Imperial College
London

MEng Individual Project

Imperial College London

Department of Computing

Parser Combinators in Languages without User-Defined Operators

Supervisor:
Jamie Willis

Author:
Eugene Lin

Second Marker:
Dr. Peter McBrien

June 20, 2022

Abstract

Parser combinators often embody the principles of software engineering best practices, from abstraction to simplicity. They provide an effective solution to the problem of parsing, offering numerous benefits such as the ability for the user to define functionality in the host language, to creating parsers that closely resemble the original grammar. However, there can be a high barrier to entry: not only can the idea of functional programming seem daunting to some, but pitfalls such as left-recursion can often serve to complicate an otherwise intuitive experience. Many existing solutions for left-recursion remain highly theoretical or incomplete, thus preventing real usage.

This project remediates some of these hurdles by creating a lightweight parser combinator library in TypeScript as well as a preprocessing pipeline to support additions, such as custom operators, to TypeScript. Furthermore, the preprocessor can also perform static analysis over the structure of the program, providing an opportunity to optimise parsers. A key contribution of this project lies in this analysis: pitfalls such as left-recursion can be detected and automatically corrected, while maintaining the desired semantics, via a series of transformation and reduction steps.

Acknowledgements

First and foremost, I need to extend an immense deal of gratitude to Jamie. Not only would this project have been impossible without his support and guidance, I would not have even been introduced to the idea of parser combinators in the first place. I would also like to extend my gratitude to those who took the time to proofread this entire report on *extremely* short notice.

Next, I would like to thank the friends I have made at university; the past four years would not have been the same without you all. And for my friends from before university; I cannot thank you all enough for your patience with me. Finally: I want to thank my family, especially my parents, for all the support they have given me.

Contents

Chapter 1

Introduction

With the exception of hand-crafted parsers, most parsers can be placed into one of two categories; parser combinators, such as *Parsley* [1] and *megaparsec* [2], and parser generators, including *Yacc* [3] and *ANTLR* [4]. One major difference lies in how the user specifies the language; the former (parser **generators**) requires a preprocessing step over a DSL (domain-specific language) representing the grammar in a form resembling EBNF (extended Backus-Naur form). This generates source code in the host language, which can then be used to parse input.

On the other hand, parser **combinators** are implemented directly in the host language and do not require a preprocessing stage. Simple functions, which are individual parsers, take in an input and either give a successful (which would contain the parsed tree) or a failed response. These can be combined in a number of ways to produce more complex parsers of arbitrary types.

As expected, both approaches come with their respective benefits and drawbacks. A major disadvantage of parser generators is the generated parse tree; this is typically in a form specified by the creators of the library and may be tedious to traverse in order to convert into a desired structure, whereas a parser combinator can directly generate the desired tree in the host language. An advantage of parser generators is the ability to specify precedence rules, whereas this may have to be done by nesting the grammar with some parser combinator libraries. However, libraries often provide this functionality on top of the base combinators.

Another major advantage for parser combinators is the ability to abstract common patterns in the host language. For example, the aforementioned precedence rules could trivially be implemented by the user when using parser combinators. This advantage can be seen in Figure 1.1. In this example, a common pattern of a comma separated list is abstracted out by a user-defined function `commaSep`. Not only is the representation simpler, it also provides the desired structure of a list of elements (rather than two parse trees that have the same shape) and has less repetition.

```
arr : '[' es ']' ;
es  : e | e ',' es ;
obj : '{' ps '}' ;
ps  : p | p ',' ps ;
```

$\rightsquigarrow$

```
def commaSep[A](p: P[A]): P[List[A]]
  = p <::> many(',' *> p)
val arr = '[' *> commaSep(e) <* ']'
val obj = '{' *> commaSep(p) <* '}'
```

Figure 1.1: Left: example grammar (*ANTLR4*) for defining array and object literals, right: representation of the same grammar with parser combinators (*Parsley*)

With parser combinators, or more generally recursive descent parsers, a key limitation is the inability to handle left-recursive grammars. A fairly mechanical technique to overcome this is to simply eliminate the left-recursive production (in general, a non-terminal production $A \rightarrow A\alpha \mid \beta$ can be written as $A \rightarrow \beta R$ and $R \rightarrow \alpha R \mid \epsilon$ [5]). This can be seen in Figure 1.2. While this can be quite easily performed with a simple grammar, it already begins to look less intuitive than the original production, and one could imagine the complexity that would arise from more complex grammars.

```
expr ::= expr '+' nat | nat          ⇝          expr  ::= nat exprR
                                                exprR ::= '+' nat exprR | ϵ
```

Figure 1.2: Left: original grammar for adding numbers, right: rewritten grammar (without left recursion)

The aim of this project is to lower the barrier to entry for using parser combinators. The first of which is the host language - the vast majority of existing libraries are built in functional languages for obvious reasons, including the functional nature of this style of parsing. However, this adds an additional burden on the user in that they may have to learn an entirely new paradigm. As such, this project introduces a parser combinator library built entirely in TypeScript: *Teaspoon*[1]. Another barrier to entry is the small nuances that exist [6], such as the previously stated limitations with left-recursive productions.

This project contributes a preprocessor over TypeScript that not only allows for the use of combinators as operators, rather than functions, but also perform analysis over the combinators in order to aid the user and remediate detected anti-patterns. The use of a preprocessor also allows for the definition of operators beyond what the language natively provides, thus permitting syntax which can be more intuitively read (compared to numerous nested function calls) such as:

```
let u = chr('_');                      // underscore
let l = re(/[A-Za-z]/);                // letter
let d = re(/[0-9]/);                   // digit
let c = (xs: string[]) => xs.join(""); // concatenate

let id_op = c <$> ((u <|> l) <:> many(u <|> l <|> d));
let id_fn =
  fmap(c, liftCons(choice(u, l), many(choice(choice(u, l), d))));
```

Outline This report begins by exploring parser combinators (and parsers in general) in detail, alongside the common pitfalls that may arise from their use in chapter 2. With this intuition on how parsing can be performed by building up smaller parsers, chapter 3 highlights the design decisions and underlying implementation of the TypeScript-based parser combinator library. While the library is functional on its own, a preprocessor is required to support custom operators. This is explored in chapter 4, which details the architecture of the preprocessor, the design decisions made, as well as the extensions made to TypeScript. The optimisations found in the preprocessor are analysed in greater detail in chapter 5, with chapter 6 verifying the soundness of the transformations performed. The efficacy of these optimisations are then evaluated in chapter 7 alongside the performance of the library in different scenarios. Finally, chapter 8 details the contributions made, as well as how this work can be explored further.

