Imperial College
London

Parslers: A staged selective parser combinator library for the Rust programming language

How to make fast and easy-to-use parser combinator libraries in Rust faster and easier to use

Jordan Hall

Individual Project
Bachelors of Engineering – Computing
September 1, 2023

INDIVIDUAL PROJECT	
Imperial College London	
Degree Programme:	Bachelors of Engineering – Computing
Identification number:	
Author:	Jordan Hall
Title:	Parslers: A staged selective parser combinator library for the Rust programming language How to make fast and easy-to-use parser combinator libraries in Rust faster and easier to use
Supervisor:	Jamie Willis
Abstract:	

Keywords:	Parser Combinator, Rust, Parslers, Parsing
Number of pages:	58
Language:	English

CONTENTS

FOREWORD

ACKNOWLEDGMENTS

I'd like to thank all those who helped support me in this project, including (but not limited to):

Jamie Willis, for inspiring my curiosity in the field of parsing and especially parser combinators.

1 INTRODUCTION

In this project, I aim to implement a library for the Rust programming language. The aim of this library is to provide tools to help non-expert parser architects develop fast parsers with an easy-to-use interface. I will leverage the Rust language's powerful meta-programming tool – Procedural Macros – to implement a domain-specific language that is both easy to use and optimizes the built parser into fast, native Rust code at compile time.

The motivation is derived from the fact that generating parsers of structured data (e.g programming languages) often comes with a lot of complexity. Programmers have to concern themselves with speed, correctness and maintainability of their parsers, when parsing is oftentimes not the focal point of the language.

Although many parser generators and libraries exist, few are able to both optimize the resulting parser and provide a user-friendly interface for the programmer. Even fewer parser libraries and generators have structures similar to the original grammar. However, functional parsers (including parser combinators) oftentimes look similar to the grammar that they parse. Recent research into parser combinators show that if the structure of the parser is known at compile time, then they can be optimized by languages that offer staged compilation (such as Haskell, Scala 3, and Rust). This makes parser combinators a promising methodology for language designers.

Languages	Automaton
Regular	Finite state automaton
Context-free	Non-deterministic pushdown automaton
Context-sensitive	Linear-bounded non-deterministic Turing machine
Recursively enumerable	Turing machine

2 BACKGROUND

The subject of parsers and parser libraries both have a rich history in the field of computer science. This section will introduce the background of parsing as a field of computer science, make the case for why further development into parsing speed is necessary, and lay out the groundwork for Parslers as the solution to writing fast and succinct parsers in the Rust programming language.

2.1 Parsing in Computer Science

In the field of computer science, parsing is a process used to analyse a string of symbols, usually text, in accordance with a set of rules known as a grammar. Early work looked to describe and understand natural languages (such as English or Spanish) in a way that made it amenable to analyse using mathematical principles. The resulting field of study, formal language theory (Jäger & Rogers 2012), laid the groundwork that made writing computer programs capable of analysing and generating said natural languages possible.

In modern times, parsing has become a core tool used by software engineers on a daily basis. Software engineers interact with compilers using a language format specification (the syntax of a language), which must be parsed into structured data for a compiler or interpreter to process. Different programs interact with one another using a serialisation/deserialisation format (such as JSON, XML) in order to communicate. Images are encoded into a shared structured format and must be parsed by image-viewing software in order to be displayed.

As parsing data is so ubiquitous in computer science, it follows that much research has been done on formalising the various methods computer scientists and programmers use to translate structured data into text, and vice versa.

2.2 Formal Grammar

For mathematicians to build upon each other's work, they must first agree on the method with which they communicate. Although linguists and etymologists have discussed the nature of languages for millennia prior, mathematicians, such as Noam Chomsky, defined the concrete notion of formal grammar as a set of rules and valid productions.

An example of such a grammar would be:

$\langle S \rangle$ ::= $\langle S \rangle$ 'a'
  | 'b'

Strings of symbols such as ba, b, $bbbaaa$, and $baba$ are all valid strings within this grammar. However, ab would not be considered a valid string within this grammar, as there's no way to reduce this string down to S.

Chomsky formulated a hierarchy of formal grammar, which describe the power of all different grammar types and the type of automaton necessary to parse such a grammar (Allott, Lohndal & Rey 2021).

2.2.1 Regular Grammar

Regular grammars are the most basic form of formal grammar. They are grammars in which each production must always start with a terminal:

⟨*S*⟩ ::= 'a' ⟨*S*⟩
  | 'b' ⟨*S*⟩

This is to say, a production such as $S \rightarrow aSa$ would not be a valid regular grammar. The most common use case of regular grammars is the regular expression language Regex. Regex provides fast text validation with simple expressions.

An example of a Regex expression is one used to validate dates in the form DD/MM/YYYY:

```
1 (((0[1-9]|[12][0-9]|3[01])([/])(0[13578]|10|12)([/])(\d{4}))|(([0][1-9]|[12][0-9]|30)([/])
  ↪  (0[469]|11)([/])(\d{4}))|((0[1-9]|1[0-9]|2[0-8])([/])(02)([/])(\d{4}))|((29)(\.|-|\/)(02)([/])
  ↪  ([02468][048]00))|((29)([/])(02)([/])([13579][26]00))|((29)([/])(02)([/])([0-9][0-9][0][48]))
  ↪  |((29)([/])(02)([/])([0-9][0-9][2468][048]))|((29)([/])(02)([/])([0-9][0-9][13579][26])))
```

2.2.2 Context-Free Grammar

Context-free grammar are written with production rules in the form:

⟨*S*⟩ ::= w

Where S is the non-terminal to be produced, and w is a string of terminals and non-terminals. What makes such a grammar context-free is that S can be produced anywhere that w is successfully matched. It does not depend on the surrounding context.

It is worth noting that most computer programming languages are described using context-free grammar because they are typically easier to generate efficient parsers for. Furthermore, context-free grammars allow programmers to quickly create mental models that can be applied when writing syntax for the language.

Although context-free grammars are the most widely-used category of grammar for everyday use cases, they are to some extent limited. A good example is parsing a number. A grammar for a floating-point number can be described using the following grammar:

⟨*Float*⟩ ::= '-' ⟨*UnsignedFloat*⟩
  | ⟨*UnsignedFloat*⟩

⟨*UnsignedFloat*⟩ ::= ⟨*UInt*⟩ '.' ⟨*UInt*⟩

⟨*UInt*⟩ ::= ⟨*Digit*⟩ ⟨*UInt*⟩
  | ⟨*Digit*⟩

⟨*Digit*⟩ ::= '0'..'9'

Although this grammar correctly describes all real numbers, a context-free grammar cannot be used to easily describe the real numbers describable by a fixed-size floating point format. It is possible by enumerating all values that fit within this criteria, however this is not useful for generating efficient parsers. For this reason, parsing is generally performed before a semantic check of the parsed data. This way, grammar for programming languages may remain context-free for simplicity.

2.3 Tools for Writing Parsers

Almost every program that deals with text in some way likely needs to convert it into structured data to be useful. To fulfil this need many tools and libraries have been developed to ease the time, speed, and correctness of parsers.

2.3.1 Parser Generators

Parser generators offer an end-to-end solution for writing parsers, the most famous of which are ANTLR and Bison. They both target multiple languages, and work by taking in formatted grammar and output autogenerated code in the target language. Parser generators work well for languages that do not have strong meta-programming capabilities, and they oftentimes use grammar formats that look similar to the original grammar. ANTLR uses Extended Backus-Naur form, and Bison uses BNF. Parser generators that use produce LR parsers, like Bison, are able to optimise the underlying parser during generation.

However, parser generators typically produce concrete syntax trees after parsing. This makes it difficult to maintain the resulting parser because changes to the grammar will result in the language designer having to also change the code that converts the concrete syntax tree into the abstract syntax tree, suitable for semantic analysis.

2.3.2 Handwritten Parsers

Handwritten parsers are generally written from scratch in the target language. The most common of which are top-down parsers.

Languages with hand-written parsers include:

- Java (OpenJDK)
- Golang
- Swift
- TypeScript
- JavaScript
- C/C++ (both GCC and Clang)

There are several reasons a language designer would write their own hand-written parsers, but Golang (abbreviated *Go*) provides an interesting case-study. Go originally used a YACC-based parser, a parser generator similar to Bison and ANTLR mentioned above. In November 2015, Go switched to a hand-written parser (Griesemer & Fitzpatrick 2015), which yielded a 18% parsing speedup, and a 3% compilation speedup for the compiler itself. Because hand-written parsers give the language designer increased control in how lexing/parsing takes place, oftentimes resulting in faster parsers. Handwritten parsers can also generate the abstract syntax tree directly during parsing.

Handwritten parsers are also not limited to context-free grammars, as opposed to many parsing libraries and parser generators.

Unfortunately, handwritten parsers suffer from maintainability issues. Because lexing, parsing, error messaging, and AST generation must all be programmed by the language designer, the parser now becomes a significant portion of the codebase where it otherwise wouldn't be. Furthermore, writing

a correct and optimised handwritten parser requires expert knowledge on how to architect such parsers.

2.3.3 Parser Combinators

Parser combinators work by composing many higher-order functions together and then applying that function to an input stream to generate an output AST (Hutton & Meijer 1996). The parsing code can be integrated with the rest of the codebase, much like a hand-written parser. They can produce the target AST directly during parsing. Furthermore, parser combinators look very similar to the original grammar in many implementations, which make them maintainable.

Until recently, parser combinators were considered slow in comparison to handwritten parsers and parser generators. However, more recent research has shown that parser combinators can be optimized in a separate step from the rest of the program, at compile time, by removing the monad (Willis, Wu & Pickering 2020). This allows parser combinators to look like the grammar when writing them, but are optimized and then compiled down to native code in the target language.

2.4 The Rust Programming Language

2.4.1 History

The Rust programming language was developed by Graydon Hoare at Mozilla in 2006. Mozilla then began officially sponsoring the development of the language in 2009. The first stable version of the Rust programming language was released in 2015. In 2021, the Rust Foundation was established, taking over development of the Rust programming from Mozilla.

The initial motivation came from a common problem in modern software engineering: concurrency. As modern computer processing unit improvements focus less and less on raw speed and more towards very high core counts, concurrent programming is seen as the way to improve the performance of computer programs. However, this comes at a cost in the mental overhead involved in programming. The difficulty stems from the fact that modern CPUs use the Total-Store-Ordering concurrent memory model. This means that changes to memory locations in RAM made by one core may be stored in a core-specific write-back buffer, and not reflected in RAM (and thereby other cores) for an unpredictable period of time. This is known as a data race. When writing concurrent software in traditional programming languages, a great deal of effort must be made to ensure that potentially concurrent memory access is atomic, and other cores are notified in a predicable time frame.

A motivating example of this was the previous Mozilla Firefox CSS engine. CSS styling is a top-down process, wherein the children of the parent object can be styled independently top-down (and thus in parallel). There have been attempts to parallelize the previous C++ engine to leverage this, however none were successful as the codebase was too large, and the mental workload for maintaining the concurrent invariants turned out to be overwhelming.

To solve this problem, Rust utilizes a "borrowing" memory model, whereby objects can be "owned" by a scope, and then borrowed immutably many times, or borrowed mutably only once, but never both at the same time. This solves the problem of data races by ensuring that no two cores can read and write to a memory location at the same time. Rust then builds an abstraction over these rules by providing mutex and atomic variable implementations, as well as smart pointers that automatically destruct objects when they are no longer in use.

2.4.2 Rise to Popularity

Rust has become incredibly popular since release, being top of Stack Overflow Developer Survey's "Most Loved Programming Language" every year from 2016-2022 (the time of writing). Many at-

tribute this popularity to its ease of use and speed.

2.4.3 Meta-programming using Procedural Macros

The Rust programming language also provides a powerful tool for assisting development. The Procedural Macro is a Rust program that is run during compile time, and transforms a sequence of parsed Rust tokens to another sequence of Rust tokens, which are then compiled alongside the rest of the Rust program. The Procedural Macro is a core part of the language as it provides a tool for library developers to make using said library easier to develop with, saving time.

Take, for example, Rocket. Rocket is a framework for developing web servers in Rust, and provides many macros to improve the ease of development.

An example of a simple web-server that returns "Hello World" when called.

```
#[macro_use]
extern crate rocket;

#[get("/")]
fn hello() -> &'static str {
  "Hello, world!"
}

#[launch]
fn rocket() -> _ {
  rocket::build()
    .mount("/", routes![hello])
}
```

The `#[get("/")]` and `#[launch]` tags are macros that tell the compiler to parse the resulting functions into a steam of tokens, and pass them into a Rust function to transform them. We can see how the rocket procedural macros expand the little code above into the code below:

```
#[macro_use]
extern crate rocket;
fn hello() -> &'static str {
  "Hello, world!"
}

#[doc(hidden)]
#[allow(non_camel_case_types)]
/// Rocket code generated proxy structure.
struct hello {}
/// Rocket code generated proxy static conversion implementations.
impl hello {
  #[allow(non_snake_case, unreachable_patterns, unreachable_code)]
  fn into_info(self) -> ::rocket::route::StaticInfo {
    fn monomorphized_function<'__r>(
      __req: &'__r ::rocket::request::Request<'_>,
      __data: ::rocket::data::Data<'__r>,
```

```
    ) -> ::rocket::route::BoxFuture<'__r> {
      ::std::boxed::Box::pin(async move {
        let ___responder = hello();
        ::rocket::route::Outcome::from(__req, ___responder)
      })
    }
    ::rocket::route::StaticInfo {
      name: "hello",
      method: ::rocket::http::Method::Get,
      uri: "/",
      handler: monomorphized_function,
      format: ::std::option::Option::None,
      rank: ::std::option::Option::None,
      sentinels: <[_]>::into_vec(
        #[rustc_box]
        ::alloc::boxed::Box::new([
          {
            #[allow(unused_imports)]
            use ::rocket::sentinel::resolution::{DefaultSentinel as _, Resolve};
            ::rocket::sentinel::Sentry {
              type_id: std::any::TypeId::of::<&'_ str>(),
              type_name: std::any::type_name::<&'_ str>(),
              parent: None,
              location: ("src/main.rs", 5u32, 15u32),
              specialized: Resolve::<&'_ str>::SPECIALIZED,
              abort: Resolve::<&'_ str>::abort,
            }
          },
          {
            #[allow(unused_imports)]
            use ::rocket::sentinel::resolution::{DefaultSentinel as _, Resolve};
            ::rocket::sentinel::Sentry {
              type_id: std::any::TypeId::of::<str>(),
              type_name: std::any::type_name::<str>(),
              parent: None.or(Some(std::any::TypeId::of::<&'_ str>())),
              location: ("src/main.rs", 5u32, 24u32),
              specialized: Resolve::<str>::SPECIALIZED,
              abort: Resolve::<str>::abort,
            }
          },
        ]),
      ),
    }
  }
  #[doc(hidden)]
  pub fn into_route(self) -> ::rocket::Route {
    self.into_info().into()
  }
}

```

As you can see, macros do a lot of work behind the scenes for developers, and allow Rust to have libraries that are incredibly easy to use.

