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. When the Scala compiler encounters a type mismatch, it will search for any implicit methods that are in scope to convert the argument to the expected type. For example, the Scala standard library defines an implicit method to convert integers to wider `Long` values[1]. This allows any function expecting a `Long` to be passed an `Int` instead, without needing to explicitly call `toLong`.

```
implicit def int2long(x: Int): Long = x.toLong
```

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, allowing parsers to resemble the structure of a BNF grammar more closely:

```
// Without implicit conversions
string("parsley") | string("garnish")
// With implicit conversions
"parsley" | "garnish"
```

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

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` has first-class support for both design patterns, defining separate implicit methods for each:

- *Implicit Conversions*: the `parsley.syntax.character` package includes implicit methods `stringLift` and `charLift` to lift string and character literals into parsers, respectively.
- *Implicit Lexer*: the `Lexer` class exposes the `implicitSymbol` method to allow string and character literals to parse as proper tokens.

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

[1] https://github.com/scala/scala/blob/v2.13.14/src/library/scala/Int.scala

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

Syntactic or semantic rule? 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[2], 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: examining a file's import clauses to deduce the presence of clashing implicits.

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    val p = "garnis" <~ 'h'
5  }
6
7  import parsley.token.Lexer
8  object lexer {
9    import parsley.token.descriptions.LexicalDesc
10   private val lexer = new Lexer(LexicalDesc.plain)
11   val implicits = lexer.lexeme.symbol.implicits
12 }
```

(a) A minimal Parsley program which fails to compile due to ambiguous implicits in the `parser` object.

```
warning: [AmbiguousImplicitConversions] These imports 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 there is only one of these imports in scope.
```

(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.

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

1.1.1 Implementation as a Syntactic Lint Rule

The rule uses the following heuristics to determine if an import clause brings an implicit into scope:

- An import of the form `import parsley.syntax.character.x`, where `x` is either a wildcard import or specifically contains the importee `stringLift`, indicates the *Implicit Conversions* pattern on string values.
- The import path for the *Implicit Lexer* pattern is subject to how the user names their lexer object, so determining this is more heuristic-based. The following cases are considered indicative of this pattern:
  - An import containing the keyword `lexer`, as well as either `implicit` or `implicits`.
  - An import clause that brings in an importee called `implicitSymbol`.

Unlike Java, Scala allows import clauses to appear anywhere in source files, not just at the top of the file. The flexibility of Scala's import system has two main repercussions:

1. Imports are visible only within the scope in which they are defined. For example, the members imported by lines 2–3 in fig. 1.1a are only available in the `parser` object, and not visible within `lexer`.
2. Imports are resolved lexically in a top-down order within the source file, so members cannot be referenced at any point before they were imported. In fig. 1.1a, the import on line 7 brings parsley's `Lexer` into scope at any scope after that line, but will not be visible before that point.

This behaviour presents extra challenges for this lint rule, as it must be aware of the exact scope of each import to determine which implicits are in scope at any given point.

The lexically ordered management of scope can be respected by visiting all import clauses in the order they appear in the source file. By treating the parent of an AST node as its enclosing scope, it is possible to use ancestor information to determine the scoping relation between two nodes:

```
extension (tree: Tree) {
  def isWithinScope(scope: Tree): Boolean =
    tree.isEqual(scope) || tree.parent.exists(_.isWithinScope(scope))
}
```

Scoping rules can then be respected by using this information to determine which previously visited imports are still in scope at any given point in the traversal. Thus, fig. 1.2 shows the core logic of the rule, implemented as a *tail-recursive* visit over the list of imports: this allows the Scala compiler to optimise the recursion into an efficient iterative loop. The `getClashingImports` method uses the heuristic rules described above to determine which

```
def visit(toVisit: List[Import], seen: List[Import], patch: Patch): Patch = toVisit match {
  case curr :: remaining =>
    val currentScope = curr.parent.get // Parent node represents import's scope
    // Which imports are in scope at this point: lexical ordering is maintained
    // by only considering imports that have been defined in the file so far
    val alreadyInScope = seen.filter(currentScope.isWithinScope(_.parent.get))
    val currentInScope = alreadyInScope :+ curr
    val clashingImports = getClashingImports(currentInScope)
    if (getClashingImports(alreadyInScope).isEmpty && clashingImports.nonEmpty) {
      visit(remaining, currentInScope, patch +
        Patch.lint(AmbiguousImplicitConversionsLint(curr, clashingImports)))
    } else {
      // Continue traversal, discarding imports that have fallen out of scope
      visit(remaining, currentInScope, patch)
    }
  case Nil => patch
}
```

Fig. 1.2: Implementation of the core logic for the *Ambiguous Implicit Conversions* lint rule.

implicits are brought into scope by an import. If an import brings in both types of implicits for the first time within its enclosing scope, a patch is generated to report a diagnostic. The traversal then continues recursively, updating its list of visited imports.

The overall entry point of the rule is thus defined as follows, where the import clauses are gathered in order via a top-down traversal of the AST:

```
override def fix(implicit doc: SyntacticDocument): Patch = {
  val orderedImports = doc.tree.collect { case i: Import => i }
  visit(orderedImports, List.empty, Patch.empty)
}
```

1.2 Remove Explicit Usage of Implicit Conversions

`parsley` users who are new to Scala may not be aware that implicit conversions are automatically applied by the Scala compiler. This misconception has been observed in the wild (i.e. WACC students in the depths of the Imperial computer labs), where users have treated `parsley`'s implicit methods as regular combinators that are explicitly applied to values.

Example This mistake can be classified as a code smell – although it does not cause compilation errors, it results in unnecessarily verbose parsers as seen in fig. 1.3a. Writing parsers like this is counterproductive, as it negates the purpose of the *Implicit Conversions* patterns. Fig. 1.3b shows how the same parsers should be idiomatically written, removing the explicit application of implicit conversion functions: as long as the implicits are in scope, the Scala compiler will automatically apply them.

```
1 import parsley.syntax.character._
2 val p: Parsley[String] = stringLift("parsley")
3 // Using the implicit method's fully qualified name instead of the import
4 val q: Parsley[Char] = parsley.syntax.character.charLift('p')
```

(a) Example parsers which are unnecessarily verbose by explicitly applying implicit conversions.

```
1 import parsley.syntax.character._
2 val p: Parsley[String] = "parsley"
3 // Using the implicit method's fully qualified name instead of the import
4 import parsley.syntax.character.charLift
5 val q: Parsley[Char] = 'p'
```

(b) The same parsers, but with the explicit implicit conversions removed.

Fig. 1.3: Example of the *No Explicit Implicit Conversions* rewrite rule in action.

1.2.1 Implementation as a Semantic Rewrite Rule

Unlike the previous rule, this is a situation where a rewrite rule can be applied to automatically fix the issue. Furthermore, since this issue arises on successfully compiled code, it can be implemented as a Scalafix semantic rule. With access to the semantic API, the rule can resolve the origin symbol of a function name and determine if it belongs to `parsley`'s set of implicit methods, even if the method is imported under a different name:

```
val implicitConv = SymbolMatcher.normalized(
  "parsley.syntax.character.charLift",
  "parsley.syntax.character.stringLift",
  "parsley.token.symbol.ImplicitSymbol.implicitSymbol"
)
```

```
override def fix(implicit doc: SemanticDocument) = doc.tree.collect {
  // Match a function call with a single argument, using the symbol matcher 'implicitConv'
  // The syntactic naming of the function is ignored,
  // as the symbol matcher will pattern match based on its original symbolic name
  case app @ Term.Apply(implicitConv(_), Term.ArgClause(List(liftedArg), _)) =>
    // Replace the function call with just its argument
    Patch.replaceTree(app, liftedArg.syntax)
}.asPatch
```

There is an uncommon edge case, though, where the implicit conversion is not fully brought into scope by an import, but is instead accessed via a qualified name. The `q` parser in fig. 1.3a demonstrates this, where the `charLift` implicit method is referenced directly by its fully qualified name. In this case, the implicit method has to be imported to bring it into scope. This case can be detected by checking if the function call is on a `Term.Select` node, representing a method call on a qualified name:

```
case qualifiedApp @ Term.Apply(
  // Specific case where the 'implicitConv' matches on a qualified Term.Select node
  qual @ implicitConv(_: Term.Select), Term.ArgClause(List(liftedArg), _)
) =>
  // Original patch to remove the explicit function call
  val removeExplicitCall = Patch.replaceTree(qualifiedApp, liftedArg.syntax)
  // Extra patch to add the import clause for the implicit method
  val importQualifiedName = addPatchAbove(qualifiedApp, s"import $qual")
  // Combine the patches to apply both rewrites
  removeExplicitCall + importQualifiedName
```

This places an import clause for the implicit method directly above the qualified method call, as shown in fig. 1.3b. Placing the import next to the implicit conversion is a deliberate design choice: it localises the scope of the implicit conversion to a smaller region, reducing the risk of causing clashing implicits in the global scope. An even more foolproof method would be to wrap the entire section into its own scope, but this would introduce a lot of syntactic noise, when the chances that such a scoping issue would arise is low.

Summary

This chapter introduced two rules to aid users with idiomatic usage of the *Implicit Conversions* family of design patterns. It also serves as a lightweight demonstration of how to implement both syntactic and semantic rules in Scalafix, and why each type of rule is appropriate for the respective issue at hand:

- *Ambiguous Implicit Conversions* (§1.1) is restricted to being a *syntactic* rule, since the nature of the issue means that it must be caught before compilation. The rule issues *lint diagnostics* to warn users of the issue, but cannot automatically fix it as it does not have enough information to resolve the ambiguity.
- *No Explicit Implicit Conversions* (§1.2) is a *semantic* rule as it can be run in a post-compiled state. By having access to semantic information, the linter can be more confident in resolving symbols, rather than relying on syntactic heuristics. The issue is also simpler in nature and is automatically solvable in the general case, so the rule provides *code rewrite* capabilities.

Chapter 2
Removing Left-Recursion

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 embody 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, the next major release of `parsley` 5.0 will introduce a `detectDivergence` combinator, which performs *dynamic* analysis to detect unproductive looping at runtime. Therefore, `parsley-garnish` could complement this functionality with an auto-fix rule to refactor left-recursive parsers to use `parsley`'s idiomatic `chain` combinators.

Running example The following left-recursive parser and its transformation into a non-left-recursive form will be used as an example for 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 is transformed into a parser that `parsley` can handle correctly.

$$\langle example \rangle ::= \langle example \rangle \text{ `a' } \mid \text{ `b'}$$

2.1 The Left-Recursion Factoring Transformation

`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 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. An unfolded parser is recombined using `chain.postfix`: this combinator encapsulates the general form of left-associative parsing, and most other iterative combinators can be derived from it [Willis 2024].

