Chapter 1
Idiomatic Implicits Usage

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.

1.1 Ambiguous Implicit Conversions

1.1.1 Problem

Unfortunately, implicit conversions are a double-edged sword. By their very nature, they can obscure the flow of the program, making it difficult to understand what 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. The two implicit conversions cannot be used in conjunction: the *Implicit Lexer* pattern is a specialisation of the former, so `implicitSymbol` is meant to be a *replacement* for `stringLift` For novice users, this may not be immediately apparent and it is easy to accidentally bring both sets of these implicits into scope; anecdotally, this issue has been encountered by a number of WACC students at Imperial.

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"
//                   ^^^^^^^^
```

In this instance, 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'
//            ^^^^^^^^^^^
```

1.1.2 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.

1.1.3 Example

Fig. 1.1a 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. 1.1b.

```
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 }
```

(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. 1.1: Example of the *Ambiguous Implicit Conversions* lint rule in action.

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

1.1.4 Implementation

Since this rule is restricted to only syntactic analysis, the method of detecting clashing implicits is more heuristic than definitive. The general idea will be to deduce the existence of implicits by examining the import expressions in the source file.

- An import expression of the form `import parsley.syntax.character.xxx`, where `xxx` is either a wildcard import or specifically contains the importee `stringLift`, indicates the *Implicit Conversions* pattern on strings.
- The import path for the *Implicit Lexer* pattern is subject to how the user names their lexer object, so determining this is much more heuristic-based.
  - An import containing the keyword `lexer` as well as the keyword `implicit(s)`.
  - The presence of an import containing the keyword `lexer` as well as the keyword `implicit(s)`, or an importee called `implicitSymbol`

  if present, indicates the *Implicit Conversions* pattern.

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

1.2 Remove Explicit Usage of Implicit Conversions

Chapter 2
Removing Left-Recursion

2.1 Problem

Left-recursive grammar rules are a common pattern to represent left-associativity. Take for example the following definition of a left-associative addition operator:

$$\langle expr \rangle ::= \langle expr \rangle \text{ `+' } \langle term \rangle \mid \langle term \rangle$$

Since the first production of $\langle expr \rangle$ is itself, this rule is said to be left-recursive. This poses a problem for recursive-descent parsers, such as those that `parsley` produces: it will try to parse $\langle expr \rangle$ by first trying to parse $\langle expr \rangle$, and so on, resulting in an unproductive infinite loop.

Although it is possible to address the issue by transforming the grammar with algorithms such as Paull's algorithm [Moore 2000], in the context of parser combinators this is considered an anti-pattern by Willis and Wu [2021]. They argue that this transformation obscures the original intent of the grammar, and exposes lower-level implementation details when this can be abstracted behind a combinator. Instead, they propose that the idiomatic method to handle left-recursion in parser combinators is to use the `chain` family of combinators [Fokker 1995]. These combinators encapsulate the behaviour of right-associating left-recursive rules and correcting the result back to a left-associative form.

Left-recursion often comes as a nasty surprise for novice users naïvely translating BNF grammars into parser implementations – this issue is not unique to parser combinators, but also extends to many popular parser generators that use recursive-descent. Thus, it would be beneficial to provide a linting rule for `parsley` that can warn users when parsers are left-recursive. In fact, `parsley` 5.0 is already planned to introduce a `detectDivergence` combinator, which performs *dynamic* analysis to detect unproductive looping at runtime. It would therefore be ideal if `parsley-garnish` could complement this functionality with an auto-fix rule to refactor left-recursive parsers to use `parsley`'s idiomatic `chain` combinators.

2.2 Solution

`parsley-garnish` bases its left-recursion factoring transformation on the work of Baars and Swierstra [2004], adapted to fit the PEG semantics of `parsley`. At a high-level, the transformation involves "unfolding" each non-terminal production into three parts:

- `results`: The semantic actions of the parser, if it can derive the empty string. Conceptually, this has type `Option[A]` where `A` is the type of the result.
- `nonLeftRec`: The non-left-recursive part of the parser that does not derive the empty string. This will have some type `Parsley[A]`.
- `leftRec`: The left-recursive call, which in the general left-recursive case, corresponds to a repeated postfix operator of type `Parsley[A => A]`. This is a function which requires the semantics of the left-recursive non-terminal argument.

This unfolded parser can then be recombined using the `chain.postfix` combinator. This combinator encapsulates the general form of left-associative parsing, and most other iterative combinators can be derived from it [Willis 2023].

```
// def postfix[A](p: Parsley[A])(op: => Parsley[A => A]): Parsley[A]

val result = results match {
  case None    => empty
  case Some(x) => pure(x)
```

```
}
val transformed = chain.postfix(nonLeftRec | results)(leftRec)
```

This transformation is applied in-order to each parser in the source file, replacing the original parser with its factored form if it was left-recursive.

2.3 Necessary Infrastructure

The comparatively simple linting rules discussed in the previous chapter were implemented by directly inspecting the generic Scala AST provided by Scalafix. However, even though parsley programs are written in Scala, it is important to remember that parsley is a DSL borrowing Scala as a host language. Domain-specific transformations like left-recursion factoring are therefore naturally defined as transformations on the parsley AST, at a higher level of abstraction than the generic Scala AST.

Thus, this section discusses the extra infrastructure used to support the left-recursion factoring transformation:

- §2.3.1 motivates the idea of using an intermediate AST representation for parsers, distinct from the general-purpose Scala AST.
- §2.3.2 shows how the AST of a Scala source file is converted into this intermediate representation, whereas §2.3.3 discusses how this is converted back into Scala code so that it can be applied as a Scalafix patch.

Running example The following left-recursive parser and its transformation into a non-left-recursive form will be used as a representative example for the remainder of this chapter:

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

The example parser intends to express the following simple grammar expressed using left-recursion. The goal is to refactor example so that it retains the intended semantics, but uses the postfix combinator so that parsley can parse it correctly.

$$\langle example \rangle ::= \langle example \rangle \text{ “a” } | \text{ “b”}$$

2.3.1 An Intermediate AST

The transformations described by Baars and Swierstra [2004] require an explicit representation of the grammar and production rules so that they can be inspected and manipulated before generating code. They achieve this by representing parsers as a deep-embedded datatype in the form of an intermediate AST, in a similar manner to parsley.

Since parsley-garnish is a linter, by nature, it has access to an explicit grammar representation in the form of the full scala.meta.Tree AST of the source program. However, this datatype represents general-purpose abstract Scala syntax, rather than the abstract syntax of a specialised parser combinator DSL. This makes it not well-suited for performing domain-specific operations over the AST.