[1]A fitting name for files with an extension of `.tsp`

Chapter 2

Background

2.1 Parsers

A parser forms the first stage of the compilation pipeline. Intuitively, parsing (also referred to as syntactic analysis) is the process of obtaining meaning from an input, typically a string, and deriving meaning in some form of structure, based on some grammar. For example, consider a simple expression: within these operators there are a series of precedence rules that should be adhered to, rules that are intuitively known. However, this needs to be formally specified to a parser in order to generate the parse tree in Figure 2.1; the original input string has no real meaning without a structure.

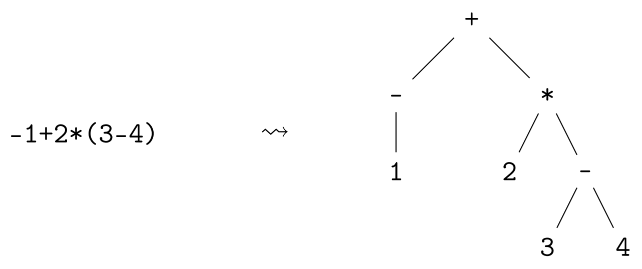


Figure 2.1: An example parse tree for a simple expression

It is important to first differentiate between the types of parsers. In general, parsers can be split into two categories; top-down (including LL(k) parsers, recursive descent) and bottom-up (including LR(k) parsers, shift-reduce). The former builds the AST from the root nodes to the leaf nodes, whereas the latter builds from the leaf nodes to the root nodes (hence bottom-up).

Top-down parsing begins with a non-terminal, and once a token is identified, it is used to 'fill' in the tree, creating a root node and child nodes based on the expected structure of the root node from the first identified token. At this point, any non-terminals that are in the tree are produced in the same way, building from the root node downwards. Due to the recursive nature of this style of parsing, it is evident that recursively applying a rule can be problematic if care is not taken, as discussed in subsection 2.3.1.

On the other hand, bottom-up parsing tries to use all the rules (whereas top-down tries to match a non-terminal by trying each of the rules), and replaces it with a non-terminal, by using the production in reverse. Bottom-up parsing succeeds when the whole input is replaced by the start symbol.

Both LL(k) and LR(k) parsers perform left-to-right scanning and have k tokens of look-ahead [5], but differs in the former having left-most derivation whereas the latter has right-most derivation. Note that LL(n) $\subseteq$ LR(n), hence LR parsers are more powerful.

2.2 Types

Throughout this section, the type `Parser` will be used. In TypeScript, the host language for the project, this is defined as the following (where `Result` captures whether input is consumed, as well as whether the parser succeeded).

```
type Parser<A> = (s: State) => Result<A>;
```

However, for brevity and readability, Haskell's type syntax will be used instead. Note that TypeScript does not have a specific type for characters, as such any use of `Char` in Haskell would be a use of string in TypeScript (however, a fixed length of 1 is assumed). Note that additional work will also be required to concatenate a list of singleton strings into a single string, which will likely lead to the produced code being more verbose.

2.2.1 Primitives

As combinator parsing involves joining together basic primitives, it is natural to begin with the most basic parsers, as described by Hutton & Meijer (1999) [7]. The first primitive is that of a parser that will always succeed if it is able to consume input and fail otherwise. This has the signature `item :: Parser Char`.

On the other hand, a parser that always fails, regardless of the input, is also required. Note that this comes from the `Alternative` typeclass. This parser can be referred to as `empty :: Parser a` - the type of which is irrelevant as no parse tree will be produced.

Finally, a primitive that lifts a pure value into the context of a parser is also required. Note that this is in both the `Applicative` and `Monad` typeclasses, referred to as `pure` and `return` respectively. As `Applicative` is a superclass of `Monad`, by default `return` refers to `pure`. This has the following signature:

```
return :: a -> Parser a
```

Alongside `(>>=)` in subsection 2.2.5, this can be used to form the parser `satisfy`, which parses a character based on a predicate.

While it is possible to define `satisfy` in terms of `item`, via the use of `bind`, it is also possible to go in the 'opposite' direction by defining `item` as `satisfy (const true)`. Using `satisfy` as the primitive is often preferable when compared to using `item` as `bind` significantly decreases the efficacy of inspection. This primitive comes from a natural interpretation of reading a character and has the type:

```
satisfy :: (Char -> Bool) -> Parser Char
```

2.2.2 Functor

A functor is a typeclass that allows for a function to be applied to the contained pure value, without altering the overall structure [8]. This is done with the following function:

```
fmap :: (a -> b) -> Parser a -> Parser b
```

It can also be written as `(<$>)`, for infix notation.

By giving parsers this property, the user is now able to modify the resulting parse tree without changing any other properties of the parse result. This includes the success of the result: if it was a failure, then nothing is done. A concrete use of this is to create a parser for natural numbers: after consuming a sequence of characters (hence forming a string), it can be converted into an integer, giving a `Parser Int`.

Using the above, a derived combinator `(<$)` arises. The first argument is used in place of the resulting parse tree from the second argument (a `Parser`), if the parse was successful. This has the signature:

```
(<$) :: a -> Parser b -> Parser a
x <$ p = const x <$> p
```

By treating `Parsers` as `Functors`, it is possible to assign meaning to a result, by transforming it beyond that of a literal string.

2.2.3 Applicative

First note that an alternative typeclass to this is `Monoidal`, where both can capture the idea of sequencing parsers. Earlier references, such as Wadler (1985) [9] and Fokker (1995) [10], provide sequencing in the form of pairs:

```
(<~>) :: Parser a -> Parser b -> Parser (a, b)
```

While this may seem intuitive for simple combinations of parsers, it can become tedious and difficult to handle when more parsers are sequenced together. For example, consider sequencing three parsers, leading to nested pairs:

```
pa   :: Parser a
pb   :: Parser b
pc   :: Parser c
pabc :: Parser ((a, b), c)
  = pa <~> pb <~> pc
```

In general use cases, the use of pairs to represent sequential operation can become too restrictive. Naturally, this leads to a more general form.

