Chapter 1
Background

1.1 Linters

Linting is the process of analysing source code to identify and report issues related to coding style and potential logical errors. The term originates from the `lint` program [Johnson 1978], which examined C source code for bugs, as well as wasteful code patterns that may be legal but error-prone. The tool was also utilised to enforce portability restrictions which aided users in writing portable code that could be compiled on multiple platforms. Since the release of `lint`, many linting tools, known as **linters**, have been developed for a wide range of programming languages.

Nowadays, many linters can be integrated into IDEs, where code analysis performed by the linter is run incrementally in the background. Any violations found by the linter are displayed directly in the editor as warnings or errors at the relevant locations in the source code. This brings early, real-time feedback to the programmer, allowing them to address issues as they write code, with minimal interruption to their development workflow. Linters can also be integrated as part of the code review process, or into continuous integration (CI) pipelines to ensure that code adheres to a set of standards before being merged into the main codebase.

Although the traditional definition for linting is concerned only with *detecting* issues in code, modern linters have broadened their scope significantly. In addition to detecting issues, many linters provide *auto-fix* capabilities to correct issues by automatically rewriting the offending code snippets. This feature is often integrated into IDEs as well: the popular Language Server Protocol for defining IDE features enables these auto-fix features via *code actions* [Gunasinghe and Marcus 2022]. When a section of code is highlighted by a linter warning, a user can apply a code action to automatically fix the issue with a single click.

1.1.1 Categories of Lint Rules

Many linters are configurable with a set of *rules*, which specify the categories of issues that the linter should detect. These rules can be enabled or disabled by users, allowing them to customise the linter to their needs. Rules are usually grouped by purpose: some rules are concerned with simply improving code style, while others are concerned with detecting suspicious code patterns indicative of potential bugs.

Style checks and code quality

Linters can suggest opportunities to improve code by utilising language features in a more idiomatic manner. Snippets of code that violate these stylistic rules are not necessarily incorrect, but should be fixed as they may be harder to read or maintain in the long term. Furthermore, many idiomatic practices exist to avoid common pitfalls that could lead to unintended behaviour. By highlighting good practices, linters can help users avoid these common mistakes that may cause bugs. For example, *ESLint*[1], one of the most popular JavaScript linters, warns against common JavaScript pitfalls such as using the regular equality operator `==` instead of its type-safe alternative `===`.

A well-designed linter can help programmers learn about useful language constructs by suggesting them in the context of their code, aiding them in adhering to best practices and common style conventions. This category of rules is therefore especially helpful as an educational tool for new users of a language, who may be unaware of these idioms. For example, the *Clippy*[2] linter for Rust [Li et al. 2023] categorises a collection of

[1] https://eslint.org/docs/latest/rules/
[2] https://doc.rust-lang.org/clippy/

rules as `clippy::complexity` rules to detect code that does something simple in a complex way and suggests a simpler alternative. Fig. 1.1 provides an example of a similar rule in Haskell, from the *HLint*[3] linter. The rule suggests to rewrite the function into an equivalent but more concise form via η-reduction, presented to the user as a code action that can be applied automatically.

```
foo xs = map (+1) xs
```

(a) A Haskell function `foo`, which can be made more concise using η-reduction.

```
Eta reduce
Found:
  foo xs = map (+ 1) xs
Why not:
  foo = map (+ 1)
hlint(refact:Eta reduce)
```

(b) The linter warning shown for `foo`.

Fig. 1.1: An example of a warning from the Haskell linter `hlint`, suggesting a fix that a user can choose to automatically apply.

Domain-specific idioms A library or especially an embedded DSL may require a particular style of usage that is different from the host language [Hora et al. 2012]. The majority of linters are designed for general-purpose application domains, so they are unlikely to detect issues specific to a more specialised domain. Therefore, linters may be developed for specific libraries or DSLs, with their own set of domain-specific rules. In this case, the accompanying linter can benefit users and improve developer productivity in a similar manner to general-purpose linters: common misuses can be detected and sometimes automatically fixed, and users can be directed to relevant documentation to learn more about correct usage. For instance, the *xUnit.net* testing framework for C# is accompanied by the `xunit.analyzers`[4] package which provides linting rules to enforce best practices specific to *xUnit*.

