Chapter 1

Background

1.1 Linting and Static Analysis

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.

Linters and related static analysis tools are increasingly becoming more important in modern software development, as modern code continues to become more complex and difficult to reason about. Industry leaders, such as Google [Sadowski et al. 2018] and Meta/Facebook [Calcagno et al. 2015], have embraced these tools as integral components of their software development workflows. The use of automated tools to detect potential issues is not only faster but in some cases more effective than human review, saving developer time and reducing error rates in the development process.

1.1.1 Types of Issues Detected by Linters

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 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*.

[1] https://eslint.org/docs/latest/rules/
[2] https://doc.rust-lang.org/clippy/
[3] https://hackage.haskell.org/package/hlint
[4] https://github.com/xunit/xunit.analyzers

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

Possible bugs or errors

Linters may also directly attempt to detect more serious issues in code, such as possible logic errors. This can be helpful for even experienced users to avoid common pitfalls. For example, *Clippy* has `clippy::suspicious` and `clippy::correctness` rule categories to identify code that is very likely to be incorrect or useless. *ESLint* 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 auto-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

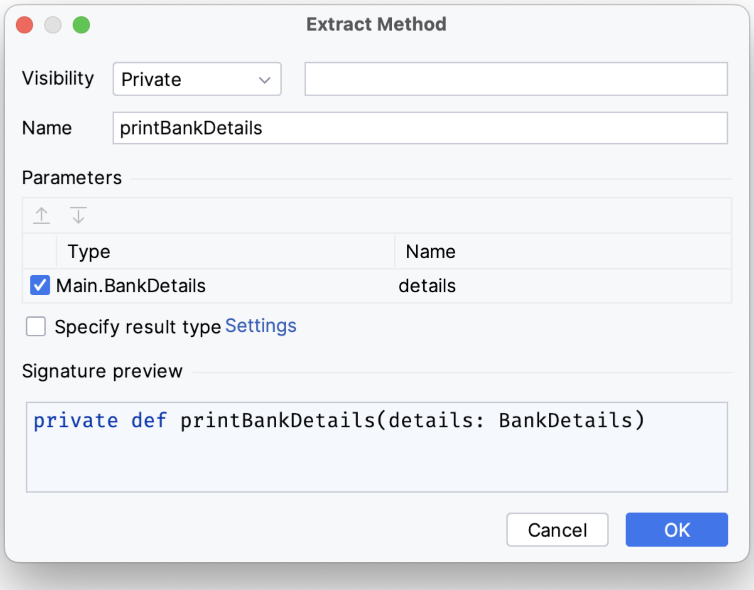
This is the old static analysis section - rework this

The vast majority of linters are implemented using static program analysis, analysing 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 are run. Although dynamic analysis can yield more precise results as it observes the actual runtime behaviour of the program, it is also more heavyweight as a result.

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

```
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.



Extract Method

Visibility: Private

Name: printBankDetails

Parameters

Type	Name
☑ Main.BankDetails	details

□ Specify result type Settings

Signature preview

```
private def printBankDetails(details: BankDetails)
```

(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.

```
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.

Given their need to be scalable and efficient enough to handle entire codebases, linters are designed to be fast and lightweight. In most practical cases, therefore, dynamic analysis is too slow and resource-intensive to be used for the purpose of linting. However, dynamic languages such as JavaScript present challenges for static analysis due to their dynamic runtime nature. This has led to the exploration of linters that incorporate dynamic analysis techniques to identify issues that static methods cannot detect [Gong et al. 2015]. Despite these developments, state-of-the-art JavaScript linters utilised in industry, such as *ESLint*, still solely rely on static analysis.

* parsley is a scala library * scala is a static language, strongly typed * this section will therefore focus on the traditional static analysis approach, since dynamic analysis is not as relevant here

https://scalacenter.github.io/scalafix/docs/users/related-projects.html Lint on compile vs lint after compile?

AST stuff

* how does HLint work? seems like its sometimes prone to false positives

Static analysis tools can reason about how to safely refactor code in an automated manner, performing refactorings as source-to-source transformations. These transformations 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.

These tools can perform a variety of tasks, ranging from detecting possible bugs [Johnson 1978; Hovemeyer and Pugh 2004] to formal software verification of program properties [Blanchet et al. 2003].

1.2 Static Analysis 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. 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. make this clearer about the unification of runtime and compile-time metaprogramming? scalameta does more than scala-reflect Scalameta is the successor of the earlier `scala.reflect` metaprogramming framework, which parsed source code into lossy trees that discarded syntactic information such as comments and whitespace [Burmako 2017]. this is called hygiene right? On the other hand, Scalameta trees are lossless and preserve all syntactic details, a key feature that allows code transformations and refactorings to 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.

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 preserve formatting details when pretty-printed, despite the underlying trees containing this information.