Note that `Applicatives` allow for sequencing of functional computations but do not allow for the use of results from earlier computations (which are provided with `Monad` in subsection 2.2.5). They also do not generally enforce an execution order (however in the context of `Parsers`, this is enforced by `Monad`). Recall the `return` 'primitive' mentioned earlier in subsection 2.2.1, in the `Applicative` context, this is referred to as `pure`, however the semantics remain the same. The method `(<*>)` can be intuitively thought of as a more general function application [11] in the context of `Parsers`:

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

The aforementioned example of `(<~>)` can be written as follows (in terms of `<*>`) - note that the first parser is mapped into a parser of a function:

```
px <~> py = (,) <$> px <*> py
```

Important derived combinators that arise from this include the following [12]:

```
(<*)   :: Parser a -> Parser b -> Parser a
p <* q = const <$> p <*> q

(*>)   :: Parser a -> Parser b -> Parser b
p *> q = flip const <$> p <*> q

(<**>) :: Parser a -> Parser (a -> b) -> Parser b
p <**> q = (flip ($)) <$> p <*> q

(<:>)  :: Parser a -> Parser [a] -> Parser [a]
p <:> ps = (:) <$> p <*> ps
```

The first two combinators execute both parsers, and only results in a successful parse when both succeed. However, only one of the two parsed results are used, and the other is discarded. The next combinator (`<**>`, or ‘reverse ap’) is similar to the original `<*>`, but instead has the function as the second parser. Finally, the `<:>` combinator, also referred to as lifted cons, allows joining an element to a list of elements in the `Parser` context.

In addition to these derived combinators, it is also possible to define ‘lift’s, which combine n parsers using a function. Note that the derived combinators can also be defined in terms of lifts. For example, `<:>` and `<**>` can be defined as the following (in TypeScript), where the first parameter is an uncurried function that contains the desired functionality:

```
let lift_cons = lift2((p: A, ps: A[]) => [p, ...ps], p, ps);
let rev_ap    = lift2((p: A, q: (a: A) => B) => q(p), p, q);
```

With these combinators, it is now possible to parse combinations of primitives, given a completely fixed structure.

2.2.4 Alternative

Note that `Alternative` exists as a subclass of `Applicative`, additionally it is closely related to `MonadPlus`, where the zero elements are `empty` and `mzero` respectively, and the combining functions are `(<|>)` and `mplus`. For simplicity, only the `Alternative` definitions are included.

The `empty` parser has been previously mentioned in subsection 2.2.1, as a primitive parser. Note that combining any parser with the `empty` parser by using the combining function will result in the original parser. This has the signature:

```
(<|>) :: Parser a -> Parser a -> Parser a
```

This combinator captures the ability to choose between alternatives. For example, the parser `x <|> y` will first attempt to parse `x`, however if it fails, it will then attempt to parse `y`. Note that with the *Parsec* [13] family of parsers, the second parser is only attempted when the first parser does not consume input. However, this behaviour may not be desired thus a parser may want to roll back any consumption, or ‘backtrack’. Backtracking is done with `try` (in the case of *Parsec*), where the parser acts as if it did not consume input on a failure. If both fail, the overall parser fails, however if either succeed, the first to succeed is the result of the parser. It is crucial to note that while `<|>` is associative, it is not commutative.

Alternatives, including `empty`, provide a number of laws that can be beneficial for performing optimisations. Note that these laws are not always general, but hold for parsers. These include the left and right neutral laws $(1, 2)$, as well as the left absorption law (3) and left catch law (4):

$$\texttt{empty <|> p} \Rightarrow \texttt{p} \tag{1}$$

$$\texttt{p <|> empty} \Rightarrow \texttt{p} \tag{2}$$

$$\texttt{empty <*> p} \Rightarrow \texttt{empty} \tag{3}$$

$$\texttt{pure x <|> p} \Rightarrow \texttt{pure x} \tag{4}$$

The left absorption (zero) law is generally accepted, whereas the right absorption law is not. In parsers, the parser `p` in `p <*> empty` may consume input, thus changing the state. The left catch law essentially states that `pure` (from `Applicatives`) represents a successful parse (specifically in the context of parsers).

By using the `Alternative` typeclass, the parsers are able to capture choice. Additionally, this allows for further combinators such as `many` to be defined, where sequences of arbitrary length can be generated; without choice, a production would either be a fixed finite number, or an infinite length. For example:

```
A ::= 'a' A -- infinitely many 'a's
B ::= 'b' C -- exactly 2 'b's
C ::= 'b'   -- exactly 1 'b'
```

However, with choice, it is therefore possible to define the following, where as many 'a's are parsed as possible, until there are no more 'a's to consume (thus terminating the first branch of the parser ('a' A)):

```
A ::= 'a' A | ε
```

2.2.5 Monad

By using the Monad typeclass, the bind operator is introduced. This has the signature:

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
```

An interpretation of this combinator is similar to that of <*> in subsection 2.2.3, however the result of the first parser is applied to a function that creates the second parser, rather than being applied to the result of the second parser (which is a function). However, as the result of the first parser can directly influence the result of the second parser (since the function can use the result of the first parse), this creates a guarantee on the order of operations. The first parser must be executed before the function that produces the second parser can be.

While this permits for context-sensitive parsing, as the function producing the second parser can access the results of the previous parse, it severely impacts the efficacy of inspection and analysis. Due to this decrease in efficacy, bind will not be a primary focus for subsequent optimisations, but it is still important to note due to its power (it can implement both <*> and <$>).

2.3 Pitfalls

2.3.1 Left Recursion

One of the motivating anti-patterns is the difficulty of handling left-recursion [6]. Recall the basic expression grammar expr, first mentioned in chapter 1.