Code smells and opportunities for refactoring

Code refactoring is a well-established practice in software development. In his influential book *Refactoring: Improving the Design of Existing Code* [Fowler 2018], Fowler defines **refactoring** as "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure". Refactoring may be employed to eliminate **code smells**, which are surface indications that could indicate deeper problems in the system. Code smells are not necessarily problematic on their own, however, they may lead to issues such as bugs or poor maintainability if left unchecked. They are conceptually similar to the stylistic issues mentioned earlier, however they may encompass higher-level structural and design-based problems that are not easily fixed by simple stylistic changes. Examples of code smells include duplicated code, which can be hard to update without introducing bugs, and long methods, which can be difficult to understand and maintain. Therefore, it is often productive to refactor code to eliminate code smells, even if the code is still correct and functional.

Certain linting rules can aid in the refactoring process by broadly identifying code smells and candidate areas for refactoring, suggesting appropriate actions that the user can take. As an example, a linter may detect a fragment of code that is repeated in multiple places: this is a code smell, as discussed previously. The linter may then suggest a code action to automatically apply an *Extract Method* [Fowler 2018] refactoring to avoid code

[3] https://hackage.haskell.org/package/hlint
[4] https://github.com/xunit/xunit.analyzers

duplication: fig. 1.2 demonstrates how this automatic refactoring process can be performed in the IntelliJ IDEA[5] IDE.

Likely bugs or errors

In some cases, linters may be able to detect with relatively high confidence that a piece of code is likely to contain a bug. Rules that detect more egregious issues like this are helpful for even experienced users to avoid common pitfalls. For example, *Clippy* has `clippy::correctness` rules which identify code very likely to be incorrect or useless, and `clippy::suspicious` rules for code that is *very sus* but could be a false positive. *ESLint* also provides several rules to warn against code patterns that are likely to cause runtime errors, such as re-assigning a `const` variable.

Again, linters may attempt to provide code fixes for these issues where possible. However, these issues are usually more complex, which may limit the effectiveness or usefulness of auto-fixes: in the case of a suspicious code pattern, the programmer's intent may not be clear, causing the linter to suggest a fix that does not align with the user's expectations.

1.1.2 Implementing Linters

Static vs Dynamic Analysis

Static analysis analyses source code to extract information about its behaviour without executing the program itself. This is in contrast to *dynamic analysis*, which is performed on programs as they run to observe their actual runtime behaviour. Both techniques have their own advantages and disadvantages, and the choice between them depends on the desired properties of a tool. Dynamic analysers are *precise* in that they only observe what a program can actually do, but requires a whole system that can be executed. On the other hand, static analysers can be more lightweight and can be applied to incomplete systems, since they do not require the program to be run. However, the trade-off is that static analysis is generally more *imprecise* since it must approximate the possible behaviours of the program.

Linters should be fast and lightweight, as it is desirable for them to provide feedback to the user in real-time within the IDE. Furthermore, it is desirable for them to be able to provide warnings on incomplete programs and analyse even just fragments of code for suspicious code patterns. For these reasons, the vast majority of linters are static analysis tools.

Metaprogramming

Linters that provide code fixes are *metaprograms*: they are programs that manipulate programs. Auto-fixes and refactorings are performed as source-to-source transformations, which may be implemented as simple text-based replacements or more robust rewrite rules that operate on the abstract syntax tree (AST) of the source code.

