

Imperial College London

MENG INDIVIDUAL PROJECT: INTERIM REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

parsley-garnish: **A Linter for the parsley Parser Combinator Library**

Author:

Rocco Jiang

Supervisor:

Dr. Jamie Willis

Second Marker:

Dr. Robert Chatley

January 25, 2024

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Project Goals	3
2	Static Analysis Tools	4
2.1	Applications of Static Analysis Tools	4
2.1.1	Automated Refactoring	4
2.1.2	Linting	5
2.2	Static Analysis for Scala	8
2.2.1	Choice of Tooling	8
3	Parser Combinators	12
3.1	Parsley	13
3.2	Design Patterns for Parsley	13
4	Project Plan	15
4.1	Proposed deliverables and milestones	15
4.1.1	Key deliverables	15
4.1.2	Optional deliverables	15
4.2	Current Progress	16
4.3	Project Timeline	16
5	Evaluation Plan	18

Chapter 1

Introduction

1.1 Motivation

Parser combinators [1] are an elegant approach for writing parsers in a manner that remains close to the original grammar specification. Parsley [2] is a parser combinator library implemented as an embedded domain-specific language (DSL) [3] in Scala, with an API inspired by the `parsec` [4] family of libraries in Haskell. However, as with many libraries, there exists a learning curve to utilising `parsley` and parser combinator libraries in an idiomatic manner.

While well-documented, the wealth of information to get started with `parsley` can be overwhelming for users, particularly those new to parser combinators. Furthermore, there exists a number of design patterns [5] for writing maintainable parsers, which even experienced users may be unaware of. A potential solution to this problem is tooling to provide automated code hints, which a user can use during the development cycle to evaluate if their code adheres to best practices.

A number of modern integrated development environments (IDEs) provide code hints to warn programmers about problems in their source code, highlighting offending snippets and suggesting actions to improve suboptimal or incorrect code [6]. Many of these code analysis tools are designed to detect general issues for the host language, rather than specifically for libraries. However, tools may also utilise domain-specific code analyses in order to detect issues specific to a particular system or problem domain [7, 8].

1.2 Project Goals

This project aims to explore the potential of harnessing static code analysis techniques to develop a new tool, `parsley-garnish`, that offers code hints aimed at assisting programmers in writing idiomatic and correct `parsley` code. Additionally, for certain issues that can be automatically fixed, `parsley-garnish` will provide automated actions to resolve the issue. The goal of `parsley-garnish` is to be used as a companion library to `parsley`, in order to improve its ease of adoption and to help users enforce best practices.

Chapter 2

Static Analysis Tools

cite

Static program analysis is the process of automatically analysing source code to extract information about its behavior without executing it, as opposed to dynamic analysis, which is performed on programs as they are run. Static analysis tools can ease the burden of software development by automating tasks that would otherwise require manual effort and meticulous attention to detail. These tools can perform a variety of tasks, ranging from detecting possible bugs [9, 10] to formal software verification of program properties [11].

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 [12] and Meta (formerly Facebook) [13], have embraced static analysis tools as integral components of their software development workflows.

2.1 Applications of Static Analysis Tools

Typically in a software development workflow, multiple static analysis tools are used in conjunction to provide a comprehensive suite of checks. Often, these tools are integrated into an IDE as plugins, allowing them to be consolidated into a single visual interface.

Refactoring and linting are two major functionalities that these tools provide. However, because many of these tools include both capabilities, the distinction between them can be blurry. This section aims to establish definitions for the two terms, which will be used as a basis for subsequent discussion within this thesis.

2.1.1 Automated Refactoring

Code refactoring is a well-established practice in software development. In his influential book *Refactoring: Improving the Design of Existing Code* [14], Fowler defines **refactoring** as “the process of changing a software system in such a way that it does not alter the external behavior 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. 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.

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.

Automated refactoring support is particularly useful for large codebases, where manual refactoring would be tedious and error-prone. Generally, when a user makes use of an automated refactoring tool in an IDE, they will manually identify the snippet of code that they wish to refactor, and then select the appropriate refactoring from a list of available options. Fig. 2.1 presents *Extract Method* [14] in the IntelliJ IDEA¹ IDE, an example of a common refactoring that can be performed automatically on a block code selected by the user.