```
expr ::= expr '+' nat | nat
```

Using the combinators that have been introduced above, this can be written as the following (actually performing the calculations):

```
lazy expr = expr <**> ((x => y => x + y) <$ chr('+')) <*> nat <|> nat;
```

Listing 2.1: Running example of simple addition parser

Now consider the execution of this parser: when the parser attempts to parse an expr, the first thing it will do is attempt the first 'choice'. In this choice, it will then attempt to parse the first parser of that sequence - this is the left-most parser, and it is expr. Naturally, it will then attempt to parse another expr; however, at this point no input has been consumed, and the input state remains unchanged - the parser is back where it started; it has looped and made no progress.

The example above shows direct left-recursion, where the production is the left-most lexeme in its own rule. In general, left-recursion can be classified into one of three categories; **direct**, **indirect**, and **hidden** [14]. A simple example of an indirect left-recursion would exist in the following grammar, where A is not directly in the right-hand side of the rule for A, but exists a level down, in the rule for B. Note that A and B are non-terminals and α and β are terminals.

$$A \rightarrow B\alpha \mid \cdots$$
$$B \rightarrow A\beta \mid \cdots$$

Finally, hidden left-recursion occurs when the first non-empty lexeme is itself. For example, this can be seen in the production $A \rightarrow \epsilon A$, where ϵ is empty. The example given for hidden left-recursion is very simple, however it can also be indirect; for example $A \rightarrow BA$, and $B \rightarrow^* \epsilon$ (where $\rightarrow^*$ means B eventually reduces to ϵ) is also left-recursive. Note that even LR(k) parsers are unable to handle hidden left-recursion, as mentioned by Nederhof & Sarbo (1993) [15].

2.3.2 'Dangling Else'

A common pitfall that could be encountered is the parsing of keywords in many programming languages. For example, if the production rules for keywords in a language consisted of $\cdots$ | `'in'` | `'include'` | $\cdots$ there may be a number of issues. The first of which is the order in which the rules are tested in. As `Alternative` will attempt the rules from left-to-right (as mentioned in subsection 2.2.4), once it matches `'in'`, it will have parsed a keyword, with the remainder being `'clude'`, rather than parsing the longest string. As such, some care will need to be taken when constructing rules for possible ambiguity.

Another pitfall may arise without the use of `attempt`. For example, if the grammar were to be rewritten as $\cdots$ | `'include'` | `'in'` | $\cdots$, without using `attempt`, once it fails to match `'include'` on the input string `'in'`, it would've already consumed input, causing `'in'` to fail to match.

2.3.3 Overuse of Backtracking

As shown previously, the use of `attempt` can essentially provide infinite lookahead - an extremely powerful property. However, using `attempt` can often come with a performance penalty (exponential time and space in the worst case [13]), therefore care must be taken to prevent this by only using it when necessary. Furthermore, the quality of error messages can also be decreased.

In order to remedy this, it is possible to left-factor the grammar to accommodate shared prefixes as discussed by Hutton & Meijer (1999) [7] thus allowing for parsing to complete in linear time. However, this can often be a tedious process that complicates the parser and reduces parity between the parser and the original grammar. As such, `attempt` may be used in lieu of designing parsers with care.

2.4 Parsing Expression Grammars

Parsing Expression Grammars (PEGs), described by Ford (2004) [16], provides a formal recognition-based language that is closely related to the top-down parsing language (TDPL) specified by Birman (1970) [17] (later named by Aho & Ullman (1972) [18]). Context-free grammars (CFGs) are designed to express natural language and do so extremely well. However, this generality can become too 'powerful' for programming languages, thus leading to ambiguities. PEGs reduce this generality and provides a key differentiating factor: deterministic choices via the use of priorities. In EBNF (Extended

Backus-Naur Form), the choice operator is commutative, therefore `a | b` is equivalent to `b | a`. However, as mentioned in subsection 2.2.4, this property does not hold for parser combinators. Instead, the semantics of alternatives closely mirror that of PEGs, where `a / b` is not equivalent to `b / a` - the former first attempts `a` and only attempts `b` if `a` fails (ignoring input consumption) and vice versa for the latter. This allows for the 'dangling else' ambiguity, introduced in subsection 2.3.2, to be trivially resolved by having the 'longest' production first.

Parsing is inherently a problem regarding the decomposition of structure. As such, a recognition-based system (such as PEGs) is more suitable compared to generative systems (including CFGs). The former determines whether a given string is valid in the language, whereas the latter defines a language by recursively generating strings [16].

In addition to the introduction of the prioritised choice binary operation, operators such as $p+$, $q*$, and $r?$ (representing one-or-more ps, zero-or-more qs, and an optional r, respectively) are also included.

Chapter 3

Parsing Library

This chapter discusses the implementation of the underlying TypeScript library, *Teaspoon*, beginning with the original port from *Parsec* (Haskell). Following the port, the library features are discussed, followed by any modifications made for performance or usability.

3.1 Porting from Parsec

It is important to first acknowledge that a significant amount of the underlying implementation is built off of the work by Hutton & Meijer (1999) [7], as well as the later work on *Parsec* by Leijen & Meijer (2001) [13]. However, a number of considerations must be made as code is essentially being ported from an implementation in Haskell to TypeScript.

The internal state contains both the position as well as the remaining string, similar to the implementation in Haskell. Similarly, the consumer-based approach was also taken, where both the success of the parse and consumption is tracked - closely related to the four tags proposed by Partridge & Wright (1996) [19]. In TypeScript, consumption is simply a flag that is set in a `Result<A>`, which also contains a reply, as shown in Figure 3.1.

```haskell
data Consumed a = Consumed (Reply a)
                | Empty (Reply a)
data Reply a    = Ok a State Msg
                | Err Msg
```

$\rightsquigarrow$

```typescript
type Result<A> = {
    consumed: boolean;
    reply: Ok<A> | Err;
};
```

Figure 3.1: Left: four ‘tags’ from *Parsec* (Haskell), right: representation of the tags in *Teaspoon* (TypeScript)

However, a significant language feature present in Haskell but not TypeScript is the presence of lazy evaluation. An incomplete implementation of laziness, which does not support sharing (preventing repeated evaluations), can be implemented by a function which takes in a ‘thunk’ (a function which can be evaluated later to delay the computation). This ‘thunk’ is then only evaluated once the lazy parser is called, thus allowing for values to be used prior to an assignment - a critical feature for recursively defined grammars.

```typescript
function lazy<A>(p: () => Parser<A>): Parser<A> {
    return (s: State) => p()(s);
}

let p = lazy(() => q); // 'Using' q before it is assigned
let q = pure(true);    // Actual definition of q
```

Listing 3.1: Rudimentary implementation of laziness (delayed evaluation only)

While this port creates a useful starting point, the performance leaves much to be desired. Not only are some features implemented in a way that is not natively 'supported' (not as well optimised) in TypeScript, some features benefit from a less 'pure' implementation.

3.2 Features

Teaspoon provides a number of primitives, as described in subsection 2.2.1, with `satisfy` being treated as a primitive alongside `item`. Regarding the predicate passed into `satisfy`; an expectation is made that the condition is applicable only to characters as TypeScript does not distinguish between characters and strings. Note that `return` is named `pure` in order to avoid conflicting with TypeScript's `return` keyword. Additionally, `try` is named `attempt` for the same reason. Finally, the failure parser is defined as a parameterless function in order to support polymorphism - `empty<A>()`. Another addition is the presence of a parsing primitive based on regular expressions. Note that using regular expressions can be preferable in many cases - in the case of natural numbers, there is a threefold performance improvement using `nat_r` in listing 3.2 versus using `nat_c` to parse a 16-digit number[1].