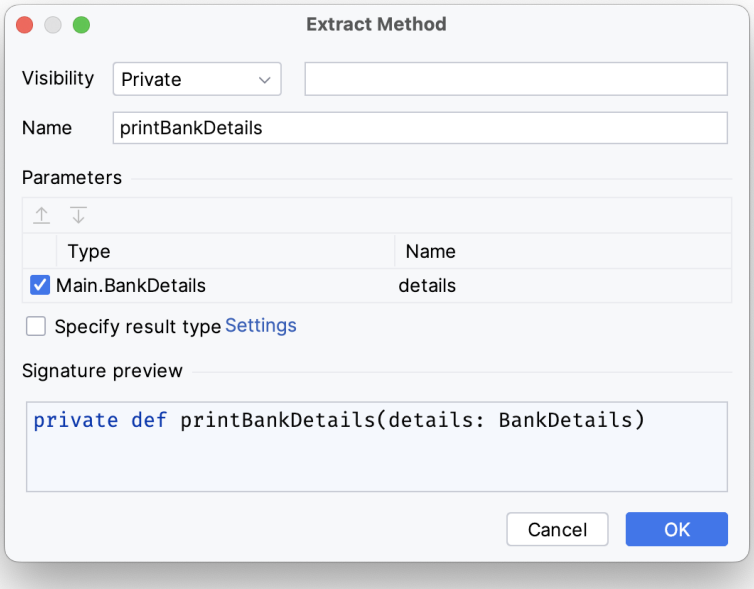
Analyses over ASTs are typically semantic-aware, respecting scoping rules and type information to provide more accurate code fixes. This semantic information is often provided by the compiler – some linters opt to run during compilation as a compiler plugin, while others run as a separate tool after compilation. Lint-on-compile and lint-after-compile tools both have their own advantages and disadvantages, and the choice between them depends on the use case [Sadowski et al. 2018].

Code-Fixing Approaches

[5] https://www.jetbrains.com/idea/

```scala
object Main {
  def main(args: Array[String]): Unit = {
    val bankDetails = getBankDetails()
    println(s"Account name: ${bankDetails.name}")
    println(s"Account balance: ${bankDetails.balance}")
  }
}
```

(a) A snippet of Scala code. A user may wish to extract the highlighted lines into a separate function.



(b) When a user selects the highlighted lines from fig. 1.2a in IntelliJ IDEA, choosing the *Extract Method* refactoring will open this dialogue to preview changes before applying them.

```scala
object Main {
  def main(args: Array[String]): Unit = {
    val bankDetails = getBankDetails()
    printBankDetails(bankDetails)
  }

  private def printBankDetails(details: BankDetails): Unit = {
    println(s"Account name: ${details.name}")
    println(s"Account balance: ${details.balance}")
  }
}
```

(c) The result of applying the *Extract Method* refactoring using the chosen parameters in fig. 1.2b.

Fig. 1.2: An example of the *Extract Method* refactoring in IntelliJ IDEA.

Our approach is formal methods using domain-specific knowledge of parser laws Not data-driven or ML/LLM approach

1.2 Writing Linters for Scala

DSL linting is hard but luckily Parsley is an eDSL so we can just use Scala metaprogramming utilities

1.2.1 Choice of Tooling

The goal of `parsley-garnish` is to provide linting and refactoring capabilities for the `parsley` parser combinator library. Since `parsley` is a Scala library, this project must be implemented using a tool capable of statically analysing Scala code. This section will therefore discuss and evaluate the choices available for implementing `parsley-garnish`.

Scala compiler plugins The most powerful approach would be to implement `parsley-garnish` as a compiler plugin [Pickering, Wu, and Németh 2019]. Using the low-level compiler API, it is possible to perform arbitrary code transformations at any step of the compilation process. Compiler plugins therefore offer full freedom to extend the Scala compiler with extra functionality, such as extra passes for code analysis and emitting lint warnings as diagnostics or even compiler errors.

However, this approach has several drawbacks. Firstly, compiler plugins are tightly coupled with the compiler itself, and therefore not portable across major compiler versions. For instance, plugins written for the Scala 3 compiler, known as `dotty`, are completely incompatible with Scala 2 plugins [LAMP/EPFL 2022]. Additionally, developing compiler plugins requires a deep understanding of arcane and poorly documented compiler internals. Exposing the full compiler API permits unsafe operations that may violate undocumented invariants assumed by the compiler, leading to exceptions during compilation or even malformed bytecode [Sherwany, Zaza, and Nystrom 2015]. The lack of higher-level abstractions also makes it difficult to implement even trivial tasks such as renaming a field.