```
def transform(
  results: Option[A], nonLeftRec: Parsley[A], leftRec: Parsley[A => A]
): Parsley[A] = {
```

```
  val result: Parsley[A] = results match {
    case None    => empty
    case Some(x) => pure(x)
  }
  // The type signature of postfix in Parsley:
  // def postfix[A](p: Parsley[A])(op: => Parsley[A => A]): Parsley[A]
  chain.postfix(nonLeftRec | results)(leftRec)
}
```

2.2 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:

- Firstly, §2.2.1 motivates the idea of using an intermediate AST representation for parsers, distinct from the general-purpose Scala AST.
- Following this, §2.2.2 shows how the source Scala AST is converted into this intermediate representation.
- Finally, §2.2.3 discusses how the intermediate AST is converted back into Scala code so that it can be applied as a Scalafix patch.

2.2.1 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, similarly 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 combinator `<*>` (pronounced "ap"). 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)
}
```

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

This involves dealing with abstract general-purpose syntax constructs like `Term.ApplyInfix`, which are low-level details irrelevant 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.

2.2.1.1 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 <*>(p: Parser, q: Parser) extends Parser
case class <|>(p: Parser, q: Parser) extends Parser
case class Str(s: String) extends Parser
case class Chr(c: Char) extends Parser
```

Fig. 2.1: A subset of the `Parser` ADT, representing the core combinators in `parsley-garnish`.

Deconstructing parsers Scala allows users to define symbolic class names (as evidenced by the definitions of `<*>` and `<|>` in fig. 2.1), and provides syntactic sugar to pattern match on these constructors using infix notation. This results in a very natural and readable pattern matching syntax:

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

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

```
extension (p: Parser) {
  def <*>(q: Parser) = <*>(p, q)
  def |(q: Parser) = <|>(p, q)
  def map(f: Term) = FMap(p, f)
}
extension (ps: List[Parser]) {
  def zipped(f: Term) = Zipped(f, ps)
}
```

This makes the syntax for writing `Parser` terms feel natural and similar 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 EXAMPLE = NonTerminal(Sym(Term.Name("example").symbol))

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

2.2.2 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 metadata required for the transformation.

Most importantly, this metadata 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)
```

2.2.2.1 Identifying Named Parsers

Finding AST nodes corresponding to the definition sites of named parsers involves pattern matching on `val`, `var`, and `def` nodes with a type inferred as some `Parsley[_]`. This type information is queried using the Scalafix semantic API on 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:

```
exampleTree match {
  case Defn.Val(_, List(Pat.Var(varName)), _, body) =>
    println(s"qualified symbol = ${varName.symbol}")
    // Query the symbol information of the variable name, and get its type signature
    varName.symbol.info.get.signature match {
      // Scalameta treats this as a zero-arg method, so the relevant part is its return type
      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.

2.2.2.2 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` 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 §2.2.1 makes a reappearance in the definition of `<*>.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 <*>(p: Parser, q: Parser) extends Parser
object <*> {
  // Match the specific symbol for parsley's <*> combinator
  val matcher = SymbolMatcher.normalized("parsley.Parsley.`<*>`")

  def fromTerm: PartialFunction[Term, <*>] = {
    // Pattern match succeeds only if the term has the structure 'p <*> q'
    case Term.ApplyInfix(p, matcher(_), _, Term.ArgClause(List(q), _)) =>
      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] = {
    // expr is an opaque AST node that cannot be further inspected
    case Term.Apply(matcher(_), Term.ArgClause(List(expr), _)) => Pure(expr)
  }
}
```