```typescript
let nat_c = fmap(
    (s: string[]) => parseInt(s.join("")),
    many1(satisfy((c: string) => c >= '0' && c <= '9'))
);
let nat_r = fmap((s: string) => parseInt(s), re(/[0-9]+/));
```

Listing 3.2: Natural number parser based on using 'traditional' primitives (`nat_c`) versus parser using regular expressions (`nat_r`)

In addition to the primitives, the library provides a number of combinators, including derived combinators, described throughout section 2.2. An important difference to note is that the implementation utilises the same semantics as previously described, albeit with uncurried functions. The combinators and their respective types are shown in Table A.1.

Alongside the 'operator'-based combinators, some commonly used derived functions are also implemented. Within the library, the implemented functions include the homogenous chain combinators, described by Willis & Wu (2021) [6], `chainl1`, `chainr1`, and `postfix`. In addition to the chain combinators, implementations exist for `liftN` (from `lift2` up until `lift9`). However, recall that a benefit of parser combinators is the ability for the user to implement this functionality, should anything be lacking. Finally, additional higher-order functions are also included to permit operations such as `uncurry`, `curry`, and `flip`. These functions, alongside `const` and `id`, are usable by the user, as well as by the preprocessor.

As this library is based on *Parsec*, the same error message semantics are included; thus, meaningful error messages are provided in the parse result. In addition to the error messages, the `label` operator (`<?>`) is also included to improve messages.

3.3 Performance

First, note that the majority of the performance optimisations will revolve around the results of the JSON parser, detailed further in section 7.3, as this grammar tests numerous library features and also performs well as an indicator of real-world performance. However, somewhat pathological cases are also observed in order to highlight certain design decisions.

A major point of optimisation lies in functions that consume an arbitrary number of repetitions - namely `many`, `many1`, `postfix`, and similar (including `chains`). With repetition

[1] $9,007,199,254,740,991$ is the value of `Number.MAX_SAFE_INTEGER` ($2^{53} - 1$) [20]

being a common pattern in numerous grammars, as well as `postfix` being heavily utilised in the removal of left recursion (detailed in section 5.2), reducing the performance penalty can bring significant benefits. With the base implementation, this is done with expensive recursive calls, with no chance of optimisation from the TypeScript compiler. However, this can trivially be replaced with an accumulator, iteration, and a mutable state. This optimisation alone gave up to a fourfold performance improvement on grammars that heavily utilise these parsers, such as JSON as seen in Figure 3.2.

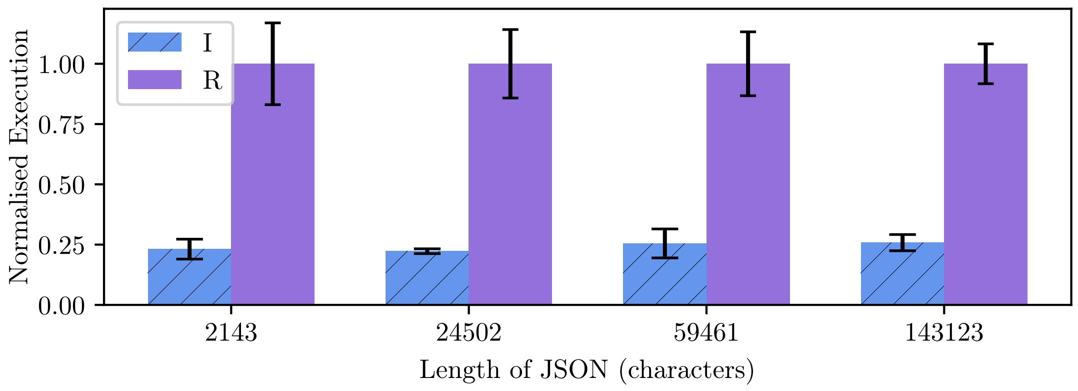


Figure 3.2: Normalised execution time of an iterative implementation (I) versus a recursive implementation (R)

Recall the previous implementation of laziness resulted in the parser being 'generated' for each use. This leads to an obvious inefficiency - it does not make sense for the parser to be regenerated as long as the generating function is pure (no side effects). While no guarantees can be made about the 'thunk', the general, non-pathological use case of this feature will have pure functions. In general, any calls to `lazy` will typically be generated by the preprocessor rather than by the user. As such, it is possible to store the result of the thunk after it is called for the first time, thus implementing sharing.

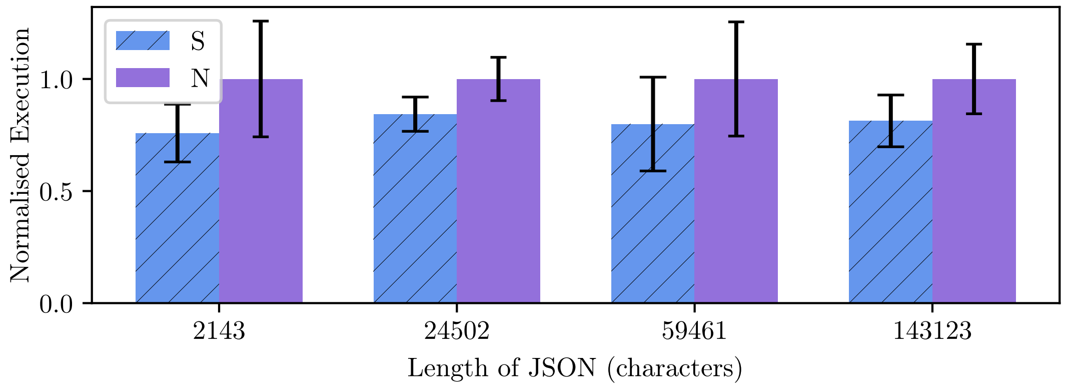


Figure 3.3: Normalised execution time of laziness with sharing (S) versus laziness with no sharing (N)

When the `lazy` function is called, an instance of the `_Lazy` class is created, which contains two properties; `_f` (the thunk) and `_p` (the actual 'generated' parser) which is initially `undefined`. Once an attempt is made to access `_p`, the stored value is checked on whether it is defined or not. If it is defined, the value is simply returned, otherwise the thunk is executed and the value stored. While the creation of classes can create some minor overhead, the optimisation still results in an overall performance uplift. This is visible in

Figure 3.3, where an improvement of ≈ 25% on execution times can be observed. It is important to note that this result is heavily dependent on the grammar and the amount of laziness required.

Another notable change made from the port is the use of uncurried functions over the use of curried functions. While there is a very minor performance improvement from using uncurried functions, the main advantage of uncurried functions over curried functions is the fact that TypeScript's type inference occasionally fails to obtain the type for a curried function, but not for the uncurried version. This issue can be seen in listing 3.3, where the type of `u` is successfully inferred to be `Parser<number>`, whereas the type of `c` fails to infer and becomes a `Parser<unknown>`.