For these reasons, it would be preferable to explore other tools that may use compiler plugins themselves but provide a higher-level interface for implementing code analysis and transformations.

Scalameta *Scalameta*[6] is a metaprogramming framework for Scala that provides a unified interface for performing common metaprogramming tasks. Many Scala tools are built with Scalameta, most notably the *Metals*[7] LSP server for Scala. It provides a high-level syntactic API for transforming and pretty-printing Scala source code, as well as a semantic API providing access to semantic information such as type inference and name resolution. Scalameta builds upon the earlier `scala.reflect` metaprogramming framework, unifying runtime and compile-time metaprogramming under a single API. Previously under `scala.reflect`, source code was parsed into lossy trees that discarded syntactic *trivia* such as comments and whitespace [Burmako 2017]. On the other hand, Scalameta trees are fully lossless and preserve all syntactic details: a crucial feature for code transformations and refactorings that must preserve formatting details.

Scalameta's semantic API is powered by *SemanticDB*, a compiler-agnostic data model for semantic information in Scala programs. This allows Scalameta to extract semantic information via compiler plugins that emit data in the SemanticDB format. Thus, Scalameta can work with any compiler that supports SemanticDB, rather than being tied to a specific compiler implementation.

[6] https://scalameta.org/
[7] https://scalameta.org/metals/

Since Scalameta provides a high-level interface for manipulating syntactic and semantic information, it is a promising choice for this project. Being able to access semantic information is especially helpful for implementing more complex code analyses. However, Scalameta's primary focus is on providing a general metaprogramming framework and therefore lacks API support specifically for implementing linting and refactoring rules. For example, the Scalameta tree transformation utilities do not fully preserve formatting details when pretty-printed, despite the underlying trees containing this information.

Scalafix *Scalafix*[8] is a code linting and rewriting tool built on top of Scalameta. It provides a higher-level API over Scalameta's tree transformation capabilities, specifically designed for implementing fine-grained code transformations that preserve the original formatting of the source file. Scalafix provides a framework for implementing linting rules to emit diagnostic warnings, as well as rewrite rules to perform automated code transformations [Geirsson 2017]. Since it is built on Scalameta, a major advantage of Scalafix is that it is also compiler-agnostic and could be integrated into any IDE if a plugin is developed for it.

Originally, Scalafix was designed to help automate the process of migrating code from Scala 2 to 3, which involved many breaking changes to the language [Geirsson 2016]. However, Scalafix has since evolved into a general-purpose tool for implementing generic code transformations and analyses. Scalafix rules can be either *syntactic* or *semantic* depending on whether they require semantic information, such as type signatures, to perform their analysis [Scala Center 2024]. Syntactic rules are faster to run, operating purely on the AST without needing to invoke compilation, but are more limited in the analyses they can perform. Semantic rules, on the other hand, are more complicated to run: they require augmenting the Scala compiler with the SemanticDB compiler plugin to provide Scalafix with the required semantic information. They are also slower as a result, since they require the source file to be compiled.

A drawback of Scalafix is that it is primarily a command-line tool, and currently does not provide an interactive interface within any code editor. Ideally, users should be able to see inline diagnostics, as well as preview and select auto-fixes interactively like in fig. 1.2. A Metals integration would be the most favourable outcome, as they already share a Scalameta backend, and it would allow Scalafix to be integrated into any code editor supporting the LSP protocol. However, at the time of writing, Scalafix only has basic integrations with Metals to run rules, but no interactive in-editor integrations [Jaglin 2024]. This feature request is an open issue on GitHub[9], but no timeline has been provided for its implementation.

Nonetheless, Scalafix emerges as the most favourable choice for implementing `parsley-garnish`. It provides high-level APIs specifically for implementing linting and rewrite rules without necessitating extensive knowledge of compiler internals. Scalafix is also being actively maintained, with good basic documentation and a growing number of examples of usage in the wild.