2.2.2.3 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:

```
// Encapsulate all valid pattern matches into a single extractor object
object VariableDecl {
  def unapply(tree: Tree): ParserDefn = tree match {
    // isParsleyType uses symbol info to check if variable type is Parsley[_]
    case Defn.Val(_, List(Pat.Var(varName)), _, body) if isParsleyType(varName) =>
      // If the pattern match is successful, convert the definition body to a Parser
      // Collect metadata and bundle into a parser definition object
      ParserDefn(
        name = varName, parser = body.toParser,
        tpe = getParsleyType(varName), originalTree = body
      )
```

```
    // ... similar cases for Defn.Var and Defn.Def
  }
}

type Grammar = Map[Symbol, ParserDefn]
val grammar: Grammar = doc.tree.collect {
  // Every AST node that satisfies the pattern match is added to the map
  case VariableDecl(parserDef) => parserDefn.name.symbol -> parserDef
}.toMap
```

2.2.3 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})"
}
```

The `Parser` trait can then define a `toString` method based on Scalameta's AST pretty-printer on the `term` field. This "stringified" term is therefore guaranteed to be syntactically well-formed and is the appropriate type expected by Scalafix's `Patch` methods.

2.3 Implementing the Left-Recursion Transformation

With the `Parser` AST defined, it is now possible to implement the left-recursion factoring transformation in a more high-level manner. Using the `grammar` map collected previously, the overall transformation can be summarised as follows (in Scala-like pseudocode for brevity):

```
val transformedGrammar = grammar.to(mutable.Map)
for (currentNT <- grammar.keys) {
  // Unfold each non-terminal parser into its three parts
  val (results, nonLeftRec, leftRec) = unfold(transformedGrammar, currentNT)
  // Update the grammar with the recombined postfix parser if it was left-recursive
  transformedGrammar(currentNT).parser = transform(results, nonLeftRec, leftRec)
}
// Generate rewrite patches for each parser definition
transformedGrammar.mapValues { defn =>
  Patch.replaceTree(defn.originalTree, defn.parser.toString)
}
```

The unfolding transformation is defined as follows, which is dynamically dispatched to the appropriate combinator's `unfold` method:

```
def unfold(env: Grammar, currentNT: Symbol) =
  env(currentNT).parser.unfold(currentNT, env, visited = Set.empty)
```

2.3.1 Unfolding the Core Combinators

The most important core combinators in the left-recursion transformation are `NonTerminal`, `Pure`, `Empty`, `<|>`, and `<*>`. Character combinators such as `Str` (`string`), `Chr` (`char`), and `Item` (`item`), although technically handled by `parsley-garnish` as primitives, are grouped together as they behave in a simple and uniform manner. The rest of the combinators in `parsley` are defined in terms of the primitive combinators, and are desugared into the core combinators during unfolding.

Non-terminals Non-terminals are the primary base case of the recursive unfolding transformation. There are three cases to consider when a non-terminal symbol is encountered:

- This non-terminal is the same as the current non-terminal being analysed, which means it may be in a left-recursive position. Recall that the `leftRec` result corresponds to a function, where the call to the left-recursive non-terminal is stripped off. Therefore in this case, the left-recursive call is the `identity` function which just returns its argument, i.e. the non-terminal itself. It is not an issue if the non-terminal is not in left-recursive position, as the `leftRec` result will be subsumed by the `nonLeftRec` result during the final recombination.
- The non-terminal refers to a different non-terminal, but has already been visited in the current unfolding process. This means it should not be recursively visited again, as it has already been factored out. It is not left-recursive and the `nonLeftRec` part is just a reference to this non-terminal's name.
- The non-terminal refers to a different non-terminal that has not been visited yet. In this case, the non-terminal is recursively unfolded, and the `visited` set is updated to include this non-terminal.

The three cases can be implemented as so:

```
case class NonTerminal(ref: Symbol) extends Parser {
  def unfold(currentNT: Symbol, env: Grammar) =
    if (ref == currentNT) (None, Empty, Pure(q"identity"))
    else if (visited.contains(ref)) (None, NonTerminal(ref), Empty)
    else env(ref).parser.unfold(currentNT, env, visited + ref)
}
```

Base cases The base cases are mostly straightforward. The unfolded `results` portion can be reasoned with by the combinators' correspondence to PEG expressions [Eichenroth, Rein, and Hirschfeld 2022], as shown in table 2.1. If the combinator can succeed by parsing the empty string, `results` is a `Some` value of its semantic action, otherwise it is `None`. The other two portions are also easy: none are left-recursive, and if the combinator consumes input on success then `nonLeftRec` is just the combinator itself. Thus, the base cases can be summed up as follows:

```
/* Pure */                 (Some(x), Empty, Empty)
/* Empty */                (None, Empty, Empty)
/* p: Str | Chr | Item */  (None, p, Empty)
```

Combinator	Can succeed without consuming input?
`pure(x)`	Yes, semantic action is `x`.
`empty`	No, combinator fails immediately regardless of input.
`string(s)`	Functionally, no. Can theoretically be given the empty string `""`, but this is illegal in `parsley` and triggers a runtime exception.
`char(c)`	No, always consumes the given character.
`item`	No, always consumes any single character.

Table 2.1: Semantic action (`results`) portion behaviour for each base case.

Choice (<|>) This recursive case is also relatively simple: each branch is unfolded, then the `nonLeftRec` and `leftRec` portions are recombined with the `|` combinator. The `results` portion is that of the left branch if it can derive the empty string, otherwise it is the right branch. This behaviour differs slightly from the original presentation of the transformation by Baars and Swierstra [2004], where `|` is based on CFG semantics. The consequence of this is that choice is unbiased in the original transformation, triggering an ambiguity error if both branches can succeed without consuming input. However, since `parsley` uses PEG semantics, this ambiguity is not possible – the choice operation represented by the `|` combinator is *left-biased*, so if the left parser succeeds the right parser is not attempted.