```
let ds = many1(satisfy((c: string) => c >= '0' && c <= '9'));
let d2n = (ds: string[]) => parseInt(ds.join(''));

let c = pamf_c(ds)(d2n); // failure: 'Parser<unknown>'
let u = pamf_u(ds, d2n); // success: 'Parser<number>'
```

Listing 3.3: Example of failed type inference on a simple natural number parser

While this is a minor annoyance for a user who is manually writing the parsers (as the development environment will likely warn them), it is significantly more problematic when the function calls are being automatically generated. This issue is caused by TypeScript's inability to infer the function's overall type, when type information is required from the second 'call'. In the example above, `pamf_c<A, B>` requires the type of `d2n` to obtain the type `B`. If the example instead used `fmap_c(d2n)(ds)`, inference would succeed as both `A` and `B` of `fmap_c<A, B>` are provided by `d2n`.

Summary

This chapter explored the underlying design decisions in the parsing library, as well as how fundamental differences between Haskell and TypeScript prevent a one-to-one port of *Parsec* from being sufficient. These differences include language features such as laziness, mutability, as well as the limitation of type inference. Throughout the sections, some examples of how the high-level interface of the library may be utilised are shown - however they differ drastically from the 'neat' operators typically associated with parser combinators. In order to support these operators without directly modifying the TypeScript compiler, a preprocessor (described in chapter 4) will be required.

Chapter 4

Preprocessor

The TypeScript-based library introduced in chapter 3 contains an implementation of primitive parsers as well as commonly used combinators. However, in order to retain the intuitive syntax that users are likely to be accustomed to from other combinator libraries (in languages with user-defined operators), a preprocessing step is required. While the introduction of the preprocessor begins to blur the lines between parser combinators and parser generators, it provides an excellent opportunity to perform deeper inspection on parsers.

This chapter begins by introducing the overall flow of data through the preprocessing pipeline. Throughout the sections, various design decisions and underlying implementation details are explored. This includes the parser for the augmented grammar, the representation used throughout the optimiser, as well as the aforementioned additions to TypeScript.

4.1 Pipeline Architecture

This section aims to provide a high-level overview of the main processing pipeline, from ingesting the source file, to optimising the program, and back to a source file. As shown in Figure 4.1, it is important to note that the optimiser itself consists of multiple stages, which can be partitioned into one of three categories; before the conversion to the IR, while the AST is in IR form, and after the conversion from the IR. The bulk of the optimisation is performed in the IR stage, whereas the steps before and after are primarily to support additional language features.

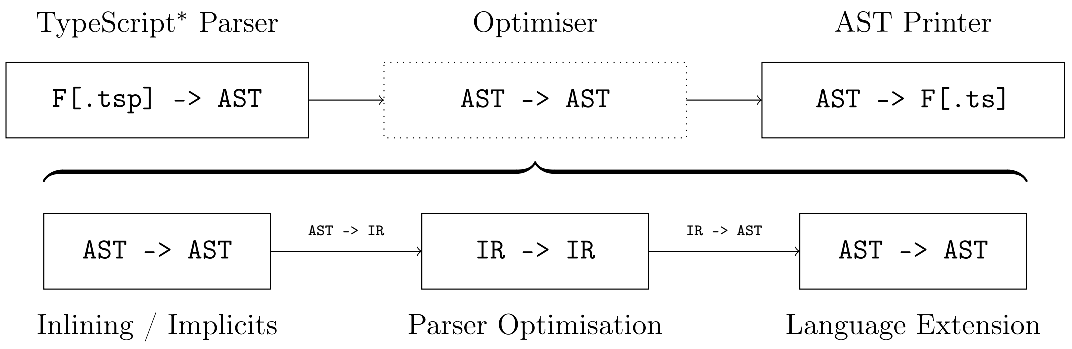


Figure 4.1: Simplified view of the main stages within the preprocessor pipeline, `F[.x]` denotes a file with extension `.x`

The primary entry point for the preprocessor uses *scopt* [21] to parse command-line options (see Table A.2), which allows for certain optimisation stages to be enabled / disabled. Once a file is read in, with the path specified by an argument, the raw input is then parsed by a TypeScript* parser. Note that TypeScript* refers to TypeScript with the additional language features augmented onto the existing grammar (listed in section 4.4).

Once the input file has been successfully parsed, the AST is fed through the optimisation pipeline, detailed further in chapter 5. Each 'stage', excluding language extensions and IR conversions, is either enabled or disabled, depending on the corresponding configuration option. Every pipeline stage extends the `PipelineStage` trait, which provides default AST traversal properties, allowing for a stage to only implement the desired functionality on a subset of the AST, but maintaining the guarantee that everything will be traversed. The latter is important to ensure that certain assumptions can be made about the AST being fed into subsequent stages. For example, it is generally safe to assume that any `Expressions` after the first conversion stage will be `IR` nodes (detailed in section 4.3). On overriding a `PipelineStage`, care should be taken to maintain the behaviour from the superclass (unless explicitly not required) in order to main the traversal properties.

Each of the pipeline stages can access a shared mutable state. However, pipeline execution typically implies an order of execution: stages cannot simply jump forward in the pipeline and back despite this functionality possibly being beneficial. For example, the preprocessor may need to report an error to the user and include TypeScript code. However, without the ability to use later pipeline stages (namely a conversion from the IR to the AST), the functionality for generating code will have to be duplicated. In order to prevent this, the mutable state can be 'locked', thus preventing any changes.

The final stage of the pipeline converts the AST back into a raw string by printing out the AST recursively. Note that this maintains no formatting, such as indent levels - however, the syntax is now fully valid TypeScript. Optionally, this raw file can then be passed through a formatter, such as *Prettier* [22].

4.2 TypeScript* Parser

The first stage of the processing pipeline involves parsing the raw input file, which contains a valid (both syntactic and semantic) TypeScript* program. Note that the remainder of the pipeline assumes semantic correctness, especially with types, as full type verification lies outside the scope of the preprocessor.