2.1.2 Linting

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 [9], 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.

Linters are provided as standalone tools separate from a compiler, since their primary goal is to suggest improvements for code readability and maintainability, rather than code optimisations. Modern linters are commonly 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 provides an ergonomic user experience, as the user can see the results of the analysis in real-time as part of the development workflow.

Although the strict definition for linting is concerned with only detecting issues in code, many advanced linters today can also provide auto-fixes for violations which can be corrected automatically by static analysis. This auto-fix capability is often integrated into IDEs as well: the popular Language Server Protocol for defining IDE features enables these auto-fix features via *code actions*. 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.

Many linters are configurable with a set of rules, which specify the types of issues that the linter should detect. These rules can be enabled or disabled by the user, allowing them to customise the linter to their needs. Rules can be categorised by their purpose: some rules are concerned

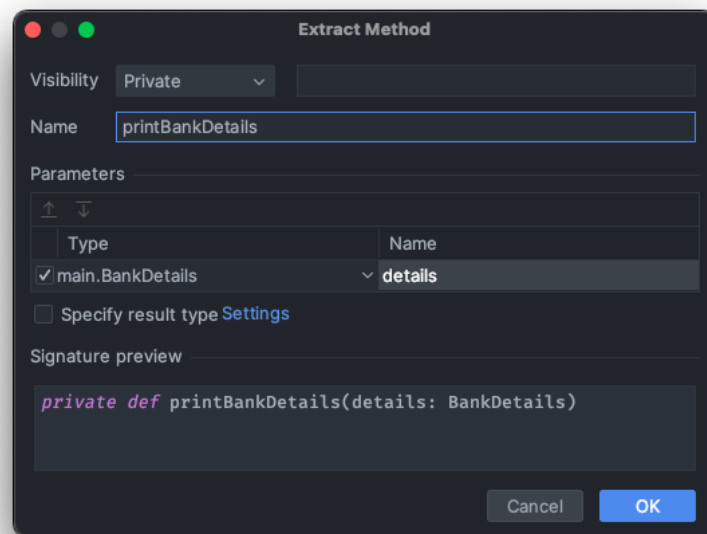
¹<https://www.jetbrains.com/idea/>

```

1      object Main {
2          def main(args: Array[String]): Unit = {
3              val bankDetails = getBankDetails()
4              println(s"Account name: ${bankDetails.name}")
5              println(s"Account balance: ${bankDetails.balance}")
6          }
7      }

```

(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. 2.1a in IntelliJ IDEA, choosing the *Extract Method* refactoring will open this dialogue to preview changes before applying them.

```

1      object Main {
2          def main(args: Array[String]): Unit = {
3              val bankDetails = getBankDetails()
4              printBankDetails(bankDetails)
5          }
6
7          private def printBankDetails(details: BankDetails): Unit = {
8              println(s"Account name: ${details.name}")
9              println(s"Account balance: ${details.balance}")
10         }
11     }

```

(c) The result of applying the *Extract Method* refactoring using the chosen parameters in fig. 2.1b.

Figure 2.1: An example of the *Extract Method* refactoring in IntelliJ IDEA.

with enforcing code style, while others are concerned with detecting code smells or other suspicious code patterns indicative of possible bugs.

Style checking and code appearance

Linters can be configured to enforce a particular style guide defining a set of conventions for how idiomatic code should be written. They can highlight style violations and in basic cases, automatically rewrite code to conform to the correct style. For example, the *Flake8*² linter for Python enforces coding conventions from the PEP 8 style guide . Stylistic rules are especially helpful for large projects with multiple contributors, where a consistent coding style can improve readability and maintainability.

cite

Identifying opportunities for refactoring

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 the *Extract Method* refactoring to avoid code duplication.