Take for example the task of combining two AST nodes Term.Name("p") and Term.Name("q"), representing named parsers p and q, with the *ap* combinator <*>. This operation can be concisely expressed with Scalameta quasiquotes, rather than manually writing out the full explicit AST:

```
q"p <*> q" ==
  Term.ApplyInfix(
    Term.Name("p"),
    Term.Name("<*>"),
    Type.ArgClause(Nil),
    Term.ArgClause(List(Term.Name("q")), None)
  )
```

However, the operation of inspecting the individual parsers p and q is not as straightforward. Although quasiquotes can be used as extractor patterns in pattern matching, this usage is discouraged due to limitations in their design that makes it easy to accidentally introduce match errors[1]. Thus, extracting the parsers necessitates a long-winded pattern match like so:

```
val ap = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

def deconstructAp(parser: Term) = parser match {
  case Term.ApplyInfix(p, ap(_), _, Term.ArgClause(List(q), _)) => (p, q)
}
```

This involves dealing with abstract general-purpose syntax constructs like `Term.ApplyInfix`, which are low-level details not relevant to the task of manipulating parsers. Although this is not an issue for simple one-off transformations, for more specialised transformations like left-recursion factoring, it would be desirable to abstract away from these low-level syntactic details. This motivates the need for an higher-level, intermediate AST representation that is more specialised to the domain of parser combinators.

The Parser ADT

`parsley-garnish` therefore takes a similar approach as Baars and Swierstra [2004] and `parsley` itself, building an intermediate AST as a deep-embedded parser combinator tree. Fig. 2.1 shows how this is implemented as a `Parser` algebraic data type (ADT). All `Parser` types represent `parsley` combinators, with the sole exception of `NonTerminal` to represent references to named parsers.

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

Fig. 2.1: A subset of the core combinators in the `Parser` ADT.

Deconstructing parsers Syntactic sugar for deconstructing parsers is provided by `unapply` methods on symbolic extractor objects. This makes pattern matching on parsers feel more natural:

```
object <*> {
  def unapply(parser: Ap): Option[(Parser, Parser)] = Some((parser.p, parser.q))
}

def deconstructAp(parser: Parser) = parser match {
  case p <*> q => (p, q)
}
```

Constructing parsers Defining infix operators as extension methods on the `Parser` trait provides syntactic sugar for constructing parsers:

```
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: Term): Parser = FMap(p, f)
```

[1] https://scalameta.org/docs/trees/guide.html#with-quasiquotes-1

```
}

implicit class MultiParserOps(private val ps: List[Parser]) extends AnyVal {
  def zipped(f: Term): Parser = Zipped(f, ps)
}
```

This makes working with `Parser` terms feel closer to writing `parsley` code. For example, notice how constructing the *code* representation of the `example` parser resembles how the original parser itself would be written:

```
val exNT = NonTerminal(Sym(Term.Name("example").symbol))

// val ex: Parsley[String] =     (ex,  string("a")).zipped(  _ + _ )  |  string("b")
   val ex: Parser          = List(exNT,   Str("a")).zipped(q"_ + _") <|>    Str("b")
```

2.3.2 Lifting to the Intermediate Parser AST

Converting the raw Scala AST to this intermediate parser combinator AST requires the following basic operations:

1. Identifying all named parsers defined in the source program – these correspond to non-terminal symbols in the grammar.
2. Lifting the definition each parser into the intermediate AST, i.e. a `Parser` object.
3. Collecting these into a map to represent the high-level grammar – the unique symbol of each named parser is mapped to its corresponding `Parser` object, along with extra meta-information required for the transformation.

Most importantly, this meta-information includes a reference to a parser's original node in the Scala AST, so lint diagnostics or code rewrites can be applied to the correct location in the source file:

```
case class ParserDefn(name: Term.Name, parser: Parser, tpe: Type.Name, originalTree: Term)
```

Identifying Named Parsers

Finding AST nodes corresponding to the definition sites of named parsers involves pattern matching on `val`, `var`, and `def` definitions with a type inferred to be some `Parsley[_]`. This type information is accessed by querying the Scalafix semantic API for the node's symbol information. Consider the labelled AST structure of the `example` parser:

```
// lazy val example: Parsley[String] = (example, string("a")).zipped(_ + _) | string("b")
// ^^^^     ^^^^^^^  ^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// mods      pats        decltpe                              rhs

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

Note that the `decltpe` field refers to the *syntax* of the explicit type annotation, not the *semantic* information the variable's inferred type. Therefore, this field will not always be present, so in the general case, the type must be queried via a symbol information lookup like so:

```
exampleTree match {
  case Defn.Val(_, List(Pat.Var(varName)), _, body) =>
    println(s"qualified symbol = ${varName.symbol}")
    varName.symbol.info.get.signature match {
      case MethodSignature(_, _, returnType) =>
        println(s"type = $returnType")
        println(s"structure of type object = ${returnType.structure}")
    }
}
// qualified symbol = path/to/package/ObjectName.example.
// type = Parsley[String]
// structure of type object = TypeRef(
//   NoType,
//   Symbol("parsley/Parsley#"),
//   List(TypeRef(NoType, Symbol("scala/Predef.String#"), List()))
// )
```

Seeing that the type of this AST node is `Parsley[String]`, `parsley-garnish` can then proceed to convert the `rhs` term into a `Parser` ADT object. The map entry uses the fully qualified symbol for `example` as the key, and the lifted `Parser` object as the value.

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` to instantiate a parser from the appropriate `scala.meta.Term`. These `fromTerm` methods perform the menial work of pattern matching on the low-level syntactic constructs of the Scala AST. All `fromTerm` methods are combined to define the `toParser` extension method on `scala.meta.Term` – this is where AST nodes are lifted to their corresponding `Parser` representation.

The pattern matching example from §2.3.1 makes a reappearance in the definition of `Ap.fromTerm`, where the arguments to the `<*>` combinator are instead recursively lifted to `Parser` objects:

```
// Type signatures in Parsley:
// p: Parsley[A => B], q: =>Parsley[A], p <*> q: Parsley[B]
case class Ap(p: Parser, q: Parser) extends Parser
object Ap {
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

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

Where a combinator takes a non-parser argument, this is treated as a black box and kept as a raw AST node of type `scala.meta.Term`:

```
// x: A, pure(x): Parsley[A]
case class Pure(x: Term) extends Parser
object Pure {
  val matcher = SymbolMatcher.normalized("parsley.ParsleyImpl.pure")
```

```
  def fromTerm: PartialFunction[Term, Pure] = {
    case Term.Apply(matcher(_), Term.ArgClause(List(expr), _)) => Pure(expr)
  }
}
```

Building the Grammar Map

The overall process of converting the source file AST to a high-level map of the grammar can therefore be expressed as a single traversal over the AST:

```
object VariableDecl {
  def unapply(tree: Tree): ParserDefn = tree match {
    case Defn.Val(_, List(Pat.Var(varName)), _, body) if isParsleyType(varName) =>
      ParserDefn(
        name = varName,
        parser = body.toParser,
        tpe = getParsleyType(varName),
        originalTree = body
      )
    // similar cases for Defn.Var and Defn.Def
  }
}

val nonTerminals: Map[Symbol, ParserDefn] = doc.tree.collect {
  case VariableDecl(parserDef) => parserDefn.name.symbol -> parserDef
}.toMap
```

