Chapter 1
Idiomatic Implicits Usage

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

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

1.1 Ambiguous Implicit Conversions

1.1.1 Problem

Unfortunately, implicit conversions are a double-edged sword. By their very nature, they can obscure the flow of the program, making it difficult to understand what code is doing and potentially hiding side effects or costly operations. A downside particularly relevant to Parsley is that implicit conversions often lead to confusing error diagnostics when the compiler is unable to resolve them.

One common issue arises from ambiguous implicits when there are multiple implicit conversions in scope. Parsley provides `stringLift` and `charLift` combinators in the `parsley.syntax.character` package for the *Implicit Conversions* pattern, and exposes an `implicitSymbol` combinator for lexers to use in the *Implicit Lexer* pattern. The two implicit conversions cannot be used in conjunction: the *Implicit Lexer* pattern is a specialisation of the former, so `implicitSymbol` is meant to be a *replacement* for `stringLift` For novice users, this may not be immediately apparent and it is easy to accidentally bring both sets of these implicits into scope; anecdotally, this issue has been encountered by a number of WACC students at Imperial.

For example, consider the following code snippet:

```
val p = 'g' ~> "arnish"
p.parse("garnish")
// [error] type mismatch;
//   found   : String("arnish")
//   required: Parsley[?]
//  Note that implicit conversions are not applicable because they are ambiguous:
//   both method stringLift in object character of type (str: String): Parsley[String]
//   and method implicitSymbol in class ImplicitSymbol of type (s: String): Parsley[Unit]
//   are possible conversion functions from String("arnish") to Parsley[?]
//    val p = 'g' ~> "arnish"
//                   ^^^^^^^^
```

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

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

1.1.2 Solution

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

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

Unfortunately, Scalafix restricts usage to only syntactic rules on the bare AST or semantic rules that operate fully post-compilation. Since the ambiguous implicit conversions will cause compilation failures, this lint must be implemented as a syntactic rule. Consequently, the solution takes a different approach: estimating the presence of clashing implicits by examining their import statements within each scope.

1.1.3 Example

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

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

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

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

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

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

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

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

1.1.4 Implementation

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

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

Unlike Java, Scala allows import statements to appear anywhere in source files, not just at the top of the file. This allows imports to be lexically scoped, limiting their visibility to a single class, object, or function. Additionally, Scala scopes imports in a top-down order within the source file, so imports cannot be referenced at any point above their definition.

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

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

1.2 Remove Explicit Usage of Implicit Conversions