The parser within the preprocessor targets ECMAScript 2015 [20], augmented with TypeScript-specific additions from early 2016 [23]. Certain existing language features have been excluded, such as (but not limited to) decorators and ambient declarations. *Parsley* [1] for Scala is used for parsing the raw input into an abstract syntax tree (AST). The use of a parser combinator library not only seemed natural due to the nature of this library and preprocessor, but also provided numerous benefits, such as the single-pass nature of the lexing, parsing, and construction stages. One disadvantage of using parser combinators is the lack of error recovery - however, as the preprocessor typically expects correct, well-formed code, this is not a major concern.

Note that the AST contains additional language features not present in TypeScript. These additions are transformed in subsequent pipeline stages into other AST nodes, which represent valid TypeScript syntax. In order to recover a valid TypeScript program at the end of the pipeline, each AST node requires a `print` function, which recursively converts the tree back into program code. At the end of the pipeline, this is called on a `Module` representing the entire program, which is then optionally passed into a code formatter.

The entirety of the preprocessing pipeline, bar the formatter which exists externally, is implemented in Scala, in order to leverage its pattern matching abilities, as well as the portability of the JVM (Java Virtual Machine). The former allows for significantly simpler analysis over the AST, which is hugely beneficial for optimisations on a tree.

4.3 Intermediate Representation

In lieu of duplicating significant fragments of code to create an entirely separate intermediate representation (IR) to represent the AST, the IR instead extends the existing `Expressions`, by generating a mapping between the IR and the original or generated `Expression`. The inheritance structure of the IR can be seen in Figure 4.2 - note that `Id` also extends `BindingPattern` and `PropertyDefinition` from the original TypeScript AST. By augmenting the existing AST, a separate representation for other parts of the AST, such as statements or declarations, is not required.

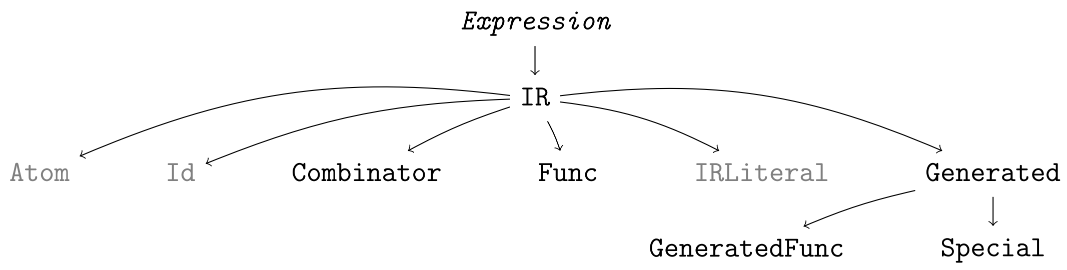


Figure 4.2: Inheritance hierarchy within the IR, nodes in `grey` are concrete whereas nodes in `black` are traits

`Atoms` contain a unique numeric identifier which provides a mapping from an `Atom` to an `Expression`, which is stored in a mutable state (implicitly used by all pipeline stages). These contain any `Expressions` that cannot be changed into any other representation. Generally, they contain nodes that require no further inspection, as they are unlikely to contain any combinators - especially not ones that can be meaningfully inspected.

On the other hand, `Combinators` are a special case of regular binary operators which contain combinators (any other operation will be converted into an `Atom`). For each different combinator, there is a different concrete implementation (`case class`), allowing for easier inspection in subsequent pipeline stages (via matching on a specific combinator, rather than checking which operation is being performed manually). Each of these combinators has two arguments, both of which are also IRs. Similarly, `Funcs` have concrete implementations for reserved functions that are provided with the TypeScript library; these have specific arguments which are assumed to be valid when processing. This adds the ability to perform some inspection on functions which have known behaviours.

`Id` and `IRLiteral` both have similar functionality in that they wrap existing nodes (namely, `Identifier` and `Literal`) to work within the IR. The former is handled separately as it can often be useful to be able to directly access variable names in later stages without the need to reference the shared state (if they were instead represented as `Atoms`). On the other hand, `IRLiterals` are used for a similar reason, however the main purpose is to allow for inspection into string or character literals, which is useful for subsequent stages.

Finally, the `Generated` trait refers to any nodes without a mapping from an `Expression`, meaning that they are only generated by the processing pipeline. This is further divided into one of two traits. Nodes which extend the `GeneratedFunc` trait represent functions that are applied to IR nodes. These functions have a 'fallback' implementation within the library; however, the preprocessor will attempt to directly alter the AST whenever possible. In contrast, nodes which extend the `Special` trait are typically used to represent specific values or are used to directly pass information to other stages. For example, it may be required to explicitly pass an `Expression` forward without any modifications or highlight characters which exist in the first set of a parser.

The example of addition from listing 2.1, after being parsed into the AST shown in listing A.1, is translated into the IR as shown in listing 4.1. Notice how only the information that is directly related to parsers is maintained, whereas the function is simply converted into `Atom(0)`.

```
(Id("expr") <**> (Atom(0) <$ Chr(IRLiteral("'+'"))) <*> Id("nat")) <|>
Id("nat")
```

Listing 4.1: IR of right-hand side of assignment

4.3.1 Function Rewriting

An example of the function rewriting is as follows, where `f0` is a function without AST rewriting (uses library functions) and `f1` is a function that is directly rewritten. While this may seem like a pathological example, these transformations are used extensively when rewriting the parsers to support left-recursion.