2.3.3 Lowering Back to the Scalameta AST

After all necessary transformations have been applied to parser terms, the final 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. For example:

```
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.

2.3.4 Implementing the Left-Recursion Transformation

TODO

Core combinators: `NonTerminal`, `Pure`, `Empty`, `Ap`, `Choice`. Combinators like `String` are theoretically core combinators but they represent boring cases. Some composite combinators are supported, and desugared into the core combinators.

Can derive empty string? (good resource from packrat parsing paper) pure(x) – yes, semantic action is x empty – no p <|> q – if p or q can derive empty, peg is ordered so semantic action is pe if it can derive empty, else qe p <*> q – if p and q can derive empty, semantic action is pe(qe) due to pure(f) <*> pure(x) == pure(f(x)) law string – only if given argument "", but this also illegal in parsely – explicitly triggers a runtime error, so basically no [error] java.lang.IllegalArgumentException: requirement failed: ‘string‘ may not be passed the empty string (‘string("")‘ is meaningless, perhaps you meant ‘pure("")‘?) char – no (not implemented as a core comb, should do

that) item – no (not implemented as a core comb, should do that?) many(p) – yes (not implemented properly?) – semantic action is empty list?? some(p) – if p can derive empty (not a primitive, defined in terms of many – p <::> many(p)) NT – if referenced rule can derive empty

Defining Utility Functions

The transformation requires the use of three higher-order functions:

- The identity function `identity[A]: A => A` is defined in the standard library.
- The flip function reverses the order of arguments applied to a function. This isn't defined in the standard library, so it must be defined manually.
- Function composition is defined in the standard library, but a more versatile curried version is required by the transformation, so it is also defined manually.

Therefore, `parsley-garnish` will insert the following definitions into the source file as a patch:

```
  def flip[A, B, C](f: A => B => C)(x: B)(y: A): C = f(y)(x)
  def compose[A, B, C](f: B => C)(g: A => B)(x: A): C = f(g(x))
```

This brings these higher-order functions into scope, allowing the transformed code to make use of it.

The `unfold` method is defined for every single combinator in the `Parser` ADT. Most important is the `Ap` combinator, which is `parsley-garnish`'s primitive for composing parsers.

```
case class Ap(p: Parser, q: Parser) extends Parser {
  def unfold: UnfoldedParser = {
    val UnfoldedParser(pe, pn, pl) = p.unfold
    val UnfoldedParser(qe, qn, ql) = q.unfold

    val result =
      if (pe.isDefined && qe.isDefined) Some(q"${pe.get}(${qe.get})")
      else None

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

    val nonLefts = {
      val lnl = pn <*> q
      val rnl = pe.map(q"f => qn.map(f)").getOrElse(Empty)
      lnl <|> rnl
    }

    UnfoldedParser(result, nonLefts, lefts)
  }
}
```

Success...?

Running the transformation on the `example` parser yields the output in fig. 2.2. This is... disappointing, to say the least. There are *many* things wrong with the transformed output:

- The parser is horrendously complex and unreadable, its intent entirely obfuscated in a sea of combinators.

```
def flip[A, B, C](f: A => B => C)(x: B)(y: A): C = f(y)(x)
def compose[A, B, C](f: B => C)(g: A => B)(x: A): C = f(g(x))

lazy val example: Parsley[String] = chain.postfix(
  empty | (empty.map((_ + _).curried) | empty <*> example) <*> string("a")
    | string("b") | empty
)(
  (empty.map(flip) <*> example | pure(identity).map(compose((_ + _).curried)))
    .map(flip) <*> string("a")
    | empty | empty
)
```

Fig. 2.2: The initial attempt at factoring out left-recursion from the `example` parser.

- Having to define the `flip` and `compose` functions is not ideal, but inlining them as lambdas would make the code even worse.
- Even worse, the parser does not even typecheck – unlike classical Hindley-Milner-based type systems, Scala only has *local* type inference [Cremet et al. 2006]. As a result, the compiler is unable to correctly infer correct types for `flip` and also asks for explicit type annotations in the lambda `(_ + _).curried`.

Chapter 3
Simplifying Parsers and Expressions

At this stage, the left-recursion factoring transformation leaves a lot to be desired in terms of output quality. This chapter takes a step back from linting rules and focuses on ensuring how transformed terms can get pretty-printed in a human-readable form. The following ideas are explored:

- First, §3.1 discusses how parser terms can be simplified via domain-specific optimisations based on parser laws.
- Afterwards, §3.2 discusses how expressions can be partially evaluated to some extent. This is achieved using another intermediate AST, this time based on the λ-calculus, which unlocks the idea of β-reduction and normalisation as tools to reduce the complexity of these terms.

3.1 Simplifying Parsers

Reusing a similar abstract syntax representation as `parsley` itself unlocks some interesting insights for `parsley-garnish`. Gibbons and Wu [2014] note that a deep-embedded DSL consists of two components:

1. A representation of the language's abstract *syntax*, in the form of the aforementioned datatype.
2. Some traversals over the datatype, which gives *semantics* to that syntax.

A deep-embedded DSL and a linter for that DSL can thus be viewed as two different semantic interpretations over the same abstract syntax:

- The DSL semantics are *evaluation*. The syntactic structure may be transformed for optimisation purposes before generating code to be evaluated.
- The linter's semantics are two-fold for lint diagnostics and code rewrites:
  - *Emitting side-effects* in the form of diagnostics, based on patterns of interest within the syntactic structure.
  - *Pretty-printing* a transformation over the syntactic structure, as a rewrite action – crucially, unlike evaluation, the transformed output is not converted into code but rather a textual representation to be rewritten over the original source file. The output of this transformation may benefit from *the same optimisation transformations as with the DSL semantics* to simplify the pretty-printed textual output.

This section shows that this is indeed the case for `parsley-garnish`: the same optimisation transformations apply for both `parsley` (the DSL) and `parsley-garnish` (the linter). The only difference lies in the purpose of performing these transformations:

- `parsley-garnish` needs to perform simplifications on the `Parser` AST to produce output of hand-written quality, or else the resulting parser would be unreadable.
- `parsley` performs simplifications on its combinator tree to produce output of hand-written quality, in order to deliver excellent parser performance.

3.1.1 Parser Laws

Willis, Wu, and Pickering [2020] note that parser combinators are subject to *parser laws*, which often form a natural simplification in one direction. Both `parsley` Scala [Willis and Wu 2018] and `parsley` Haskell [Willis 2023] use these laws as the basis for high-level optimisations to simplify the structure of deeply-embedded parsers. These same principles are used by `parsley-garnish` to simplify parser terms to resemble the natural style that a human would write by hand.

Fig. 3.1 shows the subset of parser laws utilised by `parsley-garnish` for parser simplification. Most of these laws have already been shown to hold for Parsley by Willis and Wu [2018]; an additional proof for eq. (3.8) can be found in **??**.