Suggesting idiomatic usage

Other rules can suggest opportunities to improve more precise snippets of code by utilising language features in a more idiomatic manner. These rules are especially helpful for new users of a language, who may be unaware of useful language constructs and idioms. For example, the *Clippy*³ linter for Rust 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. 2.2 provides an example of a similar rule in Haskell, from the *HLint*⁴ linter. The rule suggests an η -reduction refactoring, presented to the user as a code action that can be applied automatically.

cite article

Many idiomatic practices exist to avoid common pitfalls that may lead to unintended behaviour. By highlighting good practices, linters can help users avoid these common mistakes that may cause bugs. For example, *ESLint*⁵, 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 `===`.

Linters developed for a specific library provide rules to enforce idiomatic usage specific to the domain of the library. A library or especially an embedded DSL may require a particular style of usage that is different from the host language. Therefore, a regular linter for the host language may not be able to detect issues specific to the library. In this case, an accompanying linter can greatly benefit users: common misuses can be detected and sometimes automatically

²<https://flake8.pycqa.org/en/latest/>

³<https://doc.rust-lang.org/clippy/>

⁴<https://hackage.haskell.org/package/hlint>

⁵<https://eslint.org/docs/latest/rules/>

```
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`.

Figure 2.2: An example of a warning from the Haskell linter `hlint`, suggesting a fix that a user can choose to automatically apply.

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`⁶ package which provides linting rules to enforce best practices specific to xUnit.

Detecting potential bugs

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. 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.

2.2 Static Analysis for Scala

2.2.1 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. 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.

cite

⁶<https://github.com/xunit/xunit.analyzers>

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 . 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 . The lack of higher-level abstractions also makes it difficult to implement even trivial tasks such as renaming a field.

cite

cite

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*⁷ 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. 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 . 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.

cite

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.

cite

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*⁸ 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

⁷<https://scalameta.org/>

⁸<https://scalacenter.github.io/scalafix/>

details. Scalafix provides a framework for implementing linting rules to emit warnings, as well as rewrite rules to perform automated code transformations . 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 . 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.

Scalafix is growing to become the de-facto modern successor to earlier refactoring tools such as Abide⁹ and scala-refactoring¹⁰. 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 be 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 Plugin¹¹. 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.

WartRemover¹² is a linter implemented as a compiler plugin, with support for writing custom

⁹<https://contributors.scala-lang.org/t/whats-the-status-of-abide/>

¹⁰<https://github.com/scala-ide/scala-refactoring>

¹¹<https://github.com/JetBrains/intellij-scala>

¹²<https://www.wartremover.org/>

rules. However, it only can emit warnings or errors and does not support auto-fixes, making it less suitable for parsley-garnish's goals.

ScalaStyle¹³ 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.

¹³<http://www.scalastyle.org/>

Chapter 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 [15]. 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 [1], 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.

3.1 Parsley

Parsley [2] is a parser combinator library for Scala that provides an API inspired by the parsec [4] 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. 3.1a. The parser in fig. 3.2 will parse an expression into the AST represented by the Scala datatype in fig. 3.1b.

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.

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

(a) The grammar in EBNF.

```
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.

Figure 3.1: The grammar and AST for our simple expression language.

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.)

```

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 <~ ")"

```

Figure 3.2: A parser for our simple expression language.

Willis and Wu [5] 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.

Chapter 4

Project Plan

4.1 Proposed deliverables and milestones

This project is rather exploratory in nature, and as such, these milestones are highly subject to change as the project progresses. However, I have attempted to outline some key deliverables which I believe to be achievable, as well as some optional deliverables to explore if time permits.

4.1.1 Key deliverables

- Simple lint rules for non-idiomatic combinator usage which can be simplified with an automated refactoring. For example, `endBy(p, sep)` is the idiomatic equivalent to `many(p <* sep)`.
- Detection of left-recursive parsers, which are a bug pattern since they will cause infinite recursion at runtime. It should be possible to perform a static analysis to produce a code transformation to utilise the `postfix` family of combinators to eliminate left-recursion and resolve the bug.
- Lint rules to detect ambiguity in parsers, which can be a bug pattern since the parser may not behave as expected. It may be possible to provide an automated fix in some cases, but more research has to be done to determine the feasibility of this.