3 ETHICAL ISSUES

This project is aimed to help people make efficient, correct, and maintainable parsers. Because of this scope, there are very few ethical considerations that must be taken into account. That being said, it is important in the conduct of this project to point out that the resulting library may be used in production, or on projects where correctness is of great importance. This puts the responsibility of correctness onto me, the creator of the library. I must also be honest and clear in all places where the library may be found that the aim of the project was not to create safety-critical parsers, and there may be issues that I have not yet identified.

4 PARSLERS

This chapter introduces Parslers, the library developed for this thesis. Parslers is a Rust parser-combinator framework developed to empower non-expert parser developers to generate blazingly fast parsers while only using high-level abstractions. This is achieved by using Rust's staged compilation to take high-level parsers and compile them into native Rust code, all at the compile step.

Section Requirements lays out the requirements that Parslers would have to meet to become a viable parser combinator library competitive with the rest of the monadic parser combinator libraries already available in Rust.

Section Library Structure shows how the user is expected to interact with Parslers. This section also details how Parslers makes use of Rust's staged compilation pipeline to perform optimisations on user-generated parsers and generate native Rust code.

Section Parser Definition Interface follows by showing how Parslers is different from other monadic parser combinator libraries by removing the monad entirely, and how Parslers uses the selective combinators to regain some of the expressive power lost with this decision. It also introduces the primitive and auxiliary combinators. Finally, the Parsler trait is introduced, one of the core traits that the library users interact with in order to build their parsers.

Section The Reflect Trait starts by introducing the key challenge in the staged-compilation pipeline - how to ensure that runtime functions and values survive the build phase into the generated native code. It also discusses the challenges involved in the chosen solution, and how Parslers overcomes these challenges to achieve runtime reflection of functions - a novel approach to runtime reflection.

With the internals of the library laid out, sections Lazy Parsers, Named Parsers, Looping Parsers, and Alternative Parser discuss the parser combinators that allow for complex structure and improved efficiency. In particular, Lazy Parsers discusses the `LazyParser`, which allows Parslers to overcome the problem of infinite recursion when parsers are defined structurally instead of the commonly used higher-order function approach. Named Parsers introduces the `NamedParser`, which is used to prevent parsers from being duplicated during code generation, which is essential for recursive parsers. Looping Parsers tackles the problem of efficiency arising from the fact that Rust does not yet support tail-call optimisation. This section then goes on to solve this problem by introducing a novel parser, the `LoopParser`, which allows transforming recursive parsers into iterative parsers.

Now that the user-facing interface has been properly introduced, section Backend Optimisations discusses how fixing the structure of the parser by removing the monad can be utilised to perform static analysis on the users' parsers at compile-time. This section will lay out the current optimisations implemented by Parslers, including lawful reduction, and usage analysis optimisations. It will then finish by discussing the possible extensions to static analysis that can be implemented in the future in order to further improve Parslers' performance characteristics.

Finally, section Code Generation details how these high-level parsers are transformed into native Rust code. It also discusses how functions and values used in the high-level parser definition survive into the generated code.

4.1 Requirements

In order for Parslers to become a viable parser combinator library suitable to replace the current existing parsing libraries available to the Rust ecosystem, a number of requirements must be set and

adhered to. This section discusses those requirements, which will be the guiding decisions used in the design-process of the library, and each will be evaluated in turn at the end of this report.

4.1.1 Expressive Power

Expressive power means the range of grammars that can be parsed by Parslers. Detailing this is key, as it will help define which parsing problems Parslers can be applied to. Because Parlsers is a recursive-descent parser combinator library, it makes sense to require that Parslers can parse any PEG grammar. The evaluation section will show that Parslers can parse a wider range of grammars than PEG.

4.1.2 Performance

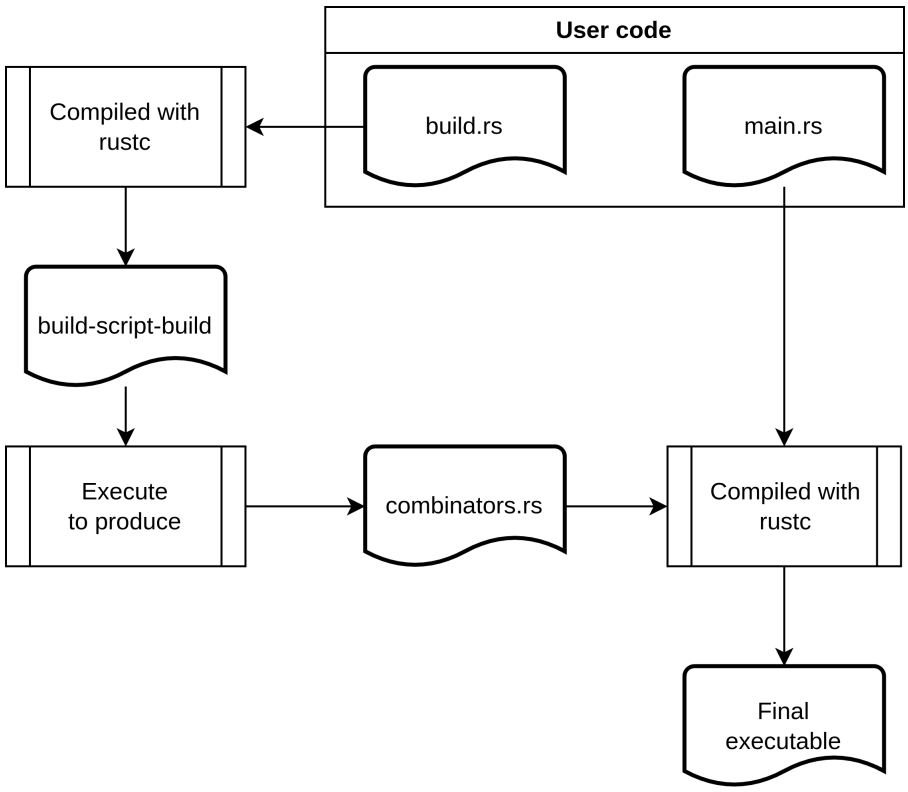
Parlsers' main concern as a parsing library is using static analysis to optimise user-written parsers at compile time. It, therefore, follows that Parslers should have performance characteristics better than other parsing libraries available in the Rust ecosystem. However, it is acceptable for Parslers to be out-performed by hand-written parsers for specific grammars, as these parsers are embedded with the full possible domain-specific knowledge available. Parslers will, however, attempt to achieve performance characteristics similar to hand-written parsers where possible.

4.1.3 Correctness

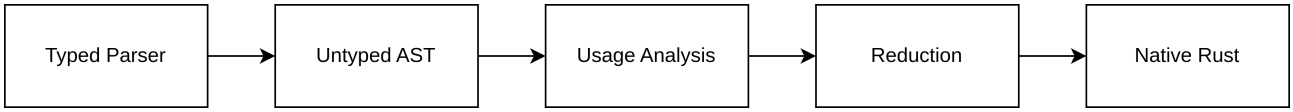
Although parsers are often tested on a large corpus of correct and incorrect example text, correctness is still one of the top priorities of Parslers. As per this requirement, Parslers is required to be type-safe in the user-facing aspects of the library. Furthermore, the restrictions on the types of parsers should prevent ill-formed parsers from being created at compile time. There should be no runtime checks on the correctness of the parsers, purely relying on the strong type-checking Rust provides at compile time.

4.2 Library Structure

Parslers takes a staged-compilation approach to parser generation. The user writes their parsers as typed Rust in the build.rs file that is compiled and run before the main binary is compiled. Parslers leverages this stage of compilation to perform optimisations on the user-written parsers before translating the parsers into native rust functions in a separate file. The user then includes this file as an import to their codebase, where it is compiled with the rest of the binary.



Because Rust will type-check the user's parser for us, we are then able to convert to an untyped AST to perform optimisations on it. Because the optimisations performed on the AST do not change the type of the produced values, they may be compiled into Rust code without needing to embed Rust's type checker into the Parslers library. The structure of Parslers is as follows:



Usage analysis and structural reduction are the two key optimisation techniques which make Parslers' generated parsers blazingly fast. We will detail these optimisations later in the report.

4.3 Parser Definition Interface

Now that Parslers' staged compilation has been explained, it is time to move on to how the library presents the type of the parsers users construct. This section starts off by introducing the monadic parser, and how representing the parser as a monad destroys opportunities for static analysis to be performed on said parsers. It then goes on to discuss how the monad plays a large role in the expressive power of monadic parsers, before introducing a combinator that helps regain some of this expressive power - the selective combinator. After this, the primitive combinators are introduced, which form the building blocks used to build parsers and auxiliary combinators. This section then finishes off by introducing the `Parslers` trait, which all parsers and combinators must implement.

4.3.1 Fixing the structural problem

The most commonly-used parser combinator libraries in Rust cannot currently perform static analysis on parsers created with them. This section will discuss why the root of this problem lies in the use of a parser as a monad, and how removing the monad can allow for static analysis in Parslers. Then, this section will discuss how removing the monad leaves us without the full expressive power it provides, and how introducing the selective parser combinator helps to retrieve

some of this power.

What separates Parslers from the monadic parser combinator libraries is that it lacks the ability to generate arbitrary parsing structure at runtime. That is to say, there is no valid combinator which, as the result of previous input, generates a new parser to be executed on the remaining input. Because Parslers requires that the structure of parsers is fixed, this allows for static analysis to be performed at compile time.

In essence, a combinator in parser combinators is usually expressed as a monad. Monadic parsers typically have two key functions:

- `pure :: a -> Parser a`
- `flatMap :: (Parser a) -> (a -> Parser b) -> (Parser b)`.

`pure` simply lifts the value `a` into a parser which produces the value `a` when applied to an input string. `pure` does not pose a problem to static analysis, because its structure is known at compile-time. The problem lies with the `flatMap` method (also known as `bind`). `a -> Parser b` can generate an arbitrary parser based on the value of `a`. If the structure of the parser is to be known at compile-time, then `flatmap` will not be available to us.

The fixed structure does come at a cost of expressive power. If the user wishes to write an integer parser, they are constrained to the types Rust provides (`u8` , `u16` , `u32` , `u64` , `usize` , `i8` , `i16` , `i32` , `i64` , `isize`). If the user would like to constrain the parsed values to fit inside of a `u8` , then the parser would have to change behaviour based on the parsed value. This would typically be done using the `flatMap` operator, yielding either the parsed value as a `u8` , or an error if the value is too large.

Because `flatMap` is not available to Parslers, a selective combinator must be introduced. Previous work on selective parser combinators (Willis, Wu & Pickering 2020) produced the `branch(b, l, r)` combinator. In essence, it is a parser with the type:

```
branch :: Parser (Either a b) -> Parser (a -> c) -> Parser (b -> c) -> Parser c
```

`branch` executes parser `b` , and based on the result of that, will execute parser `l` or `r` . With this combinator, it would not be too difficult to generate a parser which reads a string of digits, attempts to parse it into a `u8` (returning `Either<u8, ()>`), and then either return it in parser `l` or throw an error in parser `r` .

By using `branch` , some of the expressive power of the monad can be restored to the fixed structure of Parslers. Further work has been done on selective parser combinators that implements what's known as a register (Willis, Wu & Schrijvers 2022), a mutable value that can be threaded throughout the parser. Using a register with the selective `branch` combinator restores the most of the expressiveness of the monad while maintaining the fixed structure of the parser.

4.3.2 The Primitive Combinators

Parser combinator libraries are designed with the approach that complex parsers can be represented by composing higher-order functions together. The leaf nodes of these higher-order functions are primitive combinators, which perform a very basic parsing task such as reading a character. Function-based parser combinator libraries (such as nom) allow the users to create these primitive combinators manually, and provide frequently used primitive combinators. However, because

Parslers represents parser combinators structurally, the set of primitive combinators is fixed and must encapsulate everything needed for the user to write their parsers.

The list of available primitive combinators in Parslers is:

```
Pure(a)
Satisfy(f)
Attempt(p)
Or(p, q)
Then(p, q)
Before(p, q)
Ap(p, q)
Map(p, f)
Recognise(p)
Look(p)
NegLook(p)
Empty
Branch(b, l, r)
LazyParser(f)
NamedParser(p, name)
Loop(p0, pN)
```

`Pure(a)` simply returns the value `a`, consuming no input.
`Satisfy(f)` takes a function `f` in the form `FnOnce(char) -> bool`. It will apply `f` to a consumed character, and return that character upon success. Will fail if `f` returns false and backtracks.
`Attempt(p)` will perform parser `p`, backtracking if `p` fails.
`Or(p, q)` will attempt to perform parser `p`, and then parser `q` if `p` fails. Does not backtrack.
`Then(p, q)` will perform parser `p`, then parser `q`, returning only the result of parser `q`.
`Before(p, q)` will perform parser `p`, then parser `q`, returning only the result of parser `p`.
`Ap(p, q)` takes parsers `p` and `q` where `p` returns a function to be applied to `q`. It will perform both in sequence, and then apply the results.
`Map(p, f)` is equivalent to `Ap(Pure(f), p)`, but added for brevity.
`Recognise(p)` performs `p`, and then returns the string consumed by `p`.
`Look(p)` performs the parser `p`, and then backtracks on the consumed characters.
`NegLook(p)` performs the parser `p`, and fails if `p` succeeds.
`Empty` always fails.
`Branch(b, l, r)` is described in the section Fixing the structural problem
`LazyParser(f)` is described in the section Lazy Parsers
`NamedParser(p, name)` is described in the section Named Parsers
`Loop(p0, pN)` is described in the section Looping Parsers

4.3.3 Auxiliary Combinators

The primitive combinators introduced are enough to form any higher-order combinator that the user could use within the context of Parslers. However, many non-primitive combinators are frequently used in the construction of parsers. For this reason, Parslers comes pre-packaged with many frequently-used higher-order combinators to reduce the workload of the user.

