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 `implicitSymbol` 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. For example, consider the following code snippet:

```
val p = 'g' ~> "arnish"
p.parse("garnish")
// [error] type mismatch;
//   found   : String("arnish")
//   required: Parsley[?]
//  Note that implicit conversions are not applicable because they are ambiguous:
```

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

```
//   both method stringLift in object character of type (str: String): Parsley[String]
//   and method implicitSymbol in class ImplicitSymbol of type (s: String): Parsley[Unit]
//   are possible conversion functions from String("arnish") to Parsley[?]
//    val p = 'g' ~> "arnish"
//                   ^^^^^^^^
```

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

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

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.

1.1.1 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 clause of the form `import parsley.syntax.character.xxx`, where `xxx` is either a wildcard import or specifically contains the importee `stringLift`, indicates the *Implicit Conversions* pattern on strings.
- The import path for the *Implicit Lexer* pattern is subject to how the user names their lexer object, so determining this is much more heuristic-based. 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:

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

```scala
object parser {
  import parsley.syntax.character.stringLift
  import lexer.implicits._
  val p = "garnis" <~ 'h'
}

import parsley.token.Lexer
object lexer {
  import parsley.token.descriptions.LexicalDesc
  private val lexer = new Lexer(LexicalDesc.plain)
  val implicits = lexer.lexeme.symbol.implicits
}
```

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

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

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

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

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

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. 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.

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

With this, fig. 1.2 shows the core logic of the rule, implemented as a *tail-recursive* visit over the list of imports, which 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 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.

1.2.1 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:

```
def visit(unvisited: List[Import], visited: List[Import], patches: Patch): Patch = {
  unvisited 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 = visited.filter(currentScope.isWithinScope(_.parent.get))
      val currentInScope = alreadyInScope :+ curr
      val clashingImports = getClashingImports(currentInScope)

      if (getClashingImports(alreadyInScope).isEmpty && clashingImports.nonEmpty) {
        visit(remaining, currentInScope, patches +
          Patch.lint(AmbiguousImplicitConversionsLint(curr, clashingImports)))
      } else {
        // Continue traversal, discarding imports that have fallen out of scope
        visit(remaining, currentInScope, patches)
      }
    case Nil => patches
  }
}
```

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

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

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

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

```
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.