Conversely, it may also be possible to detect overuse of the `atomic` combinator. Users may overuse this combinator to avoid ambiguity, but overuse is a code smell as it can negatively impact error messages reported by the parser.

- Proofs that all code transformations preserve the semantics of the original parser.

4.1.2 Optional deliverables

- Lint rules that warn against side-effecting parsers. `Parsley` employs aggressive parser optimisations which may assume that parsers are pure. The lint rule would remind users to use the `impure` combinator to avoid `parsley` from optimising away any intended behaviour.

- Lint rules to detect when a parser using `chain/infix/postfix` combinators can be rewritten to use a precedence table, and the automated refactoring to accompany this rule. Further lints/refactorings related to precedence tables may also be possible, such as looking at type information to determine the most appropriate shape (Ops/SOps/GOps).
- An editor plugin or extension to provide integration of `parsley-garnish` into the code editor. As explored in the background section, it is possible to provide this integration for Scalafix, but the difficulty of this task is unknown.
- Exploration of possible pedagogical applications of `parsley-garnish`. For example, lint rules may be useful for marking submissions for the Second Year WACC compilers project.

4.2 Current Progress

Most of the work so far has been on research and experimentation with Scalafix. Although there was good documentation to get started, Scalafix is an active project and the documentation is not always fully comprehensive or up-to-date. I still have some concerns about whether the Scalafix API is expressive enough to implement all of the analyses I have in mind, but I will continue to explore this as the project progresses.

From my experimentation, I have a relatively good idea of how to implement most of the “simple” lint rules.

At the time of writing, I have also implemented a proof-of-concept rule for transforming left-recursive parsers, on a small subset of `parsley`’s combinators. This has been encouraging but much work remains to be done to be able to apply the rule on more complex parsers. The current results of the transformations are also sometimes too complex, so I will need to investigate methods such as defunctionalisation and/or alternate methods of representing terms to simplify the output statically. Furthermore, the transformation is likely to be correct but not proven yet.

4.3 Project Timeline

My immediate plan is to continue work on the left-recursion transformation, while also working on the simpler lint rules. I hope to complete the left-recursion rule by early March, although unforeseen difficulties may arise (such as if output simplification proves to be more difficult than expected). If it is not complete or close to complete by then, I will re-evaluate the feasibility of the rest of the milestones and adjust accordingly. I have already communicated to my supervisor that this left-recursion rule will likely be the most involved deliverable of the project, although there is also a chance that many of the techniques employed could be reused for other transformations.

The next two months will be spent exploring other milestones. I plan on approaching this by alternating research and implementation, as I have found that this has been a good way to make progress so far, while also guaranteeing that I will make progress on the final deliverable.

By the beginning of summer term, I would also like to have a very rough draft of the final report completed, based on the work completed so far. This will allow me to communicate with my supervisor on the state of the project, as well as leaving me ample time to edit and refine the report. The remaining milestones to cover will be decided during this meeting; at this point, having implemented a number of rules, I hope to have a much better grasp on how long each deliverable could take. This remaining time will then be divided between writing the report and implementing the agreed-upon milestones.

Chapter 5

Evaluation Plan

A large corpus of student WACC parsers is available to evaluate the real-world applicability of `parsley-garnish`. This can be utilised to assess the tool’s ability to achieve quantitative and qualitative metrics.

The large number of parsers available to test can help evaluate the correctness and accuracy of each lint rule. We might find that certain rules are more prone to false positives than others, or that some automated refactorings may introduce new issues such as compilation errors. These quantitative metrics can be counted and used to evaluate rules against each other, as well as the overall effectiveness of the tool.

We can also estimate the proportion of issues that `parsley-garnish` is able to detect automatically. This can be done by comparing the number of issues detected automatically against the number of issues found manually by a code reviewer.

Qualitative metrics such as the quality of automated fixes can also be assessed by manually reviewing the output of the tool. For example, an issue I pointed out in the project plan is that the output of the left-recursion transformation may be in a form that is “ugly” when it could be simplified to a much more human-readable form. The quality of fixes would affect the practicality of the `parsley-garnish`, since users may be reluctant to apply automated fixes if the output quality can be poor.