The list of pre-packaged auxiliary combinators in Parslers are:

```
some_rev(p)
many_ref(p)
some(p)
many(p)
some_map(p)
many_map(p)
match_char(c)
one_of(cs)
filtered_by(p, f)
ident(f)
tag(s)
not(c)
opt(p)
ws(p)
dynamic(p)
```

`some_rev(p)` performs `p` at least once and collects the results into a `Vec` in reverse order. This is the recursive version of `some(p)`.
`many_rev(p)` performs `p` zero or more times and collects the results into a `Vec` in reverse order.

This is the recursive version of `many(p)`.
`some(p)` does the same as `some_rev(p)`, but the results are added to the `Vec` in parsed order. This uses the `LoopParser` instead of recursion.
`many(p)` does the same as `many_rev(p)`, but the results are added to the `Vec` in parsed order. Like `some(p)`, this uses the `LoopParser` instead of recursion.
`some_map(p)` does the same as `some(p)`, but the results are collected into a `HashMap` instead of a `Vec`.
`many_map(p)` does the same as `many(p)`, but the results are collected into a `HashMap` instead of a `Vec`.
`match_char(c)` is equivalent to `Satisfy(|ch| ch == c)`. It is added for brevity.
`one_of(cs)` takes an iterator that produces at least one **`char`**, and produces a parser which attempts each character in sequence.
`filtered_by(p, f)` takes a parser with output type `O`, and function of the type `FnOnce(O) ->` **`bool`**. It succeeds if `f` returns true on the output of `p`, failing otherwise.
`ident(f)` matches a valid identifier that is not a keyword (as denoted by the function f returning false).
`tag(s)` takes `s: &str` and parses this string character by character. Returns the original string upon success.
`not(c)` does the same as `match_char(c)`, but fails if the character is matched.
`opt(p)` makes `p` optional, and is the same as `Or(Then( p, pure(()) ), pure(()) )`.
`ws(p)` consumes trailing whitespace after parsing `p`.
`dynamic(p)` converts `p` into a heap-allocated dynamically dispatched version of `p`. This is useful for recursion.

4.3.4 The Parsler Trait

The previous sections introduce that the primitive and auxiliary parsers have been introduced which show the building blocks of complex parsers written in this library, and how Parslers does not use the monadic parser. This section wraps up the user interface by introducing the trait that all of these combinators implement, the `Parslers` trait. It provides the trait methods that allow parsers to be tested and compiled into an untyped AST. Despite its importance to the library, the definition is simple:

```
trait Parsler {
    type Output;

    fn parse(&self, input: &mut Chars) -> Result<Self::Output, String>;
    fn compile(&self, info: &mut CompileContext) -> ast::Parser;

    fn ap<B, F, P>(self, p: P) -> Ap<B, F, Self, P>
    where
        Self: Sized,
        P: Parsler<Output = F>,
        F: FnOnce(Self::Output) -> B,
    {
        Ap(self, p)
    }

    fn map<B, F>(self, f: F) -> Map<B, Self, F>
    where
        Self: Sized,
        F: FnOnce(Self::Output) -> B + Clone + Reflect,
```

```
    {
        Map(self, f)
    }

    fn then<P>(self, p: P) -> Then<Self, P>
    where
        Self: Sized,
        P: Parsler,
    {
        Then(self, p)
    }

    fn or<P>(self, p: P) -> Or<Self, P>
    where
        Self: Sized,
        P: Parsler<Output = Self::Output>,
    {
        Or(self, p)
    }

    fn before<P>(self, p: P) -> Before<Self, P>
    where
        Self: Sized,
        P: Parsler,
    {
        Before(self, p)
    }

    fn attempt(self) -> Attempt<Self>
    where
        Self: Sized,
    {
        Attempt(self)
    }

    fn look(self) -> Look<Self>
    where
        Self: Sized,
    {
        Look(self)
    }

    fn neg_look(self) -> NegLook<Self>
    where
        Self: Sized,
    {
        NegLook(self)
    }
}
```

Every parser combinator in Parslers must implement the `Parsler` trait. It is built up of three components:

1. `type Output;` represents what the parsers produce. This is vital for making sure that combined parsers are well-formed upon creation.

2. `fn parse(&self, input: &mut Chars) -> Result<Self::Output, String>;` must be implemented for each combinator separately, and is used as a debug parser before code generation.

3. `fn compile(&self, info: &mut CompileContext) -> ast::Parser;` must also be implemented for each combinator, and describes how the parser can be compiled into the backend abstract syntax tree for generating native code.

4. `ap; map; then; or; before; attempt; look; negLook` merely act as chained combinators, and are implemented for all cases. They allow users to chain parsers together instead of relying on deeply nested structs, which can be hard to read.

Notice that for the `map` function, it takes a function `F: FnOnce(Self::Output) -> B + Clone + Reflect`. This is a function that can only be called once, can be cloned, and implements a `Reflect` trait. I will discuss this trait in the next section, but it is what allows us to achieve runtime reflection.

Let's take a look at the simplest implementor of `Parsler`, the `Pure` combinator:

```
#[derive(Clone, Debug)]
struct Pure<A: Reflect>(A);

impl<A: Clone + Reflect> Parsler for Pure<A> {
    type Output = A;
    fn parse(&self, _input: &mut Chars) -> Result<Self::Output, String> {
        let val = self.0.clone();
        Ok(val)
    }

    fn compile(&self, _info: &mut CompileContext) -> ast::Parser {
        ast::Parser::Pure(ast::PureVal::Val(self.0.to_string()))
    }
}

fn pure<A: Reflect>(a: A) -> Pure<A> {
    Pure(a)
}
```

All `Pure` does as a combinator is return a value. It does not consume any input, and always succeeds. Notice how `A` is also required to implement Reflect, in order to retrieve the definition of the value at compile time.

Notice that the `parse()` function takes a mutable reference to a `Chars` struct. `Chars` is an iterator that produces characters from a string. Because Parslers does not implicitly backtrack, we only need

to pass around a mutable reference to this iterator. Therefore, the `parse()` function does not need to return the remaining unparsed string.

For combinators that enable backtracking, like `Attempt`, `Chars` can be cloned. This produces a new iterator at the same starting point as the source iterator. We use this functionality in the `Attempt` operator as follows:

```
impl<P1: Parsler> Parsler for Attempt<P1> {
    type Output = P1::Output;
    fn parse(&self, input: &mut Chars) -> Result<Self::Output, String> {
        let copied_input = input.clone();
        let a = self.0.parse(input);
        if a.is_err() {
            *input = copied_input;
        }
        a
    }
    // ...
}
```

So `Attempt` will clone `Chars` before parsing it to the inner parser. If the inner parser fails, it will then backtrack by replacing the original input with the cloned input. In this way, we obtain backtracking functionality.

4.4 The Reflect Trait

Parslers is a staged selective parser combinator library. The staged aspect of Parslers is derived from the fact that Parslers compiles user-created parsers during the build stage of Rust's compilation pipeline. The native code that Parslers generates is placed into a module for the user to include in their Rust programs. This poses an interesting architectural problem, as functions are not inspectable during the runtime of Rust programs.

If Parslers is to achieve staged compilation, it must also place functions and values defined by users in the build.rs file into the generated Rust code. This is equivalent to runtime reflection to the build.rs file. Although Rust does not support this directly, work has been done by other libraries needing runtime reflection. An example of this is the game engine Bevy (Anderson 2020), which uses the helper library `bevy_reflect` in order to achieve runtime reflection of values. However, there is no equivalent library in Rust that allows for runtime reflection of functions. In order to make staged compilation viable for Parslers, it must also support runtime reflection of functions, and allow function definitions to survive into the generated native Rust code. This section details how Parslers achieves this.

The `Reflect` trait is how we are able to achieve runtime-reflection:

```
trait Reflect {
    fn name(&self) -> &'static str {
        std::any::type_name::<Self>()
    }

    fn reflect(&self) -> String;
```

```
}
```

The purpose of this trait is to have the definition of the value available at runtime. The `to_string()` method gives us the value definition as a string, which can then be inserted into the generated code during the native code compilation step. It is already defined for all primitives and compound types provided in Rust. It's implementation for **`char`** is rather simple:

```
impl Reflect for char {
    fn reflect(&self) -> String {
        format!("'{}'", self)
    }
}
```

However, it gets a bit more complicated for functions. This is because functions are black boxes in Rust, and can not be inspected at compile time or runtime. To make matters worse, you cannot implement traits over specific functions as of yet. To get around this, Parslers comes with an attribute macro *`#[reflect]`*. This macro replaces the function with a struct, and then implements `FnOnce` for that struct. The semantics of the function are the same before and after this transformation, but now that it is a struct, we can implement the Reflect trait.

To the user of this macro, they merely add the attribute macro above their function like so:

```
#[reflect]
fn is_a(c: char) -> bool {
    c == 'a'
}
```

But at compile time, the compiler will expand the macro into the following code:

```
#[allow(non_camel_case_types)]
#[allow(incorrect_ident_case)]
struct is_a;

impl FnOnce<(char,)> for is_a {
    type Output = bool;
    extern "rust-call" fn call_once(self, (c,): (char,)) -> bool {
        c == 'a'
    }
}
impl Reflect for is_a {
    fn reflect(&self) -> String {
        "fn is_a(c : char) -> bool { c == \'a\' }".to_owned()
    }
}
```

Notice that because we implement `FnOnce` for `is_a`, it can still be called as a function in the same way as before:

```
println!("{}", is_a('b')); // false
```

However now we are able to read the body of the function at runtime, thanks to the automatic implementation of `Reflect`.

There is one caveat to automatically implementing the `Reflect` trait for functions. The user may want to implement a function that returns a function, to be applied to another value. An example is the parser `match_char('a').map(append).ap(pure(vec![]))`. This parser matches the character `'a'`, converts it into a function that appends `'a'` to a `Vec`, and then applies it to an empty vec. In other words, it parses a singleton list of the character `'a'`. In Rust, the most natural way to implement `append` is a function like the following:

```
fn append<A: 'static>(a: A) -> impl FnOnce(Vec<A>) -> Vec<A> {
    move |mut v| {
        v.push(a);
        v
    }
}
```

The `impl FnOnce(Vec<A>) -> Vec<A>` signifies that we will return a type that implements a function in the form `Vec<A> -> Vec<A>`. This is because Rust does not allow naming the concrete type of closures. The Rust compiler will figure out what the concrete type of the closure is internally, and use that to resolve the `impl` return type. However, if we look at the macro-expansion for the `FnOnce` trait implemented on this function by `##[reflect]` we see the following:

```
impl<A: 'static> FnOnce<(A,)> for append {
    type Output = impl FnOnce(Vec<A>) -> Vec<A>; // impl in the associated type
    extern "rust-call" fn call_once(self, (a,): (A,)) -> Self::Output {
        move |mut v| {
            v.push(a);
            v
        }
    }
}
```

Unfortunately, Rust does not yet allow `impl` occurring in the associated type of a trait implementation. There are two options for resolving this issue:

1. Use dynamic dispatch and heap-allocate the return function.

2. Use the unstable `impl_trait_in_assoc_type`.

4.4.1 Using Dynamic Dispatch

With option 1, the user would have to reimplement the `append()` function to look like this:

```
fn append<A: 'static>(a: A) -> Box<dyn FnOnce(Vec<A>) -> Vec<A>> {
    Box::new(move |mut v| {
        v.push(a);
        v
    })
}
```

Because dynamic dispatch is used on a heap-allocated value, we have transformed the returned opaque type into a concrete type. The type `Box<dyn FnOnce(Vec<A>) -> Vec<A>>` can be used without requiring the use of a new unstable language feature. This is not without cost however, as Rust does not yet support whole program devirtualisation, the implementation of the returned function cannot be optimised by the Rust compiler. It would not be acceptable for Parslers to force users to use dynamic dispatch, as it would violate the performance guarantee. In order to account for this, the `##[reflect]` macro also comes with the ability to unbox a value in the code-generation step. The way to activate this feature is to write `##[reflect(unbox)]` instead. After macro-expansion with this feature, `##[reflect]` generates:

```
impl<A: 'static> FnOnce<(A,)> for append {
    type Output = Box<dyn FnOnce(Vec<A>) -> Vec<A>>;
    extern "rust-call" fn call_once(self, (a,): (A,)) -> Self::Output {
        Box::new(move |mut v| {
            v.push(a);
            v
        })
    }
}
impl Reflect for append {
    fn name(&self) -> &'static str {
        "append"
    }
    fn reflect(&self) -> String {
        "fn append<A:'static>(a : A,) -> impl FnOnce(Vec<A>) -> Vec<A> {
            move | mut v | { v.push(a) ; v }
        }".to_owned()
    }
}
```

Notice that in the implementation of `FnOnce`, the associated type `Output` still remains `Box<dyn FnOnce(Vec<A>) -> Vec<A>>`. However, the implementation of `Reflect::reflect` the generated code has removed the `Box<dyn ...>` type and replaced it with `impl ...`, and removed the box in the function body. This means that, with the feature enabled, the generated Rust code will use static dispatch instead of dynamic dispatch. This resolves Parslers' performance requirement.

4.4.2 Returning to Static Dispatch

Although the `##[reflect(unbox)]` macro works for many cases, it is not perfect. For example, it is completely feasible for the user to write a parser like:

```
#[reflect(unbox)]
fn f1(c: char) -> Box<dyn FnOnce() -> char> {
    Box::new(move || c)
}
#[reflect]
fn f2(f: Box<dyn FnOnce() -> char>) -> char {
    f()
}
fn parser() -> impl Parsler<Output = char> {
    pure('a').map(f1).map(f2)
}
```

As the user writes this parser, the Rust type system sees that `f1` produces a boxed function, and `f2` consumes a boxed function. However, the generated code of `f1` will generate an unboxed closure. This will cause the parser to fail its type check.