Other tools considered The main alternate contender to Scalafix is the IntelliJ Scala Plugin[10]. However, while the plugin offers superior interactive usage within the IntelliJ IDEA IDE, it is tied to the IntelliJ Scala compiler and therefore not portable across other compilers. To maintain flexibility and not tie `parsley-garnish` to a particular compiler or code editor, Scalafix is a preferable option. Furthermore, documentation is less clear on how to write a Scala plugin for IntelliJ compared to the Scalafix documentation.

WartRemover[11] is a linter implemented as a compiler plugin, with support for writing custom rules. However, it only can emit diagnostics and does not support code fixes, making it unsuitable for `parsley-garnish`'s goals.

ScalaStyle[12] is primarily a style checker which also supports custom rules. However, it is only able to perform

[8] https://scalacenter.github.io/scalafix/
[9] https://github.com/scalameta/metals-feature-requests/issues/230
[10] https://github.com/JetBrains/intellij-scala
[11] https://www.wartremover.org/
[12] http://www.scalastyle.org/

syntactic analyses and does not have access to semantic information, restricting the types of analyses it can perform.

1.2.2 A Crash Course on Scalafix

This section briefly explains how Scalafix rules are written, providing the Scalafix-specific terminology needed to understand the implementation details of `parsley-garnish`.

Rules

Scalafix rules are implemented as generic top-to-bottom traversals through the Scala AST, represented as a Scalameta `Tree` data structure. This is achieved with a collection-like API, exposing these operations in the style of standard Scala collection methods such as `traverse` and `collect`. During traversal, side effects are generated using the `Patch` data structure, which represents either a lint diagnostic or a code rewrite. Rules are applied file-by-file, and the patches generated by each rule are collected and applied by Scalafix as a batch rewrite after all rules have finished running.

Fig. 1.3 showcases the basic structure of a simple syntactic rule:

- `doc.tree.collect` performs the tree traversal, where AST nodes of interest can be visited via pattern matching.
- `case v: Defn.Val if !v.mods.exists(_.is[Mod.Lazy])` visits a `val` definition node that is not already marked as `lazy`.
- `Patch.addLeft(v, "lazy ")` describes a patch to add the `lazy` modifier to the left side of the node that was matched on.

```
class MakeValsLazy extends SyntacticRule("MakeValsLazy") {
  override def fix(implicit doc: SyntacticDocument): Patch = {
    doc.tree.collect {
      case v: Defn.Val if !v.mods.exists(_.is[Mod.Lazy]) => Patch.addLeft(v, "lazy ")
    }.asPatch
  }
}
```

Fig. 1.3: A syntactic rule that makes all `val` definitions lazy.

Safer Patches Using Quasiquotes

A careful reader may notice from fig. 1.3 that the `Patch` rewrite method receives a raw string value, which seems unsafe and could potentially lead to malformed code. That careful reader would indeed be correct: Scalafix provides no guarantees that the output of a patch is a well-formed program, and it is the rule author's responsibility to ensure so.

The approach taken by `parsley-garnish` is to represent intended rewrites as Scalameta `Tree` objects, and only convert them to strings immediately before applying the patch. Fig. 1.4 shows how *quasiquotes* [Shabalin, Burmako, and Odersky 2013] can be used as syntactic sugar to construct trees in a convenient manner:

- `q"..."` is the quasiquote's string interpolation syntax to build a `Term` node, which is a subclass of `Tree`.

- `${...}` is the *splice* or *unquote* operation to insert a term into the constructed tree. Ordinarily, spliced values need to be an instance of a `Tree`, but simple values such as integers (in this case) can be automatically lifted into the appropriate tree node.
- `Tree.structure` represents the resulting `Tree` data structure as a string, useful for debugging.
- `Tree.syntax` pretty-prints the tree back into its source code representation, which can then be fed into a `Patch`.