```
p.map(f).map(g) = p.map(g compose f)                    (3.1)
  pure(f) <*> pure(x) = pure(f(x))                      (3.2)
      pure(f) <*> x = x.map(f)                          (3.3)
            empty | u = u                               (3.4)
            u | empty = u                               (3.5)
        pure(x) | u = pure(x)                           (3.6)
         empty <*> u = empty                            (3.7)
        empty.map(f) = empty                            (3.8)
```

Fig. 3.1: Functor (3.1), Applicative (3.2, 3.3), and Alternative (3.4–3.8) laws.

Simplifying the Example Parser

It is useful to illustrate how these laws are used to simplify a parser term, by starting with the parser in fig. 2.2. First of all, most of the noise in `example` comes from the large number of `empty` combinators. These can be eliminated using eqs. (3.4), (3.5), (3.7), and (3.8):

```
lazy val example: Parsley[String] = chain.postfix(string("b"))(
  (pure(identity).map(compose((_ + _).curried))).map(flip) <*> string("a")
)
```

This already looks a lot better, but the second parameter to `postfix` can be further simplified as follows:

```
    (pure(identity).map(compose((_ + _).curried))).map(flip) <*> string("a")
=       { eqs. (3.2) and (3.3) }
    pure(compose((_ + _).curried)(identity)).map(flip) <*> string("a")
=       { eqs. (3.2) and (3.3) }
    pure(flip(compose((_ + _).curried)(identity))) <*> string("a")
=       { eq. (3.3) }
    string("a").map(flip(compose((_ + _).curried)(identity)))
```

The final simplified form of the parser is then:

```
val f = flip(compose((_ + _).curried)(identity))
lazy val expr: Parsley[String] = chain.postfix(string("b"))(string("a").map(f))
```

The parser is now expressed in a much simplified form, in a similar style to how it would be written by hand. The remaining challenge is to simplify the contents of the expression `f`, which is tackled in §3.2.

3.1.2 Implementing Rewrites on the Parser AST

Lawful simplifications are applied by a bottom-up transformation over the recursively defined `Parser` AST. Since there are many parser cases, this inevitably leads to repetitive and error-prone boilerplate code which simply exists to recursively propagate the transformation through each case. To avoid this, the recursive traversal itself can be decoupled from the definition of the transformation function. Although the traversal is still hand-written, this implementation is inspired by the generic traversal patterns offered by Haskell's `uniplate` library [Mitchell and Runciman 2007].

The traversal 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 {
  case FMap(FMap(p, f), g) => FMap(p, compose(g, f))
  case Pure(f) <*> Pure(x) => Pure(App(f, x))
  case u <|> Empty => u
  case Pure(f) <|> _ => Pure(f)
  ...
}
```

Extensibility and Safety Further design considerations are made to ensure the extensibility 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. Although this formulation of the traversal is inspired by generic traversals, it still manually defines the traversal for each case: a safer approach would be to generically derive this. In Scala, this would require the use of an external dependency such as `shapeless`[1], which is frankly overkill given the relative simplicity of the `Parser` ADT.

3.1.3 Discussion

The design of the parser simplification process for `parsley-garnish` was not intended to closely follow the methods used in `parsley`, so it is remarkable that the two approaches have ended up being so similar. However, in retrospect, this resemblance is not surprising given that the act of parser simplification and optimisation are fundamentally the same transformation. Since both `parsley` and `parsley-garnish` represent the parser AST as a deep-embedded structure, it is natural that this transformation is implemented similarly in both cases as a bottom-up traversal over the abstract syntax.

This insight can be extended to any deep embedded DSL based on an algebra, where constructs within the DSL are subject to algebraic laws and operations. It would be interesting to see an EDSL and linter pair that shares a unified data structure for its abstract syntax, in order to take full advantage of this duality.

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

3.2 Representing and Normalising Expressions

The previous section demonstrated the process of simplifying the `Parser` AST, but this is not the only syntactic structure that requires simplification. So far, parsers such as `pure` and `map` still treat expressions as black boxes in the form of raw `scala.meta.Term` AST nodes. This is evident from where the example in §3.1.1 left off, where the parser itself is in a simplified form, but the function passed to `map` isn't:

```
val f = flip(compose((_ + _).curried)(identity))
```

Therefore, this section explores the following:

- How expressions can be represented as another intermediate AST, so that they are statically inspectable enough to be simplified.
- The notion of *normalisation*, reducing expressions into a semantically equivalent but syntactically simpler form.

3.2.1 The *n*-ary Lambda Calculus

Once again, the complexity of manipulating the generic Scalameta AST can be avoided by building a new intermediate AST representation for expression terms.

Scala, as a functional programming language, uses an extension of the λ-calculus [Church 1936] as its theoretical foundations [Cremet et al. 2006; Amin et al. 2016]. The expression terms that we want to normalise are equivalent to λ-terms, just with extra syntactic sugar. In the standard λ-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*. Scala supports curried functions using multiple parameter lists, but uncurried functions are preferred for performance reasons. Since these functions will be transformed from Scala code and back, it is desirable to maintain a high-level equivalence between these two representations. Thus, the expression AST will be based on fig. 3.2, which extends the λ-calculus to support proper multi-argument functions using n-ary abstraction and application.

β-Reduction and α-Conversion

In the λ-calculus, terms are evaluated via β-reduction: fig. 3.3 shows how this can be defined for the n-ary λ-calculus. Unlike the standard λ-calculus, reduction will only take place if the expected number of arguments in $\overline{\mathbf{x}}$ are equal to the number of arguments in $\overline{\mathbf{N}}$; otherwise, evaluation is stuck.

The syntax $M[N/x]$ denotes term substitution, where all free occurrences of x in M are replaced with N. Substitution must avoid *variable capture*, when N contains free variables that are bound in the scope where x is found [van Bakel 2022]. Avoiding capture is achieved by performing α-conversion, which is the process of renaming bound variables. In the λ-calculus, two terms are considered α-equivalent if they can be transformed into each other by only renaming bound variables: the term $\lambda x.x$ is equivalent to $\lambda y.y$.

$M, N ::= x$	variable
$\mid\ (\lambda\overline{\mathbf{x}}.\ M)$	n-ary abstraction, where $\overline{\mathbf{x}} = (x_1, \ldots, x_n)$
$\mid\ (M\,\overline{\mathbf{N}})$	n-ary application, wher $\overline{\mathbf{N}} = (N_1, \ldots, N_n)$

Fig. 3.2: Syntax for the untyped λ-calculus extended with n-ary abstraction and application.

$$(\lambda \overline{\mathbf{x}}.\ M)\ \overline{\mathbf{N}} \to_\beta M[\overline{\mathbf{N}}/\overline{\mathbf{x}}] \qquad (\text{if } |\overline{\mathbf{x}}| = |\overline{\mathbf{N}}|)$$

Fig. 3.3: The β-reduction rule for the n-ary lambda calculus.