The alternative to unboxing the type at the code-generation step is to use the compiler feature `impl_trait_in_assoc_type` . This feature expands Rust's type inference capabilities to resolve opaque types from their occurrences elsewhere in the implementation. With this compiler feature enabled, we can return unboxed closures:

```
#[reflect]
pub fn append<A: 'static>(a: A) -> impl FnOnce(Vec<A>) -> Vec<A> {
    move |mut v| {
        v.push(a);
        v
    }
}
// Macro expanded code:
pub struct append;
impl<A: 'static> FnOnce<(A,)> for append {
    type Output = impl FnOnce(Vec<A>) -> Vec<A>; // Opaque type declared
    extern "rust-call" fn call_once(self, (a,): (A,)) -> Self::Output {
        // Opaque type resolved
        move |mut v| {
            v.push(a);
            v
        }
    }
}
```

4.5 Lazy Parsers

As discussed in the section Requirements, one of Parslers main goals is ensuring that users cannot write a malformed parser at the compile-time of the build.rs file. The section Parser Definition

Interface introduces building parsers as typed structures instead of higher-order functions. This is for the purpose of translating the parser into native Rust code, which would not be possible by composing higher-order functions. However, representing parsers as typed structures poses a new problem of representing recursion within these structures. This section introduces one of the parser combinators used to achieve recursion in parsers, which will be expanded upon by Named Parsers.

If the user attempts to create the following parser:

⟨*some_a*⟩ ::= 'a' ⟨*some_a*⟩
| 'a'

That is, to parse at least one 'a' character, then a parser like this would seem reasonable:

```
fn some_a() -> impl Parsler<Output = ()> {
    match_char('a').then(opt(some_a()))
}
```

However, the Rust compiler will throw the error:

```
warning: function cannot return without recursing
   --> parslers-benchmark/build.rs:153:1
    |
153 | fn some_a() -> impl Parsler<Output = ()> {
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot return without recursing
154 |     match_char('a').then(opt(some_a()))
    |                           -------- recursive call site
    |
    = help: a 'loop' may express intention better if this is on purpose
    = note: '#[warn(unconditional_recursion)]' on by default

error[E0720]: cannot resolve opaque type
   --> parslers-benchmark/build.rs:153:16
    |
153 | fn some_a() -> impl Parsler<Output = ()> {
    |                ^^^^^^^^^^^^^^^^^^^^^^^^^ recursive opaque type
154 |     match_char('a').then(opt(some_a()))
```

There are two problems here. Firstly, as the Rust compiler correctly points out, the function recurses infinitely. This is because our parsers are purely structural, and so calling this function will cause it to recurse infinitely. We are also posed with the problem that Rust's type system cannot handle infinite recursion, which is the second error message. Without some help, the user will not be able to write recursive parsers.

In order to fix the first problem, infinite recursion, we introduce the `LazyParser`. It has a simple definition:

```rust
pub struct LazyParser<P, F>(std::cell::OnceCell<P>, F)
where
    P: Parsler,
    F: FnOnce() -> P + Clone;
```

`LazyParser` a structure containing two values, a `OnceCell`, and a function, `F`. The `OnceCell` is a value that can be created as uninitialised, and then initialised with a function at a later time. `F` is a function that produces a parser. This way, we have a way of delaying initializing a parser until we need to use it. As we can see in its implementation of the `Parsler` trait:

```rust
impl<P: Parsler, F: FnOnce() -> P + Clone> Parsler for LazyParser<P, F> {
    type Output = P::Output;

    fn parse(&self, input: &mut Chars) -> Result<Self::Output, String> {
        self.0.get_or_init(self.1.clone()).parse(input)
    }

    fn compile(&self, info: &mut CompileContext) -> ast::Parser {
        self.0.get_or_init(self.1.clone()).compile(info)
    }
}
```

We can simply initialise the parser as of when we need it using the `get_or_init()` function. This solves the first problem, as we lazily initialize our parsers. However, there is still the problem of infinite type recursion, because the full type of the underlying parser behind `LazyParser` is still visible in `P`. To solve this, we use Rust's polymorphism:

```rust
pub fn dynamic<P>(p: P) -> Rc<dyn Parsler<Output = P::Output>>
where
    P: Parsler + 'static,
{
    Rc::new(p)
}
```

`Rc<dyn Parsler<Output = P::Output>>` is a reference-counted smart pointer to an object that implements `Parsler` and whose output is the same as the input parser. This maintains the type information we need while erasing the true type of the parser. Using these two concepts, we can rewrite the user's original parser into:

```rust
fn some_a() -> impl Parsler<Output = ()> {
    match_char('a').then(opt(dynamic(LazyParser::new(some_a))))
}
```

This is enough to get our parser to compile. However, there is one more step needed to achieve recursive parsing, which will be the topic of the next section. Also note that because Parslers uses staged compilation, using polymorphism and smart pointers has no effect on the performance of the resulting parser. These will be erased in the final parser into functions and called by name.

4.6 Named Parsers

The Lazy Parsers section introduces the first tool that can be used to generate recursive parsers by masking the concrete type of the parser behind dynamic dispatch. However, one more Looking back to the implementation of `LazyParser`, notice that both the `parse()` and `compile()` functions call the respective methods on the wrapped parser. For `parse()`, this is fine because the underlying parser may fail, and not recurse further. If the user writes a left-recursive parser, for example:

```
<some_a> ::= <some_a> "a"
```

Then the parser will recurse infinitely. However, this is expected of all top-down parsers, and so is not a problem. However, the `compile()` function will never fail if the parser is recursive, and so we are posed with the problem of identifying parsers that have already been compiled. In the original Staged Selective Parser Combinators paper (Willis, Wu & Pickering 2020), the author makes use of Haskell's **`StableNames`** to identify parsers that are equal. As Rust has no equivalence of **`StableNames`**, naming parsers manually is the necessary fallback. And so a new parser is introduced, the `NamedParser`:

```
pub struct NamedParser<P: Parsler>(pub P, pub String);
```

The `String` acts as an identifier that can be used to catch otherwise infinitely recursive calls to `compile()`:

```
impl<P: Parsler> Parsler for NamedParser<P> {
    type Output = P::Output;
    //...
    fn compile(&self, info: &mut CompileContext) -> ast::Parser {
        let hash = hash(self);
        let name = format!("{}_{}", self.1, hash);
        match info.register_parser(&name) {
            NamedParserStatus::Registered => ast::Parser::Ident(name),
            NamedParserStatus::Unregistered => {
                let p = self.0.compile(info);
                info.insert_parser::<P>(&name, p.clone());
                ast::Parser::Ident(name)
            }
        }
    }
}
```

Notice that the parser is also hashed to extend the identifier. We do this because manually naming parsers poses the risk of the user giving two different parsers the same name. Hashing the parsers

adds an extra safeguard against this. The call to `info.register_parser()` registers the parser's name, and then returns if was previously registered. If it was previously registered, then we just reference the parser's name. If it has not been seen before, then we compile the wrapped parser, insert it into the `CompileContext`, and then return the parser's name. This way, we catch potential infinite recursion in the compilation of self-referential parsers.

Because naming a parser often implies recursion, the library contains a helper function `name`:

```
pub fn name<P: Parsler + 'static>(
    name: &str,
    f: impl FnOnce() -> P + Clone + 'static,
) -> Rc<dyn Parsler<Output = P::Output>> {
    Rc::new(NamedParser(LazyParser::new(f), name.to_owned()))
}
```

Finally, the recursive parser from before can be re-written as:

```
fn some_a() -> impl Parsler<Output = ()> + Clone {
    name("some_a", || match_char('a').then(opt(some_a())))
}
```

4.7 Looping Parsers

It is common for a parser combinator library to implement the `some` and `many` combinators. These are simple combinators to implement:

```
pub fn some_rev<P>(p: P) -> Rc<dyn Parsler<Output = Vec<P::Output>>>
where
    P: Parsler + Clone + 'static,
    P::Output: Clone + 'static + Reflect,
{
    name("some_rev", || p.clone().map(append).ap(many_rev(p)))
}

pub fn many_rev<P>(p: P) -> Rc<dyn Parsler<Output = Vec<P::Output>>>
where
    P: Parsler + Clone + 'static,
    P::Output: Clone + 'static + Reflect,
{
    name("many_rev", || some_rev(p).or(pure(vec![])))
}
```

As can be seen above, they both rely on mutual recursion. Many functional programming languages support tail-call optimisation, which identify and replace recursion with imperative loops (Muchnick et al. 1997). Unfortunately, Rust does not officially support tail-call optimisation yet. As a result,

these two parsers will mutually recurse, which will incur the costs associated with recursion. Furthermore, `Vec` (equivalent to `ArrayList` in Java) grows from the end, making appending to the beginning of the vector expensive. Therefore, `some_rev()` and `many_rev()` will give the resulting parsed values in reverse-order from their parse order, and will require the user to reverse the list afterwards. This makes having a looping combinator necessary for writing performant parsers.

In order to accommodate for the need of loops within the parser, we introduce the `Loop` parser:

```
pub struct Loop<F, P0, PN>(P0, PN)
where
    F: FnOnce(P0::Output) -> P0::Output,
    P0: Parsler,
    PN: Parsler<Output = F>;
```

This is a combinator that takes two parsers:

`P0` : A parser which produces the initial value.

`PN` : A parser which will run in a loop, and produces a function that transforms the output of previous iterations.

This becomes more clear when looking at the implementation of the parser:

```
impl<F, P0, PN> Parsler for Loop<F, P0, PN>
where
    F: FnOnce(P0::Output) -> P0::Output,
    P0: Parsler,
    PN: Parsler<Output = F>,
{
    type Output = P0::Output;

    fn parse(&self, input: &mut Chars) -> Result<Self::Output, String> {
        let mut v = self.0.parse(input)?;
        while let Ok(f) = self.1.parse(input) {
            v = f(v);
        }
        Ok(v)
    }
    // ...
}
```

Implementing the `some()` and `many()` combinators then becomes simple:

```
pub fn some<P>(p: P) -> Rc<dyn Parsler<Output = Vec<P::Output>>>
where
    P: Parsler + Clone + 'static,
    P::Output: Clone + 'static,
```

```
{
    dynamic(Loop(p.clone().map(singleton), pure(append).ap(p)))
}

pub fn many<P>(p: P) -> Rc<dyn Parsler<Output = Vec<P::Output>>>
where
    P: Parsler + Clone + 'static,
    P::Output: Clone + 'static + Reflect,
{
    dynamic(Loop(pure(vec![]), Map(p, append)))
}
```

The only difference now is that the recursion has been removed from the implementation of the combinators. Note that the generality of the `Loop` operator allows writing any parser. For example, we can write a parser that counts the occurrences of `'a'` in a string:

```
fn count_as() -> impl Parsler<Output = usize> {
    Loop(pure(0), match_char('a').then(pure(inc)))
}
```

4.8 Alternative Parser

4.9 Untyped AST

Parslers' backend requires optimisations on the user-written parser's structure during the runtime of the build.rs module. This is done by conditionally matching on the structure of the parse tree, and applying substitutions that would improve the performance of the parser. As Rust follows the functional programming paradigm, the best way to perform this structural analysis would be via pattern matching. Unfortunately, there are several things preventing pattern matching from being used in this context.

Recall that, in its current form, the user writes parsers by combining typed structs together. This approach offers the great benefit that the user cannot write malformed parsers at compile time. Looking at the implementation of the `Or` combinator provides a concrete example:

```
pub struct Or<P1: Parsler, P2: Parsler<Output = P1::Output>>(pub P1, pub P2);
```

As we can see, there is a compile-time assertion in the definition of `Or` that the outputs of `P1` and `P2` (that is, the two alternative parsers) are the same. This would prevent a user from writing a parser that could produce two different types if `P1` succeeded or if `P2` succeeded.

However, the downside to this compile-time guarantee is that we lose the ability to perform structural pattern-matching on the parsers, as pattern matching can only be performed on enums. Furthermore, combinators such as `LazyParser` and `dyn Parsler` are not inspectable at runtime for pattern matching, as the internal structures are opaque. Furthermore, using typed enums is not possible as Rust does not currently support higher-kinded types, which would enable generic types to be defined in the individual enumerations. Although using Rust's Generic Associated Types could

allow for this, experiments performed in the writing of Parslers found that this method generates too much complexity on the side of the user.

The key insight to solving this issue is to restrict optimisations performed on parsers such that they are type-invariant. That is to say that no optimisations performed on a parser can change the output type of the parser. It will become clear that this is more of an observation of the current optimisations available rather than a limitation of the optimisations we can select from in later chapters. With this observation, we can begin to erase the types of the parsers at runtime and translate the parsers into an untyped AST using Rust's enums.

```
pub struct AnalysedParser {
    pub output_used: bool,
    pub returns_func: bool,
    pub parser: Parser,
}
pub enum Parser {
    Ident(String),
    Pure(PureVal),
    Satisfy(Func),
    Try(Box<AnalysedParser>),
    Look(Box<AnalysedParser>),
    NegLook(Box<AnalysedParser>),
    Ap(Box<AnalysedParser>, Box<AnalysedParser>),
    Then(Box<AnalysedParser>, Box<AnalysedParser>),
    Before(Box<AnalysedParser>, Box<AnalysedParser>),
    Or(Box<AnalysedParser>, Box<AnalysedParser>),
    Recognise(Box<AnalysedParser>),
    Empty,
    Branch(
        Box<AnalysedParser>,
        Box<AnalysedParser>,
        Box<AnalysedParser>,
    ),
    Loop(Box<AnalysedParser>, Box<AnalysedParser>),
}
```

Translation from a typed structure to an untyped AST is the main purpose of the `Parsler::compile` method. Now that we have an untyped AST in the form of a Rust enum, we may perform structural optimisations on the parser. Note that this is bundled as a part of the backend of Parslers, and should never be a concern of the library user.

As this method requires optimisations on the `Parser` enum to be type-invariant, the library maintainer is now responsible for ensuring that this invariance propagates throughout all optimisation passes. However, this has not yet been a problem in the writing of Parslers.

4.10 Backend Optimisations

Optimisation plays a key part in Parslers' performance characteristics. In its current state, Parslers performs two major optimisations passes: Lawful Reduction, and Usage Analysis.