Quasiquotes expand at compile-time into `Tree` constructors, so they do not preserve formatting details. This turns out to be very convenient for pretty-printing purposes: often, spliced values may contain superfluous parentheses, which are automatically removed (as seen in the final example within fig. 1.4).

```
val x = 1
val tree = q"$x + 2"

println(tree.structure)
// Term.ApplyInfix(
//   Lit.Int(1),
//   Term.Name("+"),
//   Type.ArgClause(Nil),
//   Term.ArgClause(List(Lit.Int(2)), None)
// )

println(tree.syntax)
// 1 + 2
println(q"($tree)".syntax)
// 1 + 2
```

Fig. 1.4: Using Scalameta quasiquotes to construct a tree.

By using quasiquotes instead of raw strings, the syntactic correctness of a rewrite is guaranteed. However, this safety is not foolproof: quasiquotes are not guaranteed to be well-typed or well-scoped, so the rewritten program still might not be able to compile. They are also not *hygienic*: generated code will not be able to avoid name clashes with regular code [Burmako 2017]. The lack of hygiene can cause issues with variable capture, allowing a variable to be unintentionally shadowed by a generated variable. Again, it is the rule author's responsibility to ensure that variable capture does not occur: **??** in part discusses how this is handled in `parsley-garnish`.

Semantic Information

Semantic rules are implemented in the same manner as fig. 1.3, but instead take the more powerful `SemanticDocument` as an implicit parameter. This allows rule authors to query AST nodes for their associated symbols: textual identifiers that represent definitions in the Scala program. Although symbols are not guaranteed to be unique outside the document they are defined in [Scalameta 2023], within the context of Scalafix rule authoring, they can be treated as unique since rules are applied per-file.

The following methods are the most important for querying the semantic information associated with a tree node:

- `Tree.symbol` retrieves the symbol associated with a node.
- `Symbol.info` queries a symbol for any `SymbolInformation` attached to it, including metadata such as its type signature.

- `Tree.synthetics` reveals extra compiler-generated tree nodes not present in the original source code, such as implicit conversions or inferred `.apply` methods.

Additionally, the `SymbolMatcher` trait provides a convenient way to create predicates that match specific symbols. Fig. 1.5 demonstrates how symbol matching can be used to detect integer addition operations, emitting a diagnostic warning when found. The `intAdd` symbol matcher only matches the `+` method on integers, so the rule does not emit a diagnostic for addition operations on other types, as seen in fig. 1.5b.

```scala
class IdentifyIntAddition extends SemanticRule("IdentifyIntAddition") {
  val intAdd = SymbolMatcher.normalized("scala.Int.`+`")

  override def fix(implicit doc: SemanticDocument): Patch = doc.tree.collect {
    case Term.ApplyInfix(_, intAdd(node), _, _) =>
      Patch.lint(IntAdditionLint(node))
  }.asPatch
}

case class IntAdditionLint(tree: Tree) extends Diagnostic {
  override def position: Position = tree.pos
  override def severity: LintSeverity = LintSeverity.Info
  override def message: String = "Found an addition operation on integers"
}
```

(a) A semantic rule that generates lint diagnostics *only* on integer addition operations.

```scala
object Main {
  val stringAdd = "parsley" + "garnish"
  val intAdd = 1 + 2
  val floatAdd = 1.0 + 2.0
}

// Main.scala:3:18: info: [IdentifyIntAddition]:
// Found an addition operation on integers
//   val intAdd = 1 + 2
//                  ^
```

(b) An example program showcasing the rule in fig. 1.5a.

Fig. 1.5: An example showing how semantic rules are semantic-aware on overloaded method names.

1.3 Parser Combinators

Parsing is the process of extracting structured information from a flat, unstructured representation of the data. Parsers are programs that perform this process, using a specified grammar to determine the structure of the data. They are utilised in a variety of applications such as compilers, interpreters, and processing of data storage formats such as JSON and XML.