```
case class <|>(p: Parser, q: Parser) extends Parser {
  def unfold() = {
    val (pe, pn, pl) = p.unfold
    val (qe, qn, ql) = q.unfold
    (pe.orElse(qe), pn | qn, pl | ql)
  }
}
```

Ap (<*>) This is an important case, as `parsley-garnish` chooses to use `<*>` as the primitive combinator for composing parsers. It is also the most complex:

- `p <*> q` can succeed without consuming input only if *both* `p` and `q` can do so. Parser combinators abide by the law that `pure(f) <*> pure(x) = pure(f(x))`, so the resulting semantic action is that of `p` applied as a function to the semantic action of `q`.
- In the `nonLeftRec` and `leftRec` portions, if `p` can succeed without consuming input, then `q` must be subjected to the semantic value corresponding to `p` [Swierstra and Duponcheel 1996]. This is achieved using the result shown in eq. (2.1), which is a special case of grammar right-factoring in reverse.
  - The `leftRec` portion requires re-associating semantic actions back to the left, which necessitates using the `flip` higher-order function. Therefore, the left branch of the RHS in eq. (2.1) is reformulated as `p.map(flip) <*> q`.
  - Since `leftRec` is a parser of a function, the right branch also needs adjusting: the function composition operation must be used in order to apply the function to `q` with the `<*>` combinator.

$$\texttt{(p <|> pure f) <*> q} \quad \Rightarrow \quad \texttt{(p <*> q) <|> (f <\$> q)} \tag{2.1}$$

Given the above observations, the implementation of the `<*>` combinator's unfolding is as follows:

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

    val result = for {
      f <- pe
      x <- qe
    } yield App(f, x)

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

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

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

2.3.2 Composite Combinators

The remaining combinators are defined in terms of the core combinators, and are unfolded by recursively unfolding their constituent parts. For example, the `map` combinator is defined as `p.map(f) = pure(f) <*> p`, so its unfolding is simply implemented as:

```
case class FMap(p: Parser, f: Term) extends Parser {
  def unfold() = (Pure(f) <*> p).unfold
}
```

Further high-level combinators are defined in a similar manner. The improved sequencing combinators `lift` and `zipped`, as well as bridge constructors using the *Parser Bridges* design pattern [Willis and Wu 2022], represent a further generalisation of the `map` combinator:

```
liftN(f, p1, ..., pN) // explicit lift syntax, where f is a function of arity N
f.lift(p1, ..., pN) // implicit lift syntax
(p1, ..., pN).zipped(f) // zipped syntax, improves Scala's ability to infer types
F(p1, ..., pN) // parser bridge pattern, abstracts the construction of f behind bridge F

// All are equivalent to the following desugared form,
// which is idiomatic in Haskell, but not in Scala for performance reasons
= pure(f.curried) <*> p1 <*> ... <*> pN
```

These are handled in `parsley-garnish` as the `LiftLike` family of combinators, which converts them into the desugared form in terms of core combinators `pure` and `<*>`, allowing the unfolding transformation to be applied:

```
trait LiftLike extends Parser {
  // foldLeft builds a parser of the form 'pure(f.curried) <*> p1 <*> ... <*> pN'
  def unfold() = parsers.foldLeft(Pure(q"$func.curried"))(_ <*> _).unfold
}
```

2.3.3 Defining Utility Functions

In various places within the unfolding transformations, three higher-order functions are utilised, which may not be provided by the Scala standard library:

- The flip function reverses the order of arguments applied to a function. This is not 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.
- The identity function `identity[A]: A => A` is defined in the standard library.

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))
```

Although this is not very aesthetically pleasing, it is necessary to bring these higher-order functions into scope for the transformed code to make use of them.

Success...?

With the unfolding transformations defined for all core combinators, the left-recursion factoring transformation is now complete. Running the transformation on the `example` parser yields the output in fig. 2.2.

```
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.

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. It's especially frustrating that there are so many `empty` combinators, when both `p | empty` and `empty | p` are just equivalent to `p`.
- 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`.

The result is discouraging especially because it is not impossible to factor out the left-recursion in a nice manner: a hand-written equivalent using `postfix` would resemble the concisely defined parser in fig. 2.3. There is still hope, though – if the `empty` combinators can be removed and something is done about the higher-order functions, perhaps fig. 2.2 could be salvaged into something that looks more like the human-written version.