4.10.1 Lawful Reduction

Research done on parser combinators show that the combinators that form parsers are a type of algebra (Willis, Wu & Pickering 2020), and have equivalence laws. Below are the laws that apply to the combinators presented in Parslers:

$$Ap(Pure(f),Pure(x))=Pure(f(x)) \tag{1}$$

$$Ap(u,Pure(x))=Ap(Pure(|f|f(x)),u) \tag{2}$$

$$Ap(u,Ap(v,w))=Ap(Pure(compose),Ap(u,Ap(v,w))) \tag{3}$$

$$Or(Or(p,q),r)=Or(p,Or(q,r)) \tag{4}$$

$$Or(Empty,p)=Or(p,Empty)=p \tag{5}$$

$$Ap(Empty,p)=Empty \tag{6}$$

$$Or(Pure(x),p)=Pure(x) \tag{7}$$

$$Branch(Pure(Left(x)),p,q)=Ap(p,Pure(x)) \tag{8}$$

$$Branch(Pure(Right(y)),p,q)=Ap(q,Pure(y)) \tag{9}$$

$$Branch(Then(x,y),p,q)=Then(x,Branch(y,p,q)) \tag{10}$$

$$Try(NegLook(p))=NegLook(p) \tag{11}$$

$$Look(Empty)=Empty \tag{12}$$

$$Look(Pure(x))=Pure(x) \tag{13}$$

$$NegLook(Empty)=Pure(()) \tag{14}$$

$$NegLook(Pure(x))=Empty \tag{15}$$

$$Look(Look(p))=Look(p) \tag{16}$$

$$Or(Look(p),Look(q))=Look(Or(Try(p),q)) \tag{17}$$

$$NegLook(NegLook(p))=Look(p) \tag{18}$$

$$NegLook(Or(Try(p),q))=Then(NegLook(p),NegLook(q)) \tag{19}$$

$$Or(NegLook(p),NegLook(q))=NegLook(Then(Look(p),Look(q))) \tag{20}$$

The laws stated above allow Parslers to optimise user-written parsers to a great degree. Take, for example, the following parser:

```rust
#[reflect]
fn is_a(c: char) -> bool { c == 'a' }
#[reflect]
fn neg(b: bool) -> bool { !b }
pure('a').map(is_a).map(neg) // Always returns false
```

When written out as an AST, it becomes `Ap(neg, Ap(Pure(is_a), Pure('a')))`. Using the lawful reduction rules stated above, the AST can be reduced:

```
Ap(neg, Ap(Pure(is_a), Pure('a')))

Ap(neg, Pure(is_a('a')))

Pure(neg(is_a('a')))
```

Parslers performs the lawful reduction optimisation in a bottom-up recursive pass through the parser AST, applying substitutions where possible and then retrying if a match has been found (to ensure any new substitution found after applying a reduction is checked).

4.10.2 Output Usage Analysis

Usage analysis is the most powerful optimisation that Parslers has in its toolkit. It relies on the observation that certain combinators discard the returned value after parsing has completed. Take a look at the following parser:

```rust
#[reflect]
fn digit(c: char) -> bool { ('0'..='9').contains(&c) }
#[reflect]
fn parse_usize(s: String) -> usize { s.parse().unwrap() }
fn uint() -> impl Parsler<Output = usize> {
    Recognise(some(Satisfy(digit))).map(parse_usize)
}
```

When broken down, `some(Satisfy(digit))` returns a `Vec<char>` of at least one, and then `Recognise(some(digit))` discards the output of `some(Satisfy(digit))`, and returns its consumed string. This string is then parsed into a `usize`. That `Vec<char>` requires allocation, reallocation (when the `Vec` grows), and the pushes to the `Vec` after each successful digit parsed. These clock cycles are wasted with no use. In some cases, the compiler recognises this and can discard wasted computation, but this is not perfect, as will be seen in the later chapter discussing performance benchmarks.

In order to account for this, Parslers incorporates usage analysis. This process performs a top-down pass through the untyped parser AST, and marks subtrees as unused if not used by their parents. During code-generation, if a parser subtree's output is unused, it will return the unit struct (`()`, in Rust). Because this happens all the way down the parse tree, `some(Satisfy(digit))` does not return any value, and we will see that this can have a great effect in the next section.

The `Branch(b, l, r)` combinator is the only one which prevents usage analysis from propagating into the combinator subtrees. Because `l` and `r` are conditionally executed based upon the result of `b`, `b` must always be returned.

```
Then(p, q)      // p is discarded.
Before(p, q)    // q is discarded.
Recognise(p)    // p is discarded.
Branch(b, l, r) // b is always used.
```

4.10.3 Validation Parsers

An interesting side effect of the usage analysis optimisation can be found in validating parsers. If a user has a very large parser (for example, a JSON parser), but instead of wanting to parse text into structured data just validate that the text is a valid instance of the grammar. In this case, it suffices to simply augment the parser `p` into `p.then( pure(()) )`. The usage analysis optimiser will see that the result of `p` is discarded, and deeply remove any computation and returned values. As can be seen in the benchmarking chapter, this provides a very large boost to performance.

4.10.4 Analysis of Recursive Parsers

In order for the benefits of analysis optimisation to be fully realised, it must be aggressive. Given a parser that is both recursive and unused (for example, a JSON validating parser), usage analysis must also identify recursion points and mark them to be revisited by the optimiser. Recall that recursive parsers are marked via a named hole. When it is identified that the result of this named parser is unused, the recursion point is renamed with a tag that marks it is unused, and the original parser is added to a set of parsers that are to be deeply optimised to reflect that the return value is

unused.

Optimisation on the renamed, unused parser may result in more named parsers being marked unused. Because of this, the optimisation is run repeatedly until there are no more named parser marked unused that do not already exist in the set of optimised parsers. The optimiser is guaranteed to terminate as there can only be a single unused variant of a parser for each of the named parsers, and so the loop is bounded.

4.11 Code Generation

The Untyped AST section introduces the intermediate representation of parsers as a Rust enum, which reflect the primitive combinators but strips the type safety that structs give us. Then, the Backend Optimisations details the optimisations that the untyped AST can take advantage of. This section concerns itself with how Parslers goes from the untyped AST back into native Rust code, which completes the loop.

To begin with, Parsing and Generating Rust Code with Syn and Quote will introduce the library that helps tokenise Rust code using a macro library called `quote`. Then, The type of Parsers will introduce how function definitions can be derived from user-generated parsers in the code-generation step.

4.11.1 Registering Functions

The section The Reflect Trait showed how Parslers achieves runtime reflection of function definitions using the `Reflect` trait and the `#[reflect]` macro. These functions must be added to the compile context so that they may be compiled into the native Rust code. Looking at the `CompileContext` struct, which was previously seen in the `Parsler::compile(&self, info: &mut CompileContext)` function:

```
pub struct CompileContext {
    functions: HashMap<String, String>,
    // ...
}
```

The `functions` field is a `HashMap<String, String>`. This stores the bodies of the functions, along with a generated name. The name is replaced and auto-generated as it is only the body of the function that needs to be unique, and Parslers is able to de-duplicate functions when they arise.

The `functions` field is only modified by three of the primitive combinators:

- `Pure`
- `Map`
- `Satisfy`

Now the usage in `Satisfy` and `Map` is not difficult, as both of these primitive combinators guarantee that they are constructed with a function type. Therefore, registering the function into the compile context is simple:

```rust
impl<B, P1, F> Parsler for Map<B, P1, F>
where
    P1: Parsler,
    F: FnOnce(P1::Output) -> B + Clone + Reflect,
{
    type Output = B;
    fn compile(&self, info: &mut CompileContext) -> ast::Parser {
        let name = info.add_function(&self.1);
        let p = self.0.compile(info);

        ast::Parser::Ap(
            Box::new(AnalysedParser::new(p)),
            Box::new(AnalysedParser::new(ast::Parser::Pure(ast::PureVal::Func(
                ast::Func { name },
            )))),
        )
    }
    // ...
}

impl<F: FnOnce(char) -> bool + Reflect + Clone> Parsler for Satisfy<F> {
    type Output = char;

    fn compile(&self, info: &mut CompileContext) -> ast::Parser {
        let name = info.add_function(&self.0);

        ast::Parser::Satisfy(ast::Func { name })
    }
    // ...
}
```

In these, as they both are restricted by `F: FnOnce(...) -> ...`, using `info.add_function(...)` is always safe. However, this is not the case with the `Pure` primitive combinator, because it does not restrict the type of value it is. This becomes clear when looking at `Pure`'s definition:

```rust
pub struct Pure<A: Reflect>(pub A);
```

The fact that `Pure` can accept any value that implements `Reflect` means that a bit more logic is required to differentiate functions from values. To solve this, Parslers uses the `syn` crate to attempt to parse the reflected value held in `Pure` into a `ItemFn`. If `syn` can successfully parse the string returned by `Reflect::reflect(&self)` into a function, then it is safe to assume that it is a function, and can be registered the same way as in the implementations of the other two primitive combinators:

```rust
impl<A: Clone + Reflect> Parsler for Pure<A> {
    type Output = A;

```

```
    fn compile(&self, info: &mut CompileContext) -> ast::Parser {
        if let Ok(_) = syn::parse_str::<syn::ItemFn>(&self.0.reflect()) {
            let name = info.add_function(&self.0);
            ast::Parser::Pure(ast::PureVal::Func(ast::Func { name }))
        } else {
            ast::Parser::Pure(ast::PureVal::Val(self.0.reflect()))
        }
    }
    // ...
}
```

Now Parslers is able to register every function that can be found in the user-generated parsers.

The `CompileContext::add_function(&mut self, func: &impl Reflect)` method is incredibly simple thanks to Rust's `Entry` API:

```
impl CompileContext {
    pub fn add_function(&mut self, func: &impl Reflect) -> String {
        let num_functions = self.functions.len();
        self.functions
            .entry(func.reflect())
            .or_insert(format!("f{}", num_functions))
            .clone()
    }
    // ...
}
```

This method gets the current number of functions registered (so that each unique function has a unique name) and then attempts to retrieve the unique function name from `functions` map. If this entry does not already exist, it inserts a new name with the number of registered functions and returns the new name.

During the generation of the native Rust code, the keys and values of `functions` are iterated over, and the previous name of the function body is replaced with the generated unique name. Then, the `quote` library is used to insert each of these functions into the generated Rust module. These functions can then be referenced and called within the generated parser at run-time. Both the `syn` and `quote` libraries will be discussed in the section Parsing and Generating Rust Code with Syn and Quote.

4.11.2 Parsing and Generating Rust Code with Syn and Quote

The section Registering Functions discussed how functions can survive the build.rs file into the generated binary, and expanded futher on the purpose of the section The Reflect Trait. This section concerns itself with two useful tools from the Rust ecosystem that allow Parslers to parse and generate Rust code, the `syn` and `quote` libraries respectively.

The `syn` library is used in several places in Parslers. This library is important to Rust library writers because of the way that Rust's staging works. In Rust, procedural macros are expanded after the parse stage of the compiler. This stage produces tokens in the form of a `TokenStream`, which hold

no semantic information about the code that has been parsed. Furthermore, `TokenStream` is not given in the form of a Rust AST, mainly because procedural macros allow library writers to create domain specific languages that do not necessarily conform to the Rust syntax standard (such as a HTML template in the `typed_html` library). However, in cases where the incoming `TokenStream` is fully compliant with the Rust syntax standards, just using `TokenStream` can be cumbersome. This is where the `syn` library comes in, which allows parsing `TokenStream`s into a Rust-compliant AST.

Although `syn` is used throughout the codebase of Parslers, the best example os its use is in the renaming of functions as described in Registering Functions:

```
let functions = context
        .functions
        .iter()
        .map(|(body, name)| {
            let mut body = syn::parse_str::<syn::ItemFn>(body).unwrap();

            body.sig.ident = syn::parse_str::<syn::Ident>(name).unwrap();

            // ...
        })
        .collect::<Vec<TokenStream>>();
```

In the above code, `functions` is iterated over (the keys and values are of type `String`). Then, the function body is parsed into a `syn::ItemFn`, which is a full function in the form:

```
fn func(x: i32) -> i32 {
    a + 1
}
```

Then, the new name of the function is parsed into a valid identifier. Now it becomes easy to rename the function by replacing the function signature's identifier with the new function name. The hidden code behind `// ...` is the next topic of interest, which converts the typed AST that `syn` provides back into a `TokenStream`.

The final step in code generation involves converting the untyped AST back into native Rust code. Currently, Rust does not have good first-class support for generating Rust code in an ergonomic manner. This could be solved by using string templating, but this is not ergonomic and puts too much cognitive load on the library programmer. To solve this, a macro library has been developed called `quote`. This is a simple and minimalist library with one core feature, the `quote!{...}` macro. Using this macro is quite simple.

For example, if a library developer wanted to develop a macro that generates a function to print a string, but in a shorter way than the standard `println!(...)`. To create a `pl!(...)` macro, it is quite simple:

```rust
#[proc_macro]
pub fn pl(input: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let input = proc_macro2::TokenStream::from(input);
    quote::quote!{
        println!(#input)
    }.into()
}
```

Here it can be seen on lines 4-6 that the quote macro is used to tokenize rust-like syntax. It also performs syntax analysis at the compile-time of the macro. This allows many of the potential syntax bugs that could arise in the string templating approach to code generation. In Parslers, the use of `quote!{...}` can be seen in the function generation:

```rust
let functions = context
        .functions
        .iter()
        .map(|(body, name)| {
            let mut body = syn::parse_str::<syn::ItemFn>(body).unwrap();

            body.sig.ident = syn::parse_str::<syn::Ident>(name).unwrap();

            quote! {
                #[inline(always)]
                #body
            }
        })
        .collect::<Vec<TokenStream>>();
```

Notice that `##[inline(always)]` is used here. This is because functions used in Parslers are oftentimes incredibly small, and inlining these functions can always be justified. It may be useful in the future to add inlining hints to the `##[reflect]` attribute macro, however, inlining every function results in faster parsers as found in the benchmarks.

4.11.3 The type of Parsers

Now the tools used to parse and generate valid Rust syntax have been discussed in the previous section, it is time to move on to how function definitons can be derived from user-defined parsers.

This is important as user-defined parsers should be translated into functions that can be called on some object. It is sensible for the type of these functions to match the type of the `Parsler::parse()` function to ensure cross-compatibility. Therefore, the type of the output parsing function should be `fn(input: &mut std::str::Chars) -> Result<..., &'static str>`.

Determining what the `...` in the output result type should be considered carefully. This should require runtime reflection of the parsers' output types to correctly identify and insert it into the missing space. Luckily, the Rust language's powerful monomorphization can be used to circumvent adding even more runtime reflection into Parslers.