Illustrating variable capture For example, substitution without α-conversion incorrectly β-reduces the following term:

$$\begin{aligned}(\lambda x.\lambda y.xy)y &\to_\beta (\lambda y.xy)\ [y/x] \\ &= \lambda y.yy\end{aligned}$$

The y that was substituted was originally a free variable, distinct from the y bound in the lambda $\lambda y.xy$. However, after substitution, it became captured under the lambda, where the two y terms are now indistinguishable in the incorrect expression $\lambda y.yy$. The correct β-reduction with capture-avoiding substitution would instead proceed as follows:

$$\begin{aligned}(\lambda x.\lambda y.xy)y &\to_\beta (\lambda y.xy)\ [y/x] \\ &=_\alpha (\lambda z.xz)\ [y/x] \\ &= \lambda z.yz\end{aligned}$$

Simplifying the Example Expression

The example from the beginning of the section can thus be evaluated by hand via β-reduction, representing the higher-order functions as λ-abstractions:

$$\begin{aligned}\texttt{flip(compose((\_ + \_).curried)(identity))} &= \text{flip (compose } (\lambda a.\lambda b.a + b) \text{ identity)} \\ &= \text{flip } ((\lambda f.\lambda g.\lambda x.f\,(g\ x))(\lambda a.\lambda b.\ a + b)(\lambda x.x)) \\ &\to_{\beta*} \text{flip } (\lambda g.\lambda x.\ (\lambda b.\ g\ x + b)(\lambda x.x)) \\ &\to_{\beta*} \text{flip } (\lambda x.\lambda b.\ x + b) \\ &= (\lambda f.\lambda x.\lambda y.\ f\ y\ x)(\lambda x.\lambda b.\ x + b) \\ &\to_{\beta*} \lambda x.\lambda y.\ y + x\end{aligned}$$

This normalised expression has the same meaning as the original, but is now suitable to be placed in the code rewrite! The rest of the section now explores how this process can be implemented in `parsley-garnish`.

3.2.2 Representing Names

There exists a plethora of approaches to implementing the λ-calculus, mostly differing in how they represent variable names. This affects how variable capture is handled, and also how α-equivalence of two terms can be determined. For `parsley-garnish`, cheap α-equivalence is desirable to help check equivalence of parser terms, which is useful for some transformations.

Naïve capture-avoiding substitution Representing variable names as strings is the most straightforward approach in terms of understandability. The example below shows how the simply typed λ-calculus can be represented as a generalised algebraic data type (GADT) [Cheney and Hinze 2003] in Scala:

```
type VarName = String

trait Lambda
case class Abs[A, B](x: Var[A], f: Lambda[B]) extends Lambda[A => B]
case class App[A, B](f: Lambda[A => B], x: Lambda[A]) extends Lambda[B]
```

```
case class Var[A](name: VarName) extends Lambda[A]

// λf. λx. f x
val f = Var("f")
val x = Var("x")
val expr = Abs(f, Abs(x, App(f, x)))
```

Although naïvely substituting these terms seems logically simple, it can be very tricky to get right. This approach requires calculating the free variables in a scope before performing substitution, renaming bound variables if it would lead to variable capture. Due to the inefficiency of having to traverse the whole term tree multiple times, this approach is not used in any real implementation of the λ-calculus. Furthermore, checking α-equivalence is also tedious, requiring another full traversal of the term tree to compare variable names.

Barendregt's convention Renaming all bound variables to be unique satisfies *Barendregt's convention* [Barendregt 1984], which removes the need to check for variable capture during substitution. However, to maintain this invariant, variables must also be renamed during substitution – this administrative renaming incurs a relatively high performance overhead and chews through a scarily large number of fresh variable names. The approach has been successfully optimised to very impressive performance, though: the Haskell GHC compiler uses Barendregt's convention with a technique dubbed "the Rapier" [Peyton Jones and Marlow 2002], maintaining further invariants to avoid renaming on substitution when unnecessary. Unfortunately, maintaining the invariants to keep this transformation correct becomes very difficult [Maclaurin, Radul, and Paszke 2023].

Nameless and hybrid representations Nameless representations like *De Bruijn indices* [de Bruijn 1972] eschew names entirely, instead representing variables as the number of binders between the variable and its binding site. This makes α-equivalence trivial to check, as it is just a matter of comparing the indices. Although an elegant representation, De Bruijn terms are notoriously difficult to work with, as they are not easily human-readable. Furthermore, performing substitutions with De Bruijn terms has an overhead as variable positions have to be shifted – this is undesirable given that the purpose of the AST is to normalise λ-terms. To avoid this, hybrid representations combining named and nameless representations exist [McBride and McKinna 2004; Charguéraud 2012], but they become rather complex solutions for what should be a relatively simple λ-calculus implementation for `parsley-garnish`'s needs.

Higher-order abstract syntax Using *higher-order abstract syntax* (HOAS) [Pfenning and Elliott 1988] sidesteps variable binders entirely by borrowing substitution from the meta-language, making it the meta-language's responsibility to handle variable capture instead. In contrast, the previous techniques were examples of first-order abstract syntax, which represents variables and unknowns with identifiers (whether with names or indices). A HOAS approach does not name bound variables, instead representing them as bindings in the meta-language:

```
trait HOAS
case class Abs[A, B](f: HOAS[A] => HOAS[B]) extends HOAS[A => B]
case class App[A, B](f: HOAS[A => B], x: HOAS[A]) extends HOAS[B]

// λf. λx. f x
val expr = Abs(f => Abs(x => App(f, x)))
```

Therefore, this representation performs substitution through Scala's function application, which makes it extremely fast compared to the other approaches. However, since lambda abstractions are represented as closures within Scala itself, the function body becomes wrapped under Scala's variable bindings, making them difficult to inspect and work with.

3.2.3 Normalisation Strategies

One remaining hurdle stands before deciding on an ADT representation: how normalisation will be implemented. The ideas of 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. This section briefly explains the traditional notion of reduction-based normalisation, before introducing normalisation by evaluation as a more elegant and efficient strategy.

Reduction-Based Normalisation

The β-reduction rule is a *directed* notion of reduction, which can be implemented as a syntax-directed term-rewriting system, in a similar way to how `Parser` terms are simplified. The goal is to achieve beta normal form (β-NF) by allowing β-reduction to occur deep inside λ-terms, in all redexes of a term, until no more reductions can be made.

Normalisation by Evaluation

An interesting alternative strategy stems from a notion of *reduction-free* normalisation, based on an undirected notion of term equivalence, rather than directed reduction. *Normalisation by Evaluation* (NBE) [Filinski and Korsholm Rohde 2004] achieves this by *evaluating* syntactical terms into a semantic model, then *reifying* them back into the syntactic domain. The denotational model (denoted by $[\![-]\!]$) generally involves implementing a separate datatype from the syntactic AST representation of functions. The semantics is specifically constructed to be *residualising*, meaning that terms can be extracted out into the original syntactic representation. Normalisation is then just defined as the composition of these two operations, as illustrated in fig. 3.4.