```
lazy val example: Parsley[String] = chain.postfix(string("b"))(string("a").as(_ + "a"))
```

Fig. 2.3: An idiomatic way to express the `example` parser using `chain.postfix`.

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 parsers 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* as 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, as was the case in fig. 2.2.
- `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 [2024] notes 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, Wu, and Pickering 2020; Willis 2024] 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.

3.1.1.1 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. In the following example, function terms will be greyed out as they are still currently uninspectable – this is work that is addressed in the next section.

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 the hand-written example in fig. 2.3! 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].

Partial functions Scala supports *partial functions*, which are functions only defined for a subset of its possible input values. If invoked on an undefined input, this results in a runtime error. A useful idiom utilising partial functions is the `collect` method on collections, which combines functionality of `filter` and `map` in a clean

manner. This method will attempt to apply a partial function to each element in the collection, and only include the result if the function is defined. In fact, `collect` has already been used in earlier sections to inspect and gather AST nodes of interest in the Scalameta AST:

```
doc.tree.collect { case Term.Name(name) => name }
//                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// A partial function only defined for Term.Name nodes, not on any other type of AST node
```

Rewrite rules on parsers are most naturally expressed as a partial function, since they are only defined for certain parser cases. Therefore, the traversal method `transform` takes a partial function, applying 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 p <*> q       => p.transform(pf) <*> q.transform(pf)
    case Zipped(f, ps) => ps.map(_.transform(pf)).zipped(f)
    case Pure(f)       => Pure(f)
    ...
  }
  // Apply the partial function if defined, otherwise return the original parser
  pf.applyOrElse(parser, identity[Parser])
}
```

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)
}
```

With this, 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 overkill given the relative simplicity of the `Parser` ADT.

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

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

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.

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 §3.1.1.1 left off, where the parser itself is in a simplified form, but the function passed to `map` is not:

```
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*, in order to reduce 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 present within parsers 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. As a result, 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.

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

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

3.2.1.1 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.

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

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

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 [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 $\lambda y.y$.

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

$$\begin{aligned}(\lambda x.\, \lambda y.\, x\, y)\, y &\rightarrow_\beta (\lambda y.\, x\, y)\, [y/x] \\ &= \lambda y.\, y\, y\end{aligned}$$

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 have become indistinguishable in the incorrect expression $\lambda y.yy$. Instead, the correct β-reduction using capture-avoiding substitution would proceed as follows:

$$\begin{aligned}(\lambda x.\, \lambda y.\, x\, y)\, y &\rightarrow_\beta (\lambda y.\, x\, y)\, [y/x] \\ &=_\alpha (\lambda z.\, x\, z)\, [y/x] \\ &= \lambda z.\, y\, z\end{aligned}$$

3.2.1.2 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:

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

This normalised expression has the same meaning as the original, but is now expressed in a clearer form! This is much more suitable to be placed back into the code rewrite, as it is easier to understand by the user. The rest of the section now explores how this process can be implemented in `parsley-garnish`.

3.2.2 Representing Names

There are 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]

val (f, x) = (Var("f"), Var("x"))
val expr = Abs(f, Abs(x, App(f, x))) // λf. λx. 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. For example, $\lambda x.x(\lambda y.xy)$ would be represented as $\lambda.0(\lambda.1\ 0)$: the variable y is replaced with 0 to signify that y is bound by the closest λ binder. The first occurrence of x is replaced with 0 in the same way, but its second occurrence is replaced with 1 since it is bound by the λ two levels up.

An advantage of De Bruijn indexing is that it makes α-equivalence trivial to check, as it is just a matter of comparing the indices. However, 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]

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

3.2.3.1 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.

3.2.3.2 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.

$$\text{Syntactic domain} \underset{\textit{reify}}{\overset{[\![-]\!]}{\rightleftarrows}} \text{Semantic domain}$$

$$\textit{normalise} = \textit{reify} \circ [\![-]\!]$$

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

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.

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.

```
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.

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

```
object Abs {
  // Convenience factory method to create a single-parameter abstraction
  def apply(x: Var, f: Expr) = AbsN(List(x), f)
  // Extractor allows user to pattern match on Abs(x, f)
  // instead of the more tedious 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 {
  // Convenience factory method to apply a single argument to a function
  def apply(f: Expr, x: Expr) = AppN(f, List(x))
  // Apply multiple arguments in sequence, as f(x)(y)(z) instead of f(x, y, z)
  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 have 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.

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

```
/* 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`.

- 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.

3.2.5 Lifting to the Intermediate Expression AST

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.

3.2.5.1 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"
```

3.2.5.2 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.