The Rust standard library comes pre-packaged with the `std::any` module. This is a module dedicated to compile-time reflection and makes use of monomorphization in order to achieve compile-time reflection instead of runtime reflection. This can then be used to retrieve the full type name of an object at compile-time.

`std::any::type_name<T>() -> &'static str` function does not take in any values, but instead a function generic that is used to retrieve the name of the type. When inserting a named parser into the `CompileContext`, Parslers uses an internal method `CompileContext::insert_parser<P: Parsler>(&mut self, name: &str, parser: ast::Parser)`. This method does not take the user-defined parser, but instead the untyped AST. However, it does take a generic parameter `P: Parsler`. This allows the type name of `P::Output` to be retrieved at compile-time, and be used to generate the true type of the output parser, filling in the `Result<..., &'static str>` result type signature in the generated parser function.

4.11.4 Recursively Compiling Parsers

The sections Registering Functions, Parsing and Generating Rust Code with Syn and Quote, and The type of Parsers have laid out the building blocks to converting user-defined parsers into native Rust code. However, there is one more step required before this process is complete - generating the parser bodies themselves.

The code generation is performed via a top-down post-order traversal of the untyped parser AST. It is worth noting that compilation is split into two different functions: `compile_used()`, and `compile_unused()`. This is to account for the fact that the generated output code looks very different for parser combinators whose outputs are used and unused. Looking at the implementation of some of the primitive combinators shows that thanks to Rust's pattern-matching syntax, this stage of compilation is easy to perform.

Firstly, see the `Satisfy` variant of the untyped parser AST:

```
Parser::Satisfy(ident) => {
    let ident = syn::parse_str::<syn::Expr>(&ident.name).unwrap();
    quote! {
        let old_input = input.clone();
        input.next()
        .ok_or("Found EOF when character was expected")
        .and_then(|c|
            if (#ident)(c) {
                Ok(c)
            } else {
                *input = old_input;
                Err("Expected a specific character")
            }
        )
    }
}
```

Notice two things:

1. The code within the `quote!{...}` macro looks very close to the `Parsler::parse(...)` method in the implementation of `Parsler` for `Satisfy`. This is frequently the case and

expected considering that the output generated code is Rust code.

2. `input` is never defined in the generated code but is used. This is possible because the whole generated parser mutates `input`, and every sub-parser is nested in a new scope level. Therefore, it is always valid to assume that `input` exists and is correct within its context so long as backtracking parsers handle the logic correctly.

Now the code generation logic for the `Satisfy` variant has been introduced, it is worth looking at the `compile_unused()` function to see where differences typically lie:

```
Parser::Satisfy(ident) => {
    let ident = syn::parse_str::<syn::Expr>(&ident.name).unwrap();
    quote! {
        let old_input = input.clone();
        input.next()
        .ok_or("Found EOF when character was expected")
        .and_then(|c|
            if (#ident)(c) {
                Ok(())
            } else {
                *input = old_input;
                Err("expected a specific character")
            }
        )
    }
}
```

Notice how `Ok(c)` has now become `Ok( () )`. As mentioned in Output Usage Analysis, this is an optimisation that allows for significant performance improvements. This will be expanded upon in the evaluation section.

5 EVALUATION

This section concerns itself with the performance evaluation of Parslers as a general-purpose parsing library.

5.1 Performance

5.1.1 Prior Discussion

Evaluating the performance of any general-purpose library is a problematic pursuit. Especially when benchmarks must be comparative to other libraries. This is because a user could implement a program in many ways within each library. Parser combinator libraries are no exception to this, as there are usually many different parsers that can conform to the same grammar, each with different performance characteristics.

Although great care was taken to be fair to each parser combinator library in each comparative benchmark, I am not an expert in the performance characteristics of these libraries. However, having written Parslers myself, I am an expert in its performance characteristics. For this reason, the Branflakes comparative benchmarks must not be confused with the fastest possible parsers in each library.

However, each of the libraries benchmarked in the Comparative JSON Benchmarks section have JSON parsers included with a benchmarking suite. These premade parsers were used and can be assumed to be strong comparative benchmarking platforms.

All benchmarks were run on the same machine with an Intel i5 4060k 3.5 GHz CPU and 22 GB of DDR3 1600 MHz RAM.

5.1.2 Comparative JSON Benchmarks

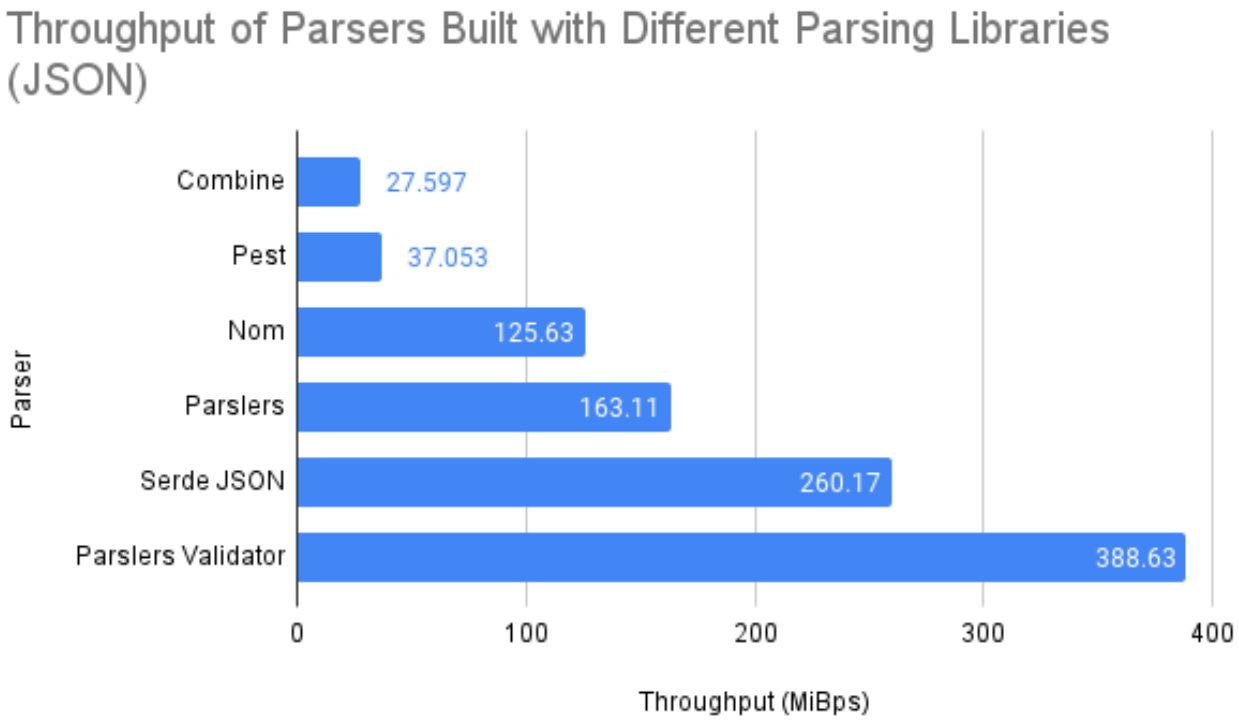


Figure 1. A plot of the throughput of parsers built with Parslers, Nom, Combine, Pest, and Serde JSON each provided by the benchmark suite of the respective library, benchmarked on the Canada JSON dataset.

JSON (JavaScript Object Notation) is one of the most-used formats for machine-to-machine com-

munication. It is commonly used as a part of client-server communication in Web APIs. This is because the grammar is quite simple, but expressive (Pezoa, Reutter, Suarez, Ugarte & Vrgoč 2016) and already has large-scale adoption and language support. This makes it a great candidate for benchmarking, and many general-purpose

In this benchmark, Parslers is put against Rust's most commonly used general-purpose parsing libraries (Nom, Pest, and Combine), and the most commonly used hand-written parser for JSON (Serde JSON). Nom is generally considered the fastest general-purpose parsing library for the Rust programming language and is also a parser combinator library. As can be seen in 1, The Parslers library outperforms the general-purpose parsing libraries but does not outperform Serde JSON. This is reasonable, as Serde JSON is a hand-written library dedicated to parsing a specific grammar, and is heavily optimised to parse as fast as possible. It is worth noting, however, that the Parslers Validate benchmark does outperform Serde JSON, which does not currently have a way to validate JSON files rather than just parse them into structured JSON objects.

5.1.3 Comparative Branflakes Benchmarks

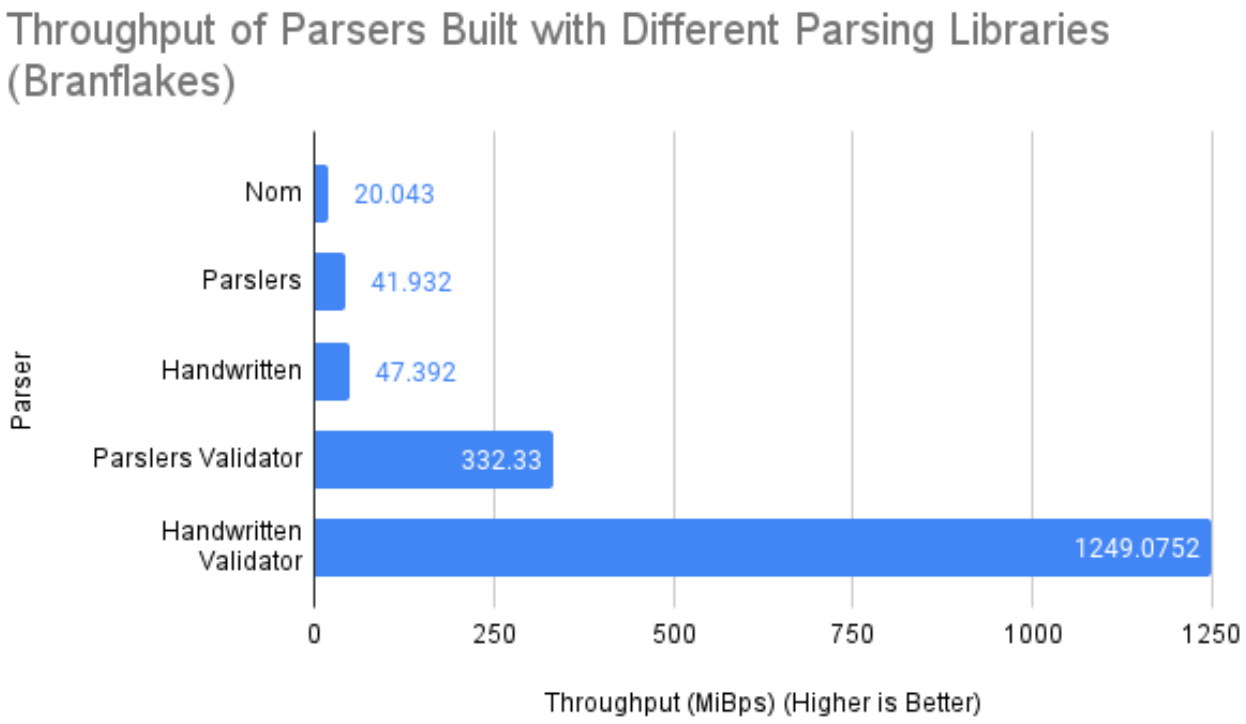


Figure 2. A plot of the throughput of parsers built with Parslers, Nom, a hand-written parser, and a hand-written validator each provided by the benchmark suite of the respective library, benchmarked on the Lost Kingdom Branflakes dataset.

Branflakes is a simple Turing-complete programming language. It is composed of an unsigned, 8-bit-valued tape memory, and a pointer to the current tape location. Values on this tape can be manipulated with eight total symbols:

+ Increment the value at the address of the current tape pointer by 1.

- Decrement the value at the address of the current tape pointer by 1.

< Decrement the tape pointer by 1.

> Increment the tape pointer by 1.

. Print out the value at the address of the current tape pointer.

, Read a character and set the value at the address of the current tape pointer to it.

[If the value at the address of the current tape pointer is 0, then jump to the next matching].

] If the value at the address of the current tape pointer is 0, then jump to the preceding matching [.

The simplicity of the grammar makes it a good benchmarking tool for comparing different parsers. Furthermore, it is a great grammar for writing an optimised hand-written parser and validator. For this benchmark, a hand-written parser was developed, lacking any recursion and is stack-based to be performant. This serves as a good comparison between hand-written parsers and Parslers in the context of a simple grammar.

Looking at the results in 2, the results show that Parslers is faster than Nom in this context as well. The handwritten parser is faster than Parslers, but surprisingly only by about 13%. However, with the optimisations available by knowing the grammar, the handwritten validator is faster than Parslers' validator by about 376%. It is worth noting that the handwritten validator has to be written separately from the actual handwritten parser, which means there is extra overhead in writing a separate structured parser and validator. This overhead does not exist for Parslers.

5.1.4 Staging and Optimisation Benchmarks

Parslers is a staged parser combinator library. However, each of the primitive parser combinators packaged with Parslers has the ability to parse before staging occurs. This makes the question of how much of a performance increase can be found in staging alone without any further optimisations applied. A second element of interest is how much different optimisations improve the performance of resulting parsers, and whether or not parsers for different grammars will benefit more from certain optimisations or others.

In order to test this, benchmarks were set up to compare different optimisation configurations, to be tested on both Branflakes and JSON parsers. These include:

- Unstaged
- Staged, but Unoptimised
- Reduction Analysis Only
- Usage Analysis Only
- Fully Optimised

From both the JSON and Branflakes benchmarks in 4 and 3, it is clear that staging is hugely beneficial to the performance of parsers, regardless of applied optimisations. This makes sense when considering that compilation is split up into compilation units, which are compiled and optimised separately. However, after staging large sections of the parser exist within the same function, and will be optimised as a single compilation unit. Furthermore, there is no dynamic dispatch in the staged parsers, whereas the unstaged parsers can heavily use dynamic dispatch in order to fix problems discussed in Lazy Parsers and Named Parsers.