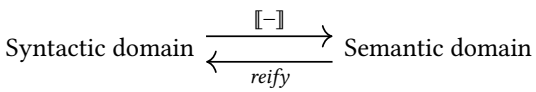
3.2.4 The Expression ADT

The final implementation of the `Expr` AST normalises terms with NBE, which results in a two-tiered representation of expression terms:

1. Scalameta AST nodes corresponding to expressions are lifted to the `Expr` ADT, which represents the syntax of lambda expressions using a simple named approach.
2. `Sem` uses HOAS to leverage Scala semantics as the denotational model for lambda expressions. During normalisation, `Expr` terms are evaluated into `Sem`, then reified back into `Expr`.

This achieves the following desired properties for `parsley-garnish`'s use cases:

- The syntactic `Expr` ADT is represented in a simple manner, which is easy to construct and manipulate as opposed to a HOAS representation. This allows function terms to be pattern matched on, as part of parser simplifications.
- Lifting the syntactic constructs to Scala semantics with HOAS unlocks extremely efficient normalisation, and easier guarantees of correctness with respect to variable capture.
- Reifying `Sem` terms back into syntactic `Expr` terms automatically α-converts names, granting α-equivalence for free.



$$normalise = reify \circ [\![-]\!]$$

Fig. 3.4: Normalisation by evaluation in a semantic model.

Fig. 3.5a shows the implementation of the untyped `Expr` ADT representing the abstract syntax of n-ary λ-terms, extended with the following:

- Optional explicit type annotations for variables – these are not used for type-checking, but are there to preserve Scala type annotations originally written by the user.
- `Translucent` terms to encapsulate open terms holding a `scala.meta.Term` which cannot be normalised further. These carry an environment of variable bindings to substitute back in during pretty-printing – in a metaprogramming context, this is analogous to splicing into a quoted expression.

This structure is largely mirrored by the HOAS-based `Sem` ADT shown in fig. 3.5b, which allows it to be reified back into `Expr` terms.

Constructing higher-order functions `Expr` also implements some helper objects to make it easier to construct and deconstruct single-parameter abstractions and applications:

```
object Abs {
  def apply(x: Var, f: Expr) = AbsN(List(x), f)
  def unapply(func: AbsN): Option[(Var, Expr)] = func match {
    case AbsN(List(x), f) => Some((x, f))
    case _ => None
  }
}

object App {
  def apply(f: Expr, x: Expr) = AppN(f, List(x))
  def apply(f: Expr, xs: Expr*) = xs.foldLeft(f)(App(_, _))
}
```

Using these objects, fig. 3.6 shows how the higher-order functions necessary for left-recursion factoring can be implemented as constructors for `Expr` terms.

Improved type safety The originally intended design was to represent `Expr` as a type-parameterised GADT for improved type safety, where it would be based on a *typed* variant of the λ-calculus. This would've also allowed `Parser` to be represented as a GADT parameterised by the result type of the parser. However, attempting to implement this ran into two main hurdles:

- `Var` and `Translucent` terms would need to be created with concrete type parameters of their inferred types. Scalafix's semantic API is not powerful enough to guarantee that all terms can be queried for their inferred types – in fact, the built-in Scalafix rule *Explicit Result Types* calls the Scala 2 presentation compiler to extract information like this[2]. This solution is complex and brittle due to its reliance on unstable compiler internals, which undermines Scalafix's goal of being a cross-compatible, higher-level abstraction over compiler details.
- Scala 2's type inference for GADTs is less than ideal, requiring extra type annotations and unsafe casts which ultimately defeat the original purpose of type safety. This situation is improved, although not completely solved, in Dotty [Parreaux, Boruch-Gruszecki, and Giarrusso 2019] – but Scalafix does not yet support Scala 3.

Evaluating Performance of Normalisation Strategies

TODO: This is worthy discussion anyway, but is is worth benchmarking the performance and comparing the old and new implementations? Does this count towards evaluation? `parsley-garnish` originally used a named

[2] https://github.com/scalacenter/scalafix/issues/1583

```
trait Expr
case class AbsN(xs: List[Var], f: Expr) extends Expr
case class AppN(f: Expr, xs: List[Expr]) extends Expr
case class Var(name: VarName, displayType: Option[scala.meta.Type]) extends Expr
case class Translucent(t: Term, env: Map[VarName, Expr]) extends Expr
```

(a) The `Expr` ADT for representing the abstract syntax of lambda expressions.

```
trait Sem
case class Abs(paramTypes: List[Option[scala.meta.Type]], f: List[Sem] => Sem) extends Sem
case class App(f: Sem, xs: List[Sem]) extends Sem
case class Var(name: VarName, displayType: Option[scala.meta.Type]) extends Sem
case class Translucent(t: Term, env: Map[VarName, Sem]) extends Sem
```

(b) The `Sem` ADT for representing the residualising semantics of lambda expressions.

Fig. 3.5: The intermediate AST for expressions.

```
/* id : A => A */
def id: Expr = {
  val x = Var.fresh()
  Abs(x, x)
}

/* flip : (A => B => C) => B => A => C */
def flip: Expr = {
  val (f, x, y) = (Var.fresh(), Var.fresh(), Var.fresh())
  Abs(f, Abs(x, Abs(y, App(f, y, x)))) // λf. λx. λy. f y x
}

/* compose : (B => C) => (A => B) => A => C */
def compose: Expr = {
  val (f, g, x) = (Var.fresh(), Var.fresh(), Var.fresh())
  Abs(f, Abs(g, Abs(x, App(f, App(g, x))))) // λf. λg. λx. f (g x)
}
def compose(f: Expr) = App(compose, f)
def compose(f: Expr, g: Expr) = App(compose, f, g)
```

Fig. 3.6: Constructors for higher-order functions represented as λ-expressions in `Expr`.

approach with Barendregt's convention, generating fresh variable names using an atomic counter. However, this required an extra α-conversion pass to clean up variable names before pretty-printing the term, since the fresh variable names were very ugly.

3.2.5 Lifting to the Intermediate Expression AST

RJ: TODO: highlight changes with tcolorbox?

The `Parser` AST is amended to take `Expr` arguments where they used to take `scala.meta.Term` values. Take the `Pure` parser as an example:

```
case class Pure(x: Expr) extends Parser
object Pure {
  def fromTerm: PartialFunction[Term, Pure] = {
    case Term.Apply(matcher(_), Term.ArgClause(List(func), _)) => Pure(func.toExpr)
  }
}
```

The `toExpr` extension method on `scala.meta.Term` is used to lift `Term` AST nodes to `Expr` terms. Expression lifting is invoked whenever a parser expects an expression (whether a function or simple value) as an argument. This section gives a high-level overview of the three cases that `toExpr` handles.

Lambda Expressions

Writing parsers often involves defining simple lambda expressions used to glue together parsers, or to transform the result of a parser, as so:

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

These lambda expressions are represented in the Scalameta AST as `Term.Function` nodes, which are recursively traversed to collect all parameter lists. This is folded into a chain of n-ary abstractions, with the final term being the body of the lambda, which is wrapped into a `Translucent` term.