Furthermore, the performance of `parsley-garnish` can be evaluated in terms of execution time and resource consumption. There are no direct competitors to benchmark against, but we can still evaluate the tool’s performance against a baseline of acceptable performance, or perhaps a collection of other Scala static analysis tools deemed to be of similar complexity. I am thinking of benchmarking the tool during development as well, since it may affect design decisions such as implementing some rules as purely syntactic rather than semantic rules. I am not sure how much resource overhead a semantic rule would incur, so this could be insignificant or a major tradeoff (between speed, correctness, and accuracy) to consider.

Finally, it is possible to gather feedback from students interested in using `parsley-garnish` to improve their WACC parsers. User surveys and interviews would provide insights into the tool’s usability, helpfulness, and areas for improvement. Although the timing of the WACC Lab

does not align with the project timeline, I am hoping that some students may still be interested in trying out the tool on their final WACC submissions or possibly personal projects using parsley.

Bibliography

- [1] Hutton G. Higher-order functions for parsing. *Journal of Functional Programming*. 1992 Jul;2(3):323-43. Available from: <https://doi.org/10.1017/S095679680000411>.
- [2] Willis J, Wu N. Garnishing parsec with parsley. In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. Scala 2018. New York, NY, USA: Association for Computing Machinery; 2018. p. 24-34. Available from: <https://doi.org/10.1145/3241653.3241656>.
- [3] Hudak P. Building domain-specific embedded languages. *ACM Comput Surv*. 1996 dec;28(4es):196-es. Available from: <https://doi.org/10.1145/242224.242477>.
- [4] Leijen D, Meijer E. Parsec: Direct Style Monadic Parser Combinators for the Real World; 2001. UU-CS-2001-27. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007. Available from: <https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/>.
- [5] Willis J, Wu N. Design patterns for parser combinators in scala. In: *Proceedings of the Scala Symposium*. Scala '22. New York, NY, USA: Association for Computing Machinery; 2022. p. 9-21. Available from: <https://doi.org/10.1145/3550198.3550427>.
- [6] Kurbatova Z, Golubev Y, Kovalenko V, Bryksin T. The IntelliJ Platform: A Framework for Building Plugins and Mining Software Data. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*; 2021. p. 14-7. Available from: <https://doi.org/10.1109/asew52652.2021.00016>.
- [7] Renggli L, Ducasse S, Gîrba T, Nierstrasz O. Domain-Specific Program Checking. In: Vitek J, editor. *TOOLS'10: Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*. vol. 6141 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer; 2010. p. 213-32. Available from: https://doi.org/10.1007/978-3-642-13953-6_12.
- [8] Gregor D, Schupp S. STLint: lifting static checking from languages to libraries. *Software: Practice and Experience*. 2006;36(3):225-54. Available from: <https://doi.org/10.1002/spe.683>.

- [9] Johnson SC. Lint, a C program checker. AT&T Bell Laboratories; 1978. Technical Report 65.
- [10] Hovemeyer D, Pugh W. Finding bugs is easy. SIGPLAN Not. 2004 dec;39(12):92-106. Available from: <https://doi.org/10.1145/1052883.1052895>.
- [11] Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, et al. A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. PLDI '03. New York, NY, USA: Association for Computing Machinery; 2003. p. 196-207. Available from: <https://doi.org/10.1145/781131.781153>.
- [12] Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C. Lessons from building static analysis tools at Google. Commun ACM. 2018 mar;61(4):58-66. Available from: <https://doi.org/10.1145/3188720>.
- [13] Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, et al. Moving Fast with Software Verification. In: Havelund K, Holzmann G, Joshi R, editors. NASA Formal Methods. Cham: Springer International Publishing; 2015. p. 3-11. Available from: https://doi.org/10.1007/978-3-319-17524-9_1.
- [14] Fowler M. In: Refactoring: Improving the Design of Existing Code. Addison-Wesley; 2018. .
- [15] Parr T. The Definitive ANTLR 4 Reference. 2nd ed. Pragmatic Bookshelf; 2013.