Scalafix *Scalafix*[7] is a refactoring and linting tool built on top of Scalameta. It specifically provides an API for implementing fine-grained code transformations that preserve comments and formatting details. Scalafix provides a framework for implementing linting rules to emit 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 code transformations and analyses, utilising the powerful syntactic and semantic APIs provided by Scalameta. Scalafix rules can be either syntactic or semantic, depending on whether they require semantic information to perform their analysis. Syntactic rules are faster to run, since they operate purely on the AST without requiring compilation to extract semantic information, but are more limited in the accuracy of analyses they can perform.

[6] `https://scalameta.org/`
[7] `https://scalacenter.github.io/scalafix/`

Scalafix is growing to become the de-facto modern successor to earlier refactoring tools such as Abide[8] and `scala-refactoring`[9]. `scala-refactoring` used `scala.reflect` to implement code transformations, with much extra work utilising the Scala Presentation Compiler AST to preserve formatting details lost by `scala.reflect`. As a result, maintaining the library became difficult and the project was abandoned in favour of a clean implementation using Scalameta, which was designed in part to address the shortcomings of `scala.reflect`.

A drawback of Scalafix is that it is primarily a command-line tool, and therefore by default does not provide a user-friendly interface for interactive usage. However, this can rectified in the future by integrating Scalafix into the Metals LSP server for Scala, which would allow it to be integrated into any IDE that supports the LSP.

Overall, Scalafix emerges as the most favorable 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 developed and 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 warnings or errors and does not support auto-fixes, making it less suitable for `parsley-garnish`'s goals.

ScalaStyle[12] is primarily a style checker which also supports custom rules. However, it is only able to perform syntactic analyses and does not have access to semantic information, restricting the types of analyses it can perform.

1.2.2 Scalafix

Two categories of scalafix rules: syntactic and semantic. [Scala Center 2024] Syntactic rules don't require compilation, so they are easier to run. However this also means they can only do limited code analysis, as they don't have access to compiler information such as symbols and types. Semantic rules, on the other hand, require compilation and are therefore slower and more complicated to run. However, they are able to perform more advanced code analysis since they have access to the compiler information that syntactic rules lack access to.

Scalafix rules can act like traditional linters, solely emitting diagnostics for issues in the code. They can also apply rewrites to transform code as an automatic fix to the issue.

Internals

Scalafix parses each source file to generate AST once, each rule gets fed this AST later.

[8] https://contributors.scala-lang.org/t/whats-the-status-of-abide/
[9] https://github.com/scala-ide/scala-refactoring
[10] https://github.com/JetBrains/intellij-scala
[11] https://www.wartremover.org/
[12] http://www.scalastyle.org/

The compiler information that semantic rules use is provided by the semanticdb-scalac compiler plugin. This is injected immediately after the typer phase of the scalac Scala compiler, allowing it to harvest and dump semantic information in SemanticDB format. [Scalameta 2023a] Basically a stripped-down version of TASTY (Typed Abstract Syntax Trees)[13] but for both scala 2 and 3. Scalafix then presents a seamless API to implement semantic rules, allowing users to inspect an AST node and any semantic information associated with it.

Implementing Rules

Scalafix rules are metaprograms – they manipulate Scala programs. This is achieved by a generic traversal through the AST, represented as a SemanticDB Tree datatype. Generate side effects with Patches, which represent either lint diagnostics or code rewrites. Rewrite Patches are pure strings, so this isn't the safest way to do things as there are no guarantees that the output is a well-formed program. This shortcoming can be somewhat mitigated by instead representing rewrites as Trees, and only converting them to strings right before applying the Patch. Patches generated by rules get collected, and applied in a batch rewrite at the end

What semantic information is available to the user? Trees may have symbol information affixed to them: global symbols are guaranteed to be unique across all documents processed by semanticDB, but local symbols are not [Scalameta 2023b]. This is not necessarily an issue for us, because scalafix rules are only applied per-file anyways so everything should be unique within the file. Scalafix API provides SymbolMatcher to easily create predicates to match specific symbols during traversal of the AST.

SymbolInformation also provides other metadata, especially useful is its type signature provided from the compiler's type-checking stage. A tree may also contains synthetics data, granting access to implicit parameter application, and surface syntactic sugar added by compiler. SemanticDocument.diagnostics checks contextual (contextualised = we know which parts of the code the warnings is for) compiler warnings.

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

[13] https://docs.scala-lang.org/scala3/guides/tasty-overview.html

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. 1.3a. The parser in fig. 1.4 will parse an expression into the AST represented by the Scala datatype in fig. 1.3b.

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.

1.3.2 Design Patterns for Parsley

(This background section is a work-in-progress, and will likely expand to include more information about the specific problems and design patterns I choose to explore in the project.)

Willis and Wu [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.

```
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. 1.3: 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. 1.4: A parser for our simple expression language.

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.