To ensure that the parameter names in the `Translucent` body term are unique, the parameters are α-converted to fresh names. The body is also transformed to make sure references to these bound variables use their new names: this conversion is well-scoped as it compares terms using their unique Scalameta symbols. The following example illustrates when this is necessary:

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

Although no sane Scala programmer would write this, this lambda demonstrates how it is possible to shadow variables – the `a` in the function body refers only to the `a` in the second parameter list, as it shadows the `a` in the first parameter list. The lifted `Expr` term would then resemble the following λ-calculus expression, where **bold** values correspond to `scala.meta.Term` nodes as opposed to `Expr` values:

```
λ(x1). λ(x2, x3). Translucent(x2 + x3, env = {x1 → x1, x2 → x2, x3 → x3})
```

This shows how the lambda body's environment maps `Term.Name` nodes to their corresponding variable terms. When the term is pretty-printed, each `Term.Name` node is replaced with their corresponding `Expr` term – this is analogous to the splicing operation on quasiquotes:

```
q"x1 => (x2, x3) => $x2 + $x3"
```

Placeholder Syntax

Scala supports a placeholder syntax using underscores to make lambda expressions more concise, so the earlier parser can be rewritten as:

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

Scalameta differentiates between regular lambda expressions and those using placeholder syntax, representing the latter as `Term.AnonymousFunction` nodes. This makes it easy to identify which approach to be taken during conversion. To convert this case, each successive underscore in the expression body is replaced with a fresh variable name. Placeholder syntax creates a fully uncurried function with a single parameter list[3]. Therefore, the converted `Expr` term is always a single n-ary abstraction, where the arguments are the freshly generated variable names in order of their occurrence in the expression body.

Eta-Expansion

If the term is not a lambda expression, `parsley-garnish` attempts to η-expand the term if possible. For example, an idiomatic parser written using the *Parser Bridges* pattern [Willis and Wu 2022] could resemble the following:

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

When `parsley-garnish` converts `asciiCode` to a `Parser`, it desugars the bridge constructor into something resembling `item.map(AsciiCode.apply)`. The η-expanded form of `AsciiCode.apply` would be as follows:

```
(char: Char) => AsciiCode.apply(char)
```

To η-expand `scala.meta.Term` nodes, `parsley-garnish` attempts to look up the method signature of its symbol using Scalafix's semantic API. This is not always possible – in that case, the term can't be statically inspected any further and is just wrapped in a `Translucent` term.

3.2.6 Normalising Expression Terms

Using NBE, normalisation therefore follows a two-step process: `Expr` values `evaluate` into `Sem` values, which are then `reify`ed back into `Expr`:

```
trait Expr {
  def normalise: Expr = this.evaluate.reify
}
```

RJ: Much code in this following section – is it necessary or should it go in an appendix

Evaluation Evaluation proceeds by carrying an environment mapping bound variables to their semantic representations. Evaluating a variable looks up its name in the environment, while evaluating a lambda abstraction produces a closure using the current environment – using HOAS allows these closures to be represented as native Scala closures. The interesting case is evaluating function application: this allows β-reduction within the *semantic domain* at any point within the term, not just on the head term. The function and its arguments are first evaluated separately – then, if the function evaluates to an abstraction, the arguments are passed to the Scala closure `g: List[Sem] => Sem`, collapsing the term structure by one step.

```
trait Expr {
  def evaluate: Sem = {
    def eval(func: Expr, boundVars: Map[Var, Sem]): Sem = func match {
      case v @ Var(name, displayType) =>
        boundVars.getOrElse(v, Sem.Var(name, displayType))
      case AbsN(xs, f) =>
        Sem.Abs(xs.map(_.displayType), vs => eval(f, boundVars ++ xs.zip(vs)))
      case AppN(f, xs) => eval(f, boundVars) match {
```

[3] https://www.scala-lang.org/files/archive/spec/2.13/06-expressions.html#anonymous-functions

```
        case Sem.Abs(_, g) => g(xs.map(eval(_, boundVars)))
        case g => Sem.App(g, xs.map(eval(_, boundVars)))
      }
      case Translucent(term, env) =>
        Sem.Translucent(term, env.mapValues(eval(_, boundVars)))
    }

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

Reification Once the syntactic terms are fully evaluated into their semantics, the expression is normalised to β-NF. Reification is then a simple process of converting each level of the term back into its syntactic counterpart. When a lambda abstraction is reified, bound variables are assigned names from a fresh name supply. This step is what grants α-equivalence for free, as the fresh name generator can be made deterministic: given two terms that evaluate to the same semantic structure, reifying both will yield syntactic representations with the same names.

```
trait Sem {
  def reify: Expr = {
    def reify0(func: Sem)(implicit freshSupply: Fresh): Expr = func match {
      case Abs(tpes, f) =>
        val params = tpes.map(Expr.Var(freshSupply.next(), _))
        Expr.AbsN(params, reify0(
          f(params.map { case Expr.Var(name, tpe) => Sem.Var(name, tpe) } )
        ))
      case App(f, xs) => Expr.AppN(reify0(f), xs.map(reify0))
      case Translucent(t, env) => Expr.Translucent(t, env.mapValues(reify0))
      case Var(name, displayType) => Expr.Var(name, displayType)
    }

    reify0(this)(new Fresh)
  }
}
```

3.2.7 Lowering Back to the Scalameta AST

Lowering expressions back to their `scala.meta.Term` representations is achieved with quasiquotes in the same way as parsers in §2.3.3.

- Lambda abstractions are transformed into a lambda expression of form `q"(...params) => body"`
- Function application is transformed into method calls.
- Variables are simply `Term.Name` nodes with their syntactic names.
- Translucent terms splice their environment bindings back into their term body.

3.2.8 Discussion

`parsley` Haskell, as a staged parser combinator library, also has the ability to inspect and optimise the code of user-defined functions. The approach taken by `parsley-garnish` and parsley share many similarities, both using the λ-calculus as a core language to normalise expressions. In both cases, the need to reduce expression terms is motivated by how parser simplifications involve fusion, which results in function applications that can be partially evaluated.

However, the two have different motivations and requirements for normalising expressions, so their approaches differ in some ways – fig. 3.7 illustrates these differences.

Syntactic representation

Unlike `parsley-garnish`, `parsley` has a two-level syntactic representation for expressions. `Defunc` is akin to a deep embedding of higher-order functions, representing them as a GADT: this process is known as *defunctionalisation* [Reynolds 1972; Danvy and Nielsen 2001]. This helps facilitate certain parser law optimisations which require pattern matching on functions as well as parsers, for example:

```
pure(identity) <*> u = u
```

After this step, `Defunc` values are then brought into the lower-level λ-calculus representation `Lambda`, to be normalised by β-reduction.

At the moment, `parsley-garnish` does not have a need to implement any parser simplifications based on these laws, although this may change in the future. Adding an extra defunctionalised layer to the expression AST would be fairly straightforward.

Normalisation strategy

`parsley` normalises terms to full $\eta\beta$-NF, whereas `parsley-garnish` only normalises to β-NF. This is because η-reduction in Scala 2 is not as straightforward as in Haskell, and is not always possible – in most cases the appropriate reduction is instead to convert lambdas to placeholder syntax. This is left as future work.

In `parsley`, normalisation is implemented as a reduction-based approach over the HOAS `Lambda` datatype. Normalisation by β-reduction with Haskell function application brings this to β-WHNF. Then, code generation brings this further to β-NF as desired, as well as an extra step for η-reduction to put the term into full $\eta\beta$-NF.

The main reason why `parsley-garnish` takes a different normalisation approach is because unlike `parsley`, there is still a need for α-equivalence checking after normalisation. In `parsley`, the normalised forms are immediately utilised for code generation, so they can be kept as HOAS the entire time, without representing variables with any names. Conversely, in `parsley-garnish`, these normalised terms undergo further analysis before being transformed into code patches for pretty-printing.

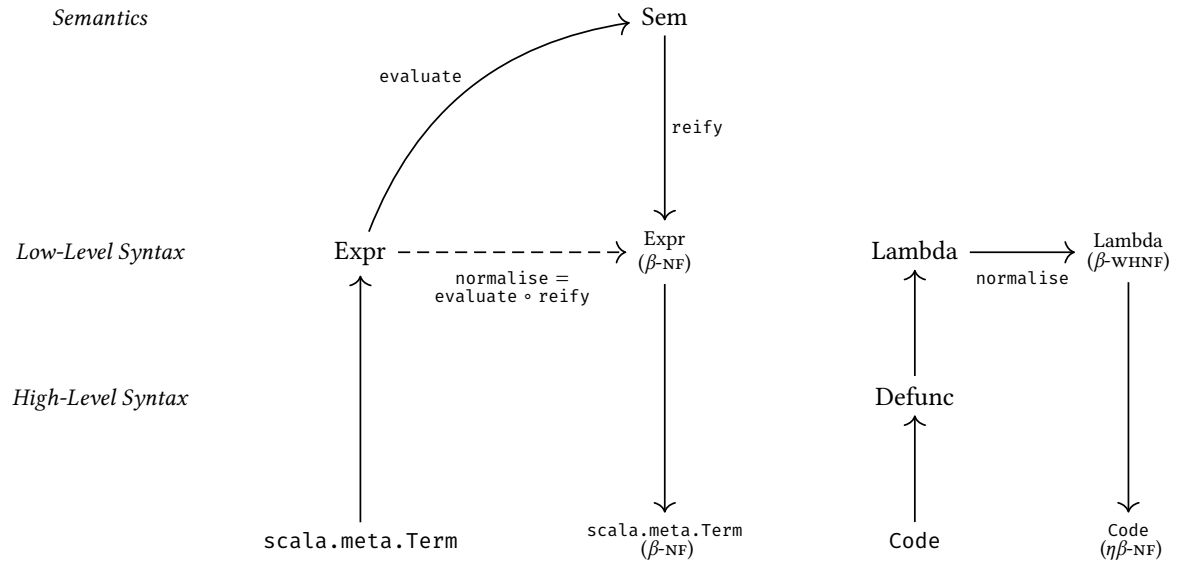


Fig. 3.7: Comparison of expression normalisation in `parsley-garnish` (left) and `parsley` Haskell (right).

Summary

This chapter introduced the idea of simplifying parsers and normalising expressions, by representing both as intermediate ASTs to improve their static inspectability. It also demonstrated how these processes are related to the optimisation techniques used in both `parsley` Scala and `parsley` Haskell.

With promising results applying these simplifications on the `example` parser from last chapter, the improved `Parser` AST now unlocks the potential for more powerful and interesting transformations utilising specialised domain knowledge of parser combinators.

Chapter 4
Lint Rules Using the New Parser ast

4.1 Removing Left-Recursion: Revisited

Armed with the new ast representation of parsers, the left-recursion factorisation transformation can be revisited. What has changed? * Parsers should be simplified before making the patch * Update to use `Expr` rather than quasiquotes, to allow the application of higher-order functions to be partially evaluated * Resugaring * Optimisation: normalise leftrec part to see if it comes to empty – if so, don't inline in the NT case

```
trait Parser {
  def prettify: Parser = this.simplify.normaliseExprs.resugar

  def resugar: Parser = this.rewrite {
    // p.map(\x -> \y -> y) <*> q == p ~> q
    case FMap(p, Abs(_, Abs(Var(y, _), Var(z, _)))) <*> q if (y == z) => p ~> q
    // p.map(\x -> \y -> x) <*> q == p <~ q
    case FMap(p, Abs(Var(x, _), Abs(_, Var(z, _)))) <*> q if (x == z) => p <~ q

    // f.curried.map(p) <*> q == (p, q).zipped(f)
    case FMap(p1, Abs(x1, Abs(x2, body))) <*> p2 =>
      Zipped(AbsN(List(x1, x2), body), List(p1, p2))

  }.transform {
    // Scala 2 cannot resolve implicit stringLifts on "s".map(f)
    case FMap(Str(s, _), f) => FMap(Str(s, implicitSyntax = false), f)
  }

  def normaliseExprs: Parser // applies Expr.normalise on all parsers with Expr arguments
}
```

The Final Result

```
lazy val example: Parsley[String] =
  chain.postfix[String](string("b"))(string("a").map(x1 => x2 => x2 + x1))
```

4.2 Simplify Parser

The improved `Parser` ast also gives a new auto-fix rule for free: automatic simplification of parsers.

Idea * For each parser, simplify it via parser laws, and compare to its original. * If the result is different, apply the simplified version as a patch.

The implementation of the entire rule is only 20 lines long:

```
class SimplifyParser extends SemanticRule("SimplifyParser") {
  override def fix(implicit doc: SemanticDocument): Patch = {
    getAllParserDefns.map { case ParserDefinition(_, parser, _, originalTree) =>
      val simplifiedParser = parser.prettify
      if (parser.normaliseExprs != simplifiedParser) {
        val simplifiedParserTerm = simplifiedParser.term.syntax
        Patch.replaceTree(originalTree, simplifiedParserTerm)
      } else {
```

```
        Patch.empty
      }
    }.asPatch
  }
}
```

* Apply parser laws, re-using Parser and Func representations to do cool things prettify = resugar . normaliseFunctions . simplify

4.3 Avoid Parser Redefinition

Similar in spirit to the previous rule * Catch cases when user manually writes out a parser that is already defined in the library

Section 3.1.2 makes it easy to write syntax-directed rewrite rules on parsers. Less annoying than working with scalameta ast directly better design patterns idk, symbolmatcher etc all in one place defined as a trait for parser

so: Looking for dumb definitions of things e.g. endBy(p, sep) implemented as many(p <* sep); or count implemented as a foldLeft, fold fusion

4.4 Convert to Parser Bridge

* 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