3.2.5.3 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 cannot 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
}
```

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

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 {
        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 The NBE semantics utilised by `parsley-garnish` are *intensional* [Lindley 2005], meaning that once 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

Once any transformations on the `Expr` terms are complete, they need to be converted back to strings to be placed in a Scalafix patch. This is achieved in the same way as parsers in §2.2.3, by lowering them back as `scala.meta.Term` ast nodes and re-using Scalameta's pretty-printer to generate a syntactically well-formed string representation. These can also be achieved using quasiquotes:

- 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.

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.

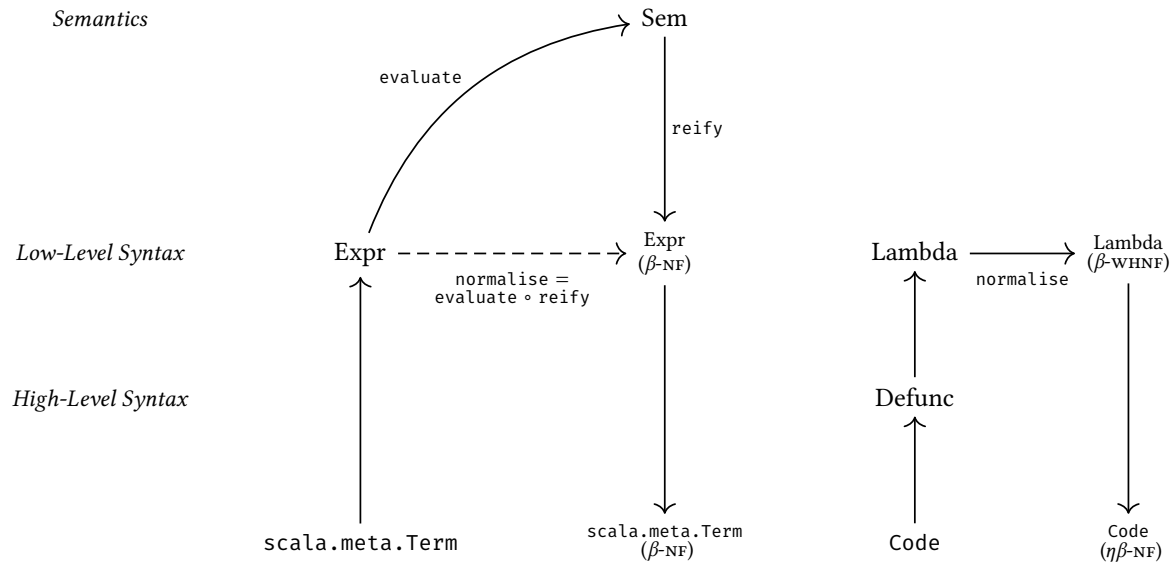

Fig. 3.7: Comparison of expression normalisation in `parsley-garnish` (left) and `parsley` Haskell (right).

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. Implementing 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.

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 unlocks the potential for more powerful and interesting transformations utilising specialised domain knowledge of parser combinators.

Chapter 4
Using the New Parser AST

The ability to simplify parsers, as well as normalising their expression arguments, makes it feasible to use the intermediate `Parser` AST for high-level parser transformations. This chapter first revisits the left-recursion factoring rule from Chapter 2, demonstrating how parser simplification drastically improves the quality of the rule. This is followed by a brief exploration of other rules that can be trivially implemented on top of the intermediate AST, rather than the generic Scalameta AST.

4.1 Removing Left-Recursion: Revisited

Chapter 2 left off with a disappointing result for the left-recursion factorisation transformation, producing a mostly unreadable output in fig. 2.2. The transformation can now be revisited given the improvements made to the `Parser` AST. This section briefly explores some of the changes made to the rule, and the final result of the transformation on the `example` parser.

4.1.1 Pretty Parsers

The need to obtain simpler parser terms was the main driving motivation behind the new `Parser` AST, which now defines the following methods on top of `simplify`:

- `normalise`: Simplifies parsers via parser laws and normalises their expression arguments. This is used to check parser equivalence using their canonical forms.
- `prettify`: Adds an extra step after normalisation, using a set of "resugaring" rules to rebuild the combinators that were desugared during the left-recursion unfolding.

After the left-recursion transformation, `prettify` is used to make parser terms more readable, before applying the final Scalafix patch. The following demonstrates the implementation of these methods, as well as a subset of the resugaring rewrite rules:

```
trait Parser {
  def normalise: Parser = this.simplify.normaliseExprs
  def isEquivalent(other: Parser) = this.normalise == other.normalise
  def prettify: Parser = this.normalise.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
    // 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 in some positions
    // Replace with explicit 'string' combinator so the compiler does not complain
    case FMap(Str(s, _), f) => FMap(Str(s, implicitSyntax = false), f)
  }
}
```

4.1.2 Evaluating Expressions

The higher-order functions `flip` and `compose` are represented as in fig. 3.6, eliminating the need to patch in their definitions as Scala code. Instead, they get partially evaluated as lambda expressions during normalisation.

The currying of functions passed to `LiftLike` combinators are also expressed with the `Expr` AST, to ensure that they are well-formed during the transformation. This also significantly reduces the occurrence of `.curried` method calls in the transformed output, which helps Scala's type inference and reduces syntactic noise in the final output.