Traditionally, parsers have either been written by hand or by using parser generator frameworks such as ANTLR [Parr 2013]. Hand-rolling a parser is a tedious process, requiring the programmer to manually implement the parsing algorithm for the grammar. However, this approach is the most powerful and flexible and can provide excellent performance. Alternatively, parser generators lift the burden of implementing the parsing algorithm,

instead requiring the programmer to specify the grammar in the format of a domain-specific language (DSL) similar to a high-level grammar. The grammar is then compiled by the parser generator tool to produce a parser in a target language. This approach is less flexible but can be more convenient and less error-prone.

Parser combinators [Hutton 1992], which stem from a functional programming background, are a middle ground between the two approaches. They take the form of an embedded DSL written directly in a general-purpose language, rather than the parser generator approach where the DSL is a separate language. With a parser generator, the provided DSL is often limited in its expressiveness. This is not the case with parser combinators, as the full power of the host language is available to the programmer. This approach also reduces boilerplate code: for example, the programmer does not need to convert between the AST produced by the parser generator and their own AST.

A downside of parser combinators, however, is that they are unstandardised compared to parser generators. Across different implementations, parser combinator APIs can vary significantly, making it difficult to transfer knowledge between different libraries. Experienced users of parser combinators may approach a new library with prior knowledge of general concepts but may have misconceptions about the specifics of the API which can lead to confusion and frustration. This is another motivating reason for the development of `parsley-garnish`, to lower the barrier of entry for new users of the `parsley` library.

1.3.1 Parsley

TODO: proper, worked example showcasing relevant design patterns and stuff which will be picked up by the linter

`Parsley` [Willis and Wu 2018] is a parser combinator library for Scala that provides an API inspired by the `parsec` [Leijen and Meijer 2001] style of parser combinators. This section provides an illustrative example of a simple expression parser to demonstrate what a parser written in `parsley` looks like.

Consider the EBNF grammar for a simple expression language shown in fig. 1.6a. The parser in fig. 1.7 will parse an expression into the AST represented by the Scala datatype in fig. 1.6b.

Notice how the parser closely resembles the high-level EBNF grammar. The main differences of note include the use of:

- `map` to transform the result of a parser to help construct tree nodes consisting of a single value.
- `zipped` to combine the results of two parsers to help construct tree nodes consisting of multiple values.
- `<~` and `~>` operators to guide the direction of parsers.

Except for the possibly cryptic-looking implementation of `num` to parse a series of digits into an integer, the parser is relatively straightforward to understand.

Types of left recursion

Willis and Wu [2022] describe several design patterns for writing maintainable parsers using parser combinators in Scala. They identified common problems and anti-patterns in parser design, and proposed solutions in the form of design patterns. This provides a guideline for writing idiomatic `parsley` code for practical parser design, which enables opportunities for the development of linting and refactoring rules.

This thesis hopes to explore how these common problems can be formalised into code smells and suspicious code patterns that can be automatically detected using linting rules. Some of the design patterns are also theoretically amenable to automated refactoring, which we hope to explore and implement in `parsley-garnish`.

1.3.2 Parsley Haskell

```
ident ::= "x" | "y" | "z"
num ::= digit+
expr ::= factor "+" expr
factor ::= atom "*" factor
atom ::= ident | num | "(" expr ")"
```

(a) The grammar in EBNF.

```scala
sealed trait Expr
case class Ident(name: String) extends Expr
case class Num(x: Int) extends Expr
case class Add(x: Expr, y: Expr) extends Expr
case class Mul(x: Expr, y: Expr) extends Expr
```

(b) The Scala AST to parse into.

Fig. 1.6: The grammar and AST for our simple expression language.

```scala
val ident = "x" | "y" | "z"
val num: Parsley[Int] = digit.foldLeft1(0)((n, d) => n * 10 + d.asDigit)

lazy val expr: Parsley[Expr] = (factor, "+" ~> expr).zipped(Add)
lazy val factor: Parsley[Expr] = (atom, "*" ~> factor).zipped(Mul)
lazy val atom: Parsley[Expr]
  = ident.map(Ident) | num.map(Num) | "(" ~> expr <~ ")"
```

Fig. 1.7: A parser for our simple expression language.

TODO: Should I introduce the idea that haskell parsley uses staged metaprogramming for optimisation purposes here?