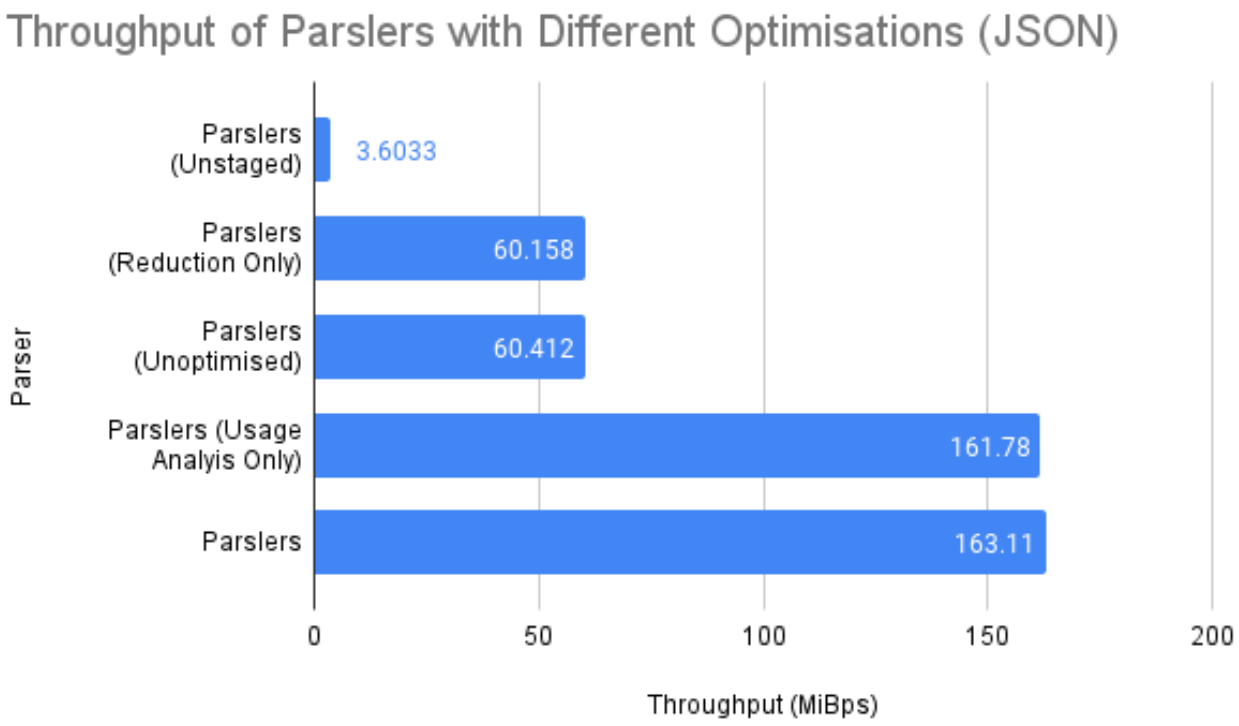


Figure 3. A plot of the throughput of parsers built with Parslers, each with different optimisation configurations, benchmarked on the Canada JSON dataset.

There is also a difference in which optimisation provides the largest performance benefit between the Branflakes parser and the JSON parser. In the Branflakes parser (see 4), reduction analysis provides the greatest benefit in performance improvement, but usage analysis makes no significant change to performance. On the other hand, the JSON parser (see 3) benefits greatly from usage analysis, but almost not at all from reduction analysis. This can be explained by the parsers written for the benchmarks having very different properties. For example, the Branflakes parser does not do any wasteful computation that can be optimised by the usage analysis optimiser but does have a more verbose structure that can be optimised using reduction analysis. However, the JSON parser does not change much during reduction analysis but does make heavy use of the `Recognise` parser, which can be heavily optimised using usage analysis.

5.1.5 Conclusion

Parslers clearly provides performance gains in comparison to the fastest general-purpose parsing library, nom. This shows that staging, and the opportunity for optimisations that staging provides, can help considerably close the gap between hand-written parsers and high-level general-purpose parsing libraries. Although Parslers is still not as fast as hand-written parsers (and may never be in the general case), there are still several opportunities for further optimisations available. These optimisations will be expanded upon in the Future Work section.

5.2 Expressive Power

Moving on to the discussion around the expressiveness of Parslers as a parser combinator library. As stated in 4.1, expressiveness in the context of Parslers means the range of grammars that can be expressed. As a part of the requirements, Parslers must be able to express any grammar expressed as a PEG grammar.

In order to demonstrate that Parslers is at least as expressive as PEG grammars, each of the avail-

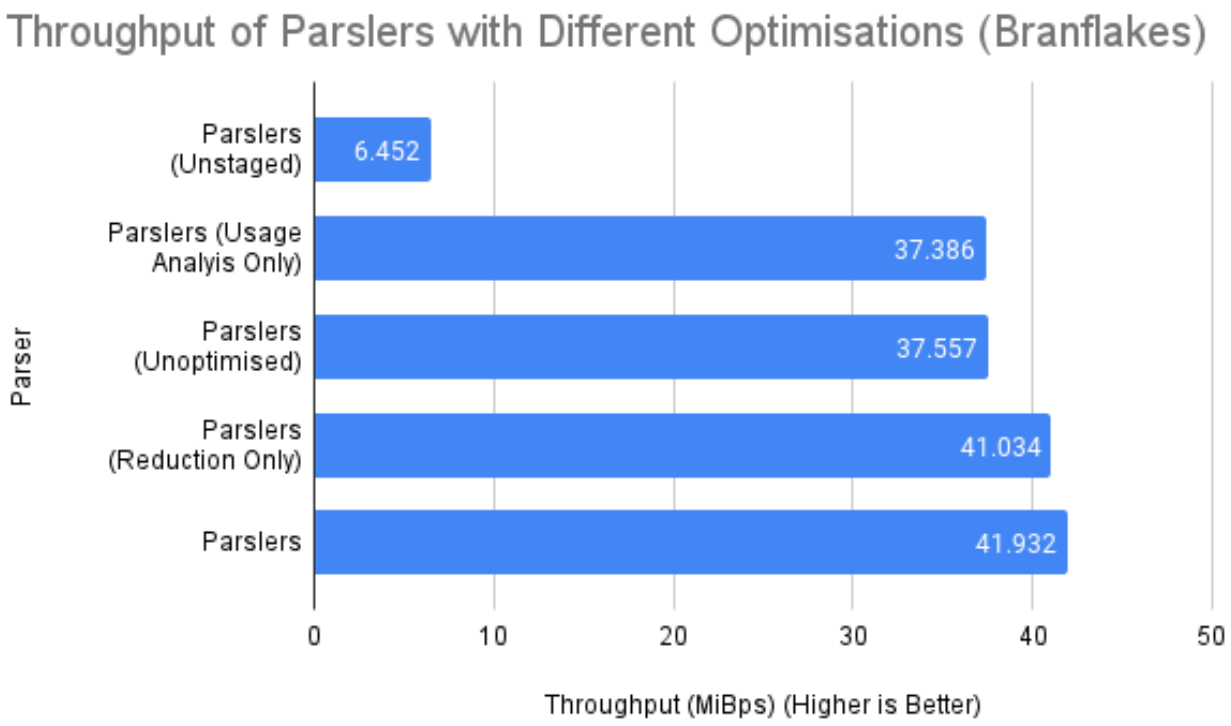


Figure 4. A plot of the throughput of parsers built with Parslers, each with different optimisation configurations, benchmarked on the Lost Kingdom Branflakes dataset.

able operators in PEG will be discussed, and a corresponding combinator will be given in Parslers, demonstrating the equivalence of the two.

It is worth noting that in PEG, failure always results in backtracking the whole consumed input. In Parslers, this is not implicit in order to improve performance. Backtracking upon failure must be explicitly added via the `Attempt` combinator.

Recall that PEG grammars follow the following operators are available:

5.2.1 Sequence

In PEG, the sequence operator takes two parsing expressions e_1 and e_2, and performs them in sequence in the form $e_1 e_2$. That is, e_2 is parsed if e_1 succeeds, and either e_1 or e_2 failing also results in the whole sequence failing.

In Parslers, sequencing can be replicated using either the `Before` or `Then` combinators. In this form, `e1.attempt().then(e2.attempt())` or `e1.attempt().before(e2.attempt())` are semantically identical to $e_1 e_2$.

5.2.2 Ordered Choice

In PEG, ordered choice is in the form e_1 / e_2. This operator attempts to parse e_1, but if e_1 fails then the consumed input is backtracked, and then e_2 is parsed.

In Parslers, backtracking is explicit except in the case of `Satisfy`, which backtracks upon failure. Although `Or` does fit, there is the caveat that it does not automatically backtrack upon the failure of the first parser. However, Parslers does have the `Attempt` combinator. Therefore, `e1.attempt().or(e2.attempt())` is equivalent to e_1 / e_2.

5.2.3 Zero/One-or-More

In PEG, zero-or-more and one-or-more are written in the form $e*$ and $e+$ respectively. They attempt to parse several matches of the parser e until failure is reached, never backtracking.

In Parslers, the section 4.7 covers these semantics and extensions made to Parslers to optimise these sorts of parsers. `Loop(pure(()), e.attempt())` and `Loop(e.attempt(), e.attempt())` are equivalent to zero-or-more and one-or-more operators respectively.

5.2.4 Optional

In PEG, the optional operator is written in the form $e?$, and will attempt to parse either zero or one occurrence of e in the input stream.

In Parslers, this is achievable by writing a combinator in the form `e.attempt().then(pure(())).or(pure(()))`. However, Parsers comes built-in with the `opt(e)` auxiliary combinator that is less verbose.

5.2.5 And/Not Predicates

The and predicate is an operator written $\&e$. The semantics are to parse e, but always to backtrack upon success or failure. This is look-ahead semantics. Similarly, the not predicate is written $!e$, and fails if parser e succeeds, never consuming input.

In Parslers, both of these are achievable via the primitive combinators `Look` and `NegLook`, and can be written as `e.look()` and `e.neg_look()` to represent and not operators respectively. These maintain the backtracking semantics found in both of these operators.

5.2.6 Conclusion

The above arguments show that an isomorphism exists between PEG and Parslers, as any PEG grammar can be converted into a Parslers parser by the steps shown above. This, therefore, implies that Parslers must be at least as expressive as PEG. However, the context sensitivity introduced by the selective combinator `Branch` extends the expressive power of Parslers beyond PEG. This requirement is satisfied, and the Future Work section will discuss how the expressive power of Parslers can be improved upon.

5.3 Correctness

The correctness of Parslers is also defined in 4.1, and requires that Parslers maintains the type safety of user-written parser before and after staging. It also requires that Parslers apply restrictions on the types of parsers to be malformed before staging - in this way, disallowing the user to write malformed parsers.

This section does not concern itself with the correctness of the compiler, as this would require formal verification that Rust does not currently support. However, given that the optimisations performed are lawful analysis or usage analysis, these do not change the type of the resulting parser or its semantics.

There is a caveat to the above statement. Part of usage analysis is removing unnecessary computation. To demonstrate how this may be a problem, take a look at the following parser:

```
use std::sync::atomic::*;
static atomic: AtomicUsize = AtomicUsize::new(0);

#[reflect]
fn to_u32(c: char) -> u32 {
```

```
    atomic.fetch_add(1, Ordering::SeqCst);
    c as u32
}

fn parser() -> impl Parsler<Output = u32> + Clone {
    match_char('a').map(to_u32)
}
```

As can be seen, the function `to_u32` mutates a global variable during execution. Under normal circumstances, this function will continue to be executed as the user expects. However, usage analysis on a similar parser, `parser().then(pure(()))` will remove the application of `to_u32` to the parsed character. Therefore, this function will never run.

Because of this, Parslers makes it a requirement that functions used in the creation of the Parser are pure and side-effect free. However, in Rust it is not yet possible to restrict functions to be pure apart from using the `const` keyword. Although using `const` is a viable solution, Rust considers most forms of allocation as non-const, and requires that functions that the user may call within a `const` function are also labelled `const` . This would become a major disadvantage to the usability of Parslers, and so was omitted. Therefore, it is possible that if functions produce side-effects, then the optimisations within Parslers may alter the behaviour of the program.

Apart from the caveat mentioned above, Parslers still requires that users write well-defined parsers at compile-time due to the type restrictions introduced by the `Parsler` trait (and its implementations over the primitive combinators). Therefore, Parslers can be considered correct in this perspective.

5.4 Usability

Parslers is a parser combinator library designed to be performant and easy to use. The core idea is to remove the burden of optimising parsers from the users while maintaining the type-safe Rust frontend.

There are a number of points that can be discussed here, which will be split up into subsections and discussed individually.

5.4.1 Staging

Staging is the heart of how Parslers can perform optimisations and native code generation at compile time. Without staging, Parslers could not be as performant as it is. However, staging does not come without issues of usability.

Firstly, staging requires some form of runtime reflection. This is because Parslers generates native Rust code, so pure values and referenced functions cannot make the jump from the build.rs file to the generated module without some way to serialize them and recreate them using Rust code. Although the `Reflect` trait, along with its macros make this possible, reflection does not yet exist for closures, as they cannot be inspected at compile time or at runtime. This means that Parslers can only accept a subset of the Rust programming language.

Furthermore, for larger parsers, the compile times can really bloat. One of the major advantages to the staging of parsers is that it restricts the number of code generation units the compiler can use,

giving more opportunity to potential optimisations being applied. However, this is at the cost of compile times - an issue the Rust compiler already faces. With sufficiently large parsers, this can cause a lot of problems for language servers used in code editors' incremental compile checks.

Finally, structures used within parsers must be declared in a separate crate. This is because the build.rs file is unable to access constructs defined in the crate's binary. Likewise, it does not make sense to define these structures directly in the build.rs file and use reflection to include them in the compiled Rust code. This might be an issue for users who do not want to use workspaces or split up their programs into separate crates. However, recent shifts within the Rust community suggest that for larger projects, it is idiomatic to split programs up into separate crates to make use of faster compile times and separation of concerns. Therefore, this requirement may not be an issue for users as the Rust community moves closer towards a modularised program architecture.

5.4.2 The `Reflect` Trait

The development of the `Reflect` trait has largely allowed this project to be possible. It allows functions and values to jump the staging into the generated Rust code. However, its introduction has come with disadvantages as well as advantages to the usability of the library. One of the biggest issues, which to my knowledge, cannot be fixed while using the `Reflect` trait. The 6.1 section discusses how this may be fixed.

Take, for example, the following function:

```
#[reflect]
fn zip<A, B>(a: A) -> impl FnOnce(B) -> (A, B) {
    move |b| (a, b)
}
```

This is a function which takes a value and will return a function that converts the second value into a tuple of the two values. This is allowed without the `#[reflect]` macro, but not with it.