```
trait Expr {
  // Curries a lambda expression, flattening into a chain of unary functions
  def curried: Expr = this match {
    case Abs(x, f)        => Abs(x, f.curried)
    case AbsN(x :: xs, f) => Abs(x, AbsN(xs, f).curried)
    case _                => this
  }
}

trait LiftLike extends Parser {
  def unfold() = {
    val curriedFunc = Pure(func match {
      // If the Expr is still opaque, it is treated as before
      case Translucent(term, env) => Translucent(q"($term).curried", env)
      // If statically inspectable, curry the lambda expr properly
      case _ => func.curried
    })
    parsers.foldLeft(curriedFunc)(_ <*> _).unfold
  }
}
```

4.1.3 The Improved Rule

Being able to normalise parsers lets `parsley-garnish` make more informed decisions about the result of the transformation. By normalising the `leftRec` portion of the unfolded parser, it can be determined whether the transformation was successful or not:

- If `leftRec` resolves to `empty`, the original parser was not left-recursive, so `parsley-garnish` does not bother rewriting it.
- If it simplifies to a `pure(x)`, this indicates a deeper problem which rewriting into the `postfix` combinator cannot solve. This generates a lint error diagnostic instead of an auto-fix.
- Otherwise, the parser is left-recursive and can be fixed using the `postfix` combinator. This result is prettified before applying the final patch.

These cases are implemented in the `transform` method to generate the a Scalafix patch per unfolded parser:

```
def transform(results: Option[Expr], nonLeftRec: Parser, leftRec: Parser) = {
  val result = results match {
    case None    => Empty
    case Some(t) => Pure(t)
  }

  leftRec.normalise match {
    case Empty   => Patch.empty // not left-recursive, do not rewrite
    case Pure(_) => Patch.lint(LeftRecDerivesEmptyLint) // left-recursive but unfixable
    case _ =>
      val transformed = Postfix(nonLeftRec | results, leftRec).prettify
      Patch.replaceTree(originalTree, transformed.toString)
  }
}
```

The Final Result

Finally, the `example` parser from CHAPTER 2 can be automatically transformed into the following form:

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

This is a *significant* improvement over the original output in fig. 2.2, and is visually very similar to the "optimal" hand-written version in fig. 2.3. In fact, the two parsers can be proven to be equivalent – **??** explores this further with some different examples.

4.2 Simplify Parsers

The improved `Parser` AST also grants a new auto-fix rule for free: automatic simplification of parsers. At a high-level, the rule works as follows:

- For each parser, get its prettified form and compare it to the original.
- If the result is different, apply the simplified version as a code rewrite.

The implementation of the entire rule is very straightforward, and 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
  }
}
```

This rule therefore lints for overly complex parsers that can be automatically simplified with parser laws. It is unclear how useful this rule would be in practice, but it demonstrates the power of the new `Parser` AST in enabling high-level transformations, and is gained with very minimal effort. For example:

```
/* Before */  val anise = pure("basil").map(add(_, "coriander"))
/* After */   val anise = pure(add("basil", "coriander"))
```

4.3 Avoid Parser Redefinitions

Another rule that can be easily implemented with the new `Parser` AST is to catch cases where a user manually writes out a parser that is already defined in the library. This is similar in spirit to the previous rule, but makes use of the new `rewrite` and `transform` methods on parsers introduced in §3.1.2, which makes it incredibly convenient to write syntax-directed rewrite rules on parsers. Expressing high-level rewrites is much less error-prone than working with the Scalameta AST directly, and prevents re-duplication of efforts. As an example, consider the following rewrite:

```
parser.rewrite { case ManyP(p <~ sep) => EndBy(p, sep) }
```

This catches a case where a user writes a parser in the form of `many(p <~ sep)`, which is already defined in `parsley` as `endBy(p, sep)`. Compare this to the equivalent pattern match if utilising the low-level AST:

```
tree.collect {
  case Term.Apply.After_4_6_0(
    manyMatcher(_),
    Term.ArgClause(
      List(Term.ApplyInfix.After_4_6_0(
        Term.Name("p"),
        Term.Name("<~"),
        Type.ArgClause(Nil),
        Term.ArgClause(List(Term.Name("sep")), None)
    )), None)) => q"endBy(p, sep)"
}
```

This rule is likely to be much more useful in practice than *Simplify Parsers*, serving as a pedagogical tool to guide users towards using the library's higher-level combinators.

Summary

This chapter demonstrates how the left-recursion factoring rule is made feasible with the simplification machinery in the new `Parser` AST. It also introduces two rules, *Simplify Parsers* and *Avoid Parser Redefinitions*, that are trivial to implement on top of the new `Parser` AST, showcasing the power of the new high-level AST in expressing transformations in a more maintainable and readable way. The improved `Parser` AST machinery makes it easy to extend `parsley-garnish` with more complex rules on top of a parser abstraction, without having to worry about manually simplifying its output. Future work could explore more complex rules, such as detecting ambiguous parsers or conversely, overuse of the `atomic` combinator.