we get it to evaluate to 2? – except currently this would be a Translucent term)

TODO: 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.

* 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

1.3 Limitations

Tried to make Parser and Function typed, but it didn't work out due to limitations in Scala's type inference with GADTs.

Chapter 2
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.

2.1 Avoid Redefining Existing Parsers

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

2.2 Simplify Complex Parsers

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

2.3 Ambiguous Implicit Conversions

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[?]
```

```
//  Note that implicit conversions are not applicable because they are ambiguous:
//   both method stringLift in object character of type (str: String): Parsley[String]
//   and method implicitSymbol in class ImplicitSymbol of type (s: String): Parsley[Unit]
//   are possible conversion functions from String("arnish") to 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 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. 2.1a 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 similar to that shown in fig. 2.1b.

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

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

```scala
object parser {
  import parsley.syntax.character.stringLift
  import lexer.implicits._

  val p = "garnis" <~ 'h'
}

object lexer {
  import parsley.token.Lexer, parsley.token.descriptions.LexicalDesc

  private val lexer = new Lexer(LexicalDesc.plain)
  val implicits = lexer.lexeme.symbol.implicits
}
```

(a) 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._
  ^^^^^^^^^^^^^^^^^^^^^^^^
```

(b) The warning message reported at line 3 by the *Ambiguous Implicit Conversions* lint rule.

Fig. 2.1: Example of the *Ambiguous Implicit Conversions* lint rule in action.

* 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

2.4 Remove Explicit Usage of Implicit Conversions

2.5 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

2.6 Left Recursion Factoring