The compiler will throw the following error:

```
error[E0207]: the type parameter 'B' is not constrained by the impl trait,
                self type, or predicates
   --> parslers-test/build.rs:282:15
    |
282 | fn zip<A, B>(a: A) -> impl FnOnce(B) -> (A, B) {
    |               ^ unconstrained type parameter
```

In order to figure out why the compiler thinks this, the code generated by `#[reflect]` must be inspected:

```
#[allow(non_camel_case_types)]
#[allow(incorrect_ident_case)]
#[derive(Clone, Copy, Debug, Hash)]
pub struct zip;

```

```rust
impl<A, B> FnOnce<(A,)> for zip {
    type Output = impl FnOnce(B) -> (A, B);
    extern "rust-call" fn call_once(self, (a,): (A,)) -> Self::Output {
        move |b| (a, b)
    }
}
```

The issue is in the line `impl<A, B> FnOnce<(A,)> for zip // ...`, where we declare that the trait is generic over `A` and `B`, but do not constrain what `B` is bound by. This is a limitation of the `FnOnce` trait, which has not been kept up to date with the addition of generic associated types into the language. A workaround could be implemented by adding `B` as a generic to the `zip` struct, and rewriting the generated code to:

```rust
#[allow(non_camel_case_types)]
#[allow(incorrect_ident_case)]
#[derive(Clone, Copy, Debug, Hash, Default)]
pub struct zip<B>(std::marker::PhantomData<B>);

impl<A, B> FnOnce<(A,)> for zip<B> {
    type Output = impl FnOnce(B) -> (A, B);
    extern "rust-call" fn call_once(self, (a,): (A,)) -> Self::Output {
        move |b| (a, b)
    }
}
```

Which would work in theory. However, in practice, this would ruin the usage of `zip` as being identical to a function. A user would instead have to write something akin to `zip::<_>::default()` in order to use `zip` in their parser, which is not ergonomic. The 6.1 section will describe how this can be overcome with later additions of Parslers.

Finally, Reflect poses another issue. All structs used within Parslers parsers must implement the `Reflect` trait. This is an issue only because Rust does not allow foreign traits to be implemented on foreign types (those are, traits/types not declared in the crate). This is to disallow the same trait from being implemented on the same type in two separate crates used as dependencies in a main binary. There are workarounds to this, but this is less an issue with Parslers and more of an issue with the general Rust ecosystem. It is still worth noting that this can be considered a usability issue.

5.4.3 Conclusion

Although the review of Parslers' usability flaws has been critical, these issues that have been pointed out do not make Parslers a difficult library to work with. The code that must be written is still very close to pure Rust code, and the separation between the build.rs file and the generated Rust code should not deter many from writing code using Parslers. These are simply flaws that may be improved upon in the future while still remaining small enough that they should not be a deal breaker to those wishing to use Parslers within their own projects.

6 CONCLUSIONS

During this project, I have shown that it is possible to use the Rust programming language to implement a staged selective parser combinator library as a domain-specific language. Furthermore, I have shown that the use of staging in this way can produce parsers that are faster than the state-of-the-art libraries currently provided in the Rust ecosystem in the benchmarking process.

Although imperfect, I believe Parslers helps paves the way towards using staging to transform domain-specific languages written in Rust into native Rust code, while performing domain-specific optimisations on user-generated structures all at compile time.

During development, Parslers started off as a Rust macro DSL that used a custom syntax to describe parsers. However, I wanted to allow users to describe their parsers strictly as Rust code, using Rust's type system to prevent malformed parsers from being written accidentally. So Parslers started using Rust's staged compilation model to have parsers written as typed Rust structures, and compile them into native Rust code to be included in the final binary. This, however, needed a way to include functions and values declared in the build.rs file to be transfered over to the compiled Rust code, leading the way for something new in Rust: runtime reflection. The `Reflect` trait and macros became an entirely separate library which Parslers was built on top of.

With further work, Parslers could become a part of the Rust ecosystem. Furthermore, the Reflect package could help power more staged domain-specific languages in the Rust ecosystem, assisting in compile-time optimisations of user-written code.

6.1 Future Work

During subsequent versions of Parslers, there are several things I would improve upon.

6.1.1 The `Reflect` Trait

Although the `Reflect` trait was the enabling feat that allowed Parslers to make the jump between the different stages of compilation, it is not perfect. This has been discussed in the evaluation section previously.

One of the key issues with Reflect is that it still lacks semantic information. This could, in theory, be fixed by running Rust doc, a tool that automatically generates documentation from Rust code. This documentation can be outputted in JSON format and subsequently analysed in order to retrieve the declarations of functions and values used in Parslers at compile-time. The reason why I elected not to use this tool for Parslers during the project is that Cargo (the Rust build tool) creates a file lock when running commands that affect the target directory (such as building, checking, or documenting). This is to prevent race conditions in the target directory. However, towards the end of this project, another project by the name of cargo px was launched, which aliases cargo for the purposes of executing cargo commands during compile-time. If cargo px was launched prior to the start of this project rather than towards the end, this may have been a better route to go down rather than implementing runtime reflection for the purposes of Parslers.

6.1.2 Performance Optimisations

Parslers currently stands as the fastest general-purpose parsing framework for the Rust programming language. However, it is still far away from approaching the performance of hand-written parsers. Through implementing further optimisation passes, I believe the speed of hand-written parsers can be approached in some cases.

One of the large potential optimisations that could be explored as a part of the next iteration would be consumption analysis. Consumption analysis works by factoring out length checks in the parser

where possible. Take the following parser for example:

```
match_char('a').then(match_char('b')).then(match_char('c')).then(match_char('d'))
```

The generated code for this parser looks like the following:

```
pub fn parser(input: &mut std::str::Chars) -> Result<char, &'static str> {
    {
        {
            let old_input = input.clone();
            input
                .next() // Check for next character here
                .ok_or_else(|| "Found EOF when character was expected")
                .and_then(|c| {
                    if (f0)(c) {
                        Ok(())
                    } else {
                        *input = old_input;
                        Err("expected a specific character")
                    }
                })
        }
        .and_then(|_| {
            let old_input = input.clone();
            input
                .next() // Check for next character here
                .ok_or_else(|| "Found EOF when character was expected")
                .and_then(|c| {
                    if (f1)(c) {
                        Ok(())
                    } else {
                        *input = old_input;
                        Err("expected a specific character")
                    }
                })
        })
    }
    .and_then(|_| {
        let old_input = input.clone();
        input
            .next() // Check for next character here
            .ok_or("Found EOF when character was expected")
            .and_then(|c| {
                if (f2)(c) {
                    Ok(c)
                } else {
                    *input = old_input;
                    Err("Expected a specific character")
                }
```

```
            })
    })
}
```

The exact semantics of the generated code is not of much importance for this example. However, notice that `input.next()` is called three times (corresponding to the three characters matched during the parser with `match_char`). Each occurrence of `input.next()` makes a check to see if the end of the input stream has been reached. However, it is worth noting that the parser will always fail if the input stream has less than three characters. Therefore, it stands to reason that these length checks may be factored out of the parser at compile time via an optimisation pass on the parser. This exact analysis is detailed further in (Willis, Wu & Schrijvers 2022).

Parsers for languages such as Rust, C, C++, Java, and many others will have a large number of checks for keywords whose lengths are known at compile-time to the parser. This optimisation could provide performance benefits to those parsers.

Furthermore, parsers such as `match_char('a').or(match_char('b'))`, where a character is read on one branch of an `Or` combinator and the other, may also benefit from having only one length check at the beginning of the `Or` sub-parser execution.

6.1.3 Error Messages

For many programming language compiler frontends, parsers must produce detailed error messages in the case a syntax error is encountered. In the future, I hope to add this detailed error messaging into Parslers automatically. Using staging, reasonable error messages could be inferred and embedded into the parser at compile-time. This would provide great utility to users of Parslers, as this could potentially remove the need to format error messages themselves, as is required in several other parsing libraries. The notable exception to this is Chumsky (Barretto 2021), which produces very readable error messages upon failure. However, Chumsky's performance characteristics are rather poor, which does not make a good use case for high-performance parsing.

6.1.4 Asynchronous Parsing

Another avenue that would be exciting to explore is the idea of asynchronous parsing. That is, to parse an input stream as it is being received from an asynchronous executor. This is possible thanks to Rust's strong `async`/`await` functionality being implemented using `Future`s, which allows functions to be paused and continued as asynchronous data is required/given.

An interesting use case is for low-latency applications such as market making, where the faster you are able to parse, analyse and make a decision on data, the better you can perform in the markets. Being able to write fast custom parsers that can process data as it is being received opens up the avenue for reducing latency in data analysis pipelines.

Another example of how this may be useful is parsing data from a disk using buffered asynchronous readers, whereby a double buffer is used to parse data and fetch data at the same time. This would remove the need to store the whole data in memory, but at no cost of complexity to the user.

For grammars with finite look-ahead, the asynchronous parsers would require a ring buffer of size equal to the look-ahead in order to backtrack successfully upon failure. Using consumption analysis, the parser could be analysed to figure out the buffer size required at compile time, without any input from the user.

6.2 Example of the Branflakes Parser

This section will include an annotated example of a Branflakes parser built with Parslers. Branflakes is chosen because the grammar is small and simple, but complex enough that it can demonstrate how a user would interact with Parslers in their own projects.

Firstly, in a separate crate, define the structures used in the results of the parsers. Use the `Reflected` macro to automatically derive the `Reflect` trait on these structures.

```rust
use parslers_lib::reflect::Reflect;
use parslers_macro::*;

#[derive(Clone, Debug, Reflected)]
pub struct BranflakesProgram(pub Vec<Branflakes>);

#[derive(Clone, Debug, Reflected)]
pub enum Branflakes {
    Add,
    Sub,
    Left,
    Right,
    Read,
    Print,
    Loop(BranflakesProgram),
}
```

Then, construct the parser, and use the `Builder` struct provided to compile the parsers together, apply optimisations, and compile them to a branflakes.rs file in the target build directory.

```rust
use parslers_branflakes::Branflakes;
use parslers_lib::{
    builder::Builder,
    parsler::{auxiliary::*, *},
    reflect::*,
};
use parslers_macro::reflect;

#[reflect]
fn branflakes_val(
    p: Vec<parslers_branflakes::Branflakes>,
) -> parslers_branflakes::BranflakesProgram {
    parslers_branflakes::BranflakesProgram(p)
}
#[reflect]
fn branflakes_loop(p: parslers_branflakes::BranflakesProgram)
    -> parslers_branflakes::Branflakes {
    parslers_branflakes::Branflakes::Loop(p)
}

```

```rust
fn branflakes_program()
    -> impl Parsler<Output = parslers_branflakes::BranflakesProgram> + Clone {
    let left = match_char('<').then(pure(Branflakes::Left));
    let right = match_char('>').then(pure(Branflakes::Right));
    let add = match_char('+').then(pure(Branflakes::Add));
    let sub = match_char('-').then(pure(Branflakes::Sub));
    let print = match_char('.').then(pure(Branflakes::Print));
    let read = match_char(',').then(pure(Branflakes::Read));
    let loop_ = || {
        match_char('[')
            .then(branflakes_program())
            .map(branflakes_loop)
            .before(match_char(']'))
    };
    name("branflakes", move || {
        many(
            left.or(right)
                .or(add)
                .or(sub)
                .or(print)
                .or(read)
                .or(loop_()),
        )
        .map(branflakes_val)
    })
}

fn main() {
    let out_dir = std::env::var("OUT_DIR").unwrap();
    Builder::new("branflakes", branflakes_program())
        .add_parser("branflakes_validate", branflakes_program().then(pure(())))
        .reduce()
        .usage_analysis()
        .build(&format!("{out_dir}/branflakes.rs"));
}
```

Finally, use the `include!()` macro to add the generated file to the binary, and call the named parser directly.

```rust
mod branflakes {
    #![allow(warnings, unused)]
    include!(concat!(env!("OUT_DIR"), "/branflakes.rs"));
}

fn main() {
    let data = std::fs::read_to_string("LostKng.b").unwrap();

    let chars = &mut data.chars();
```

```
    let parsed = branflakes::branflakes(chars);
    println!("Output: {:?}", parsed.is_ok());
    println!("Remainder: {:?}", chars.as_str());

    let chars = &mut data.chars();
    let validate = branflakes::branflakes_validate(chars);
    println!("Validate: {:?}", validate.is_ok());
    println!("Remainder: {:?}", chars.as_str());
}
```

And with about 90 total lines of code, a ready-to-use Branflakes parser and validator are ready to be used.

REFERENCES

Allott, Nicholas; Lohndal, Terje & Rey, Georges. 2021, Synoptic Introduction, *A Companion to Chomsky*, pp. 1–17.

Anderson, Carter. 2020, *Bevy: A refreshingly simple data-driven game engine built in Rust.* Available: https://bevyengine.org/.

Barretto, Joshua. 2021, *Chumsky: A parser library for humans with powerful error recovery.* Available: https://crates.io/crates/chumsky.

Griesemer, Russ, Robert. Cox & Fitzpatrick, Brad. 2015, *cmd/compile/internal/gc: recursive-descent parser.* Available: https://go-review.googlesource.com/c/go/+/16665.

Hutton, Graham & Meijer, Erik. 1996, *Monadic parser combinators.*

Jäger, Gerhard & Rogers, James. 2012, Formal language theory: refining the Chomsky hierarchy, *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 367, no. 1598, pp. 1956–1970.

Muchnick, Steven et al.. 1997, *Advanced compiler design implementation*, Morgan kaufmann.

Pezoa, Felipe; Reutter, Juan L; Suarez, Fernando; Ugarte, Martín & Vrgoč, Domagoj. 2016, Foundations of JSON schema, In: *Proceedings of the 25th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, pp. 263–273.

Willis, Jamie; Wu, Nicolas & Pickering, Matthew. 2020, Staged selective parser combinators, *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, pp. 120:1–120:30. Available: https://doi.org/10.1145/3409002.

Willis, Jamie; Wu, Nicolas & Schrijvers, Tom. 2022, Oregano: staging regular expressions with Moore Cayley fusion, In: Nadia Polikarpova, ed., *Haskell '22: 15th ACM SIGPLAN International Haskell Symposium, Ljubljana, Slovenia, September 15 - 16, 2022*, ACM, pp. 66–80. Available: https://doi.org/10.1145/3546189.3549916.