```
let f0 = flip(curry(unpairFirstArg(
    ([x, y]: [number, number], z: number) => x + y - z
)));
let f1 = ([y, z]: [number, number]) => (x: number) => x + y - z;
```

Listing 4.2: Example of function rewriting performed by the preprocessor

Not only does the direct rewrite improve code clarity by omitting multiple function calls, there is also a small but measurable performance uplift stemming from the reduced number of invocations. This can be seen in Figure 4.3, which performs 10,000 iterations of the function in listing 4.2. Within each iteration, the function is transformed and executed n times. This result demonstrates that with repeated invocations, the performance gap between the two forms becomes narrower. However, the directly rewritten form consistently performs better, especially with few invocations when it can complete execution in approximately half the time.

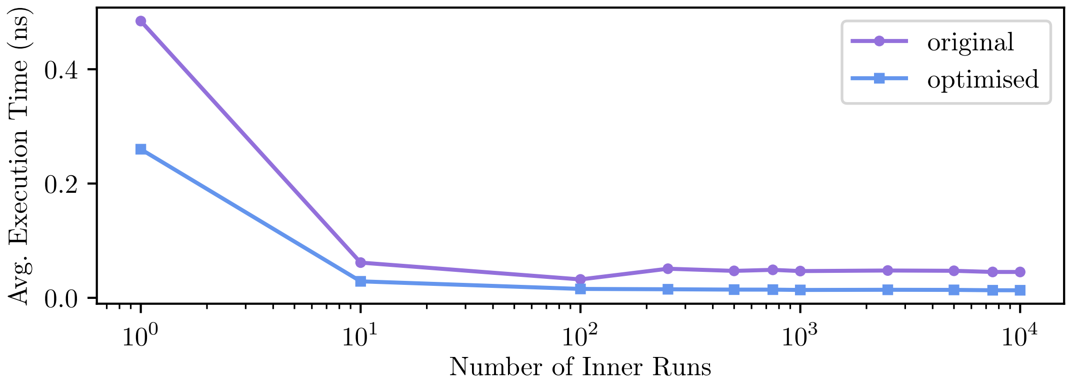


Figure 4.3: Change in average execution time based on number of repetitions

4.4 Language Extensions

As part of the preprocessor, a number of languages features are implemented that augment the existing TypeScript grammar, most of which are implemented to provide 'syntactic sugar' for writing parser combinators. This section introduces these additions alongside some context surrounding usage. Note that for all the subsequent code snippets in this section, the snippet on the left refers to the augmented grammar (`.tsp`) and the snippet on the right refers to the transpiled TypeScript (`.ts`).

4.4.1 Declarations

Two new lexical declarations have been added, namely `val` and `lazy`. The former is simply an alias for `const`, with no additional functionality. On the other hand, `lazy` allows for rudimentary laziness, thus allowing for parsers to be recursively defined - required for supporting recursive syntax. This can be seen in the example below, where `p` references itself. Without the additional wrapping for laziness, the code would be semantically invalid as a variable would be used prior to its assignment.

```
lazy p = p;
val q = x;
```

$\rightsquigarrow$

```
const p = lazy(() => p);
const q = x;
```

4.4.2 User-Defined Operators

In order to provide an intuitively legible syntax for writing parsers, user-defined operators have been implemented. Note that the snippet below does not provide usage for all implemented operators, but rather demonstrates how they are supported. In the example, `p` is an example of how a simple combinator is written into a function call. The left-associative property of combinators is also respected, as shown in `q`, with support for parentheses shown in `s`. Similarly, the lower precedence of the choice combinator (`<|>`) is shown in `r`. Finally, arbitrary user-defined functions can be supported, as shown in `t` - this addition allows for combinators to be defined beyond what is provided by the library, without requiring changes to the preprocessor.

```
let p = a <~> b;
let q = a *> b *> c;
let r =
  a <*> b <|> c <*> d;
let s =
  a <**> (pf <* b) <*> c;
let t = a <xyz> b;
```

$\rightsquigarrow$

```
let p = mult(q, r);
let q = apR(apR(a, b), c);
let r =
  choice(ap(a, b), ap(c, d));
let s =
  ap(pa(a, apL(pf, b)), c);
let t = xyz(a, b);
```

4.4.3 Implicit Conversions

In order to reduce the amount of conversion the user has to manually perform, implicit conversions can be enabled in the pipeline. With this enabled, a use of a string, character, or regular expression literal in place of where a parser is expected will cause the preprocessor to automatically apply a conversion (wrapping the function with `chr`, `str`, or `re`, depending on the type of the literal). This addition provides some ability to have 'overloaded strings', as described by Willis & Wu (2021) [6].

```
let p = f <*> 'a' <*> /b/i;
let q = 'c' $> 'd';
let r = attempt('ef');
```

$\rightsquigarrow$

```
let p =
  p(ap(f, chr('a')), re(/b/i));
let q =
  constFmapR(chr('c'), 'd');
let r = attempt(str('ef'));
```

4.4.4 ‘Macros’

One limitation of the preprocessor, when performing analysis, is observing the behaviour of more ‘complex’ functions or declarations. While it may be feasible to inline some variable declarations, if they are known to be immutable, it can often negatively impact the readability of the produced code. It may not be desirable to inline a complex declaration. On the other hand, it is likely difficult to analyse an arbitrary function, even if the underlying semantics are fairly simple. For example, consider the function `between`, provided in the library. The invocation `between(l, p, r)` simply denotes `l *> p <* r` - in this case, the function is simply shorthand. For these cases, the user should be able to specify when a replacement can be safely performed, via the use of the `inline` ‘declaration’.

```
inline paap(x, y, z) =
  x <**> y <*> z;
inline o = add <$ chr('+');
inline add = (x: number) =>
  (y: number) => x + y;

let r = paap(a, o, b);
```

$\rightsquigarrow$

```
let r = ap(pa(a,
  constFmapL(
    (x: number) =>
      (y: number) => x + y,
    chr('+'))
), b);
```

The snippet demonstrates how a substitution is performed, for substitutions with and without parameters. Notice that the substitution is not restricted to only parsers, as seen with `add`. It is important to note that this does not perform a naïve text replacement, but rather recursively splices in parts of the AST. By modifying the AST, further analysis can be performed in subsequent stages. For example, rather than seeing `r` as being an arbitrary function call, which may have behaviour that cannot trivially be inspected, it is seen as a combinator tree which can be analysed further. The ability to provide these ‘hints’ to the preprocessor is beneficial to the efficacy of optimisations that analyse combinator trees.

Consider the earlier example of `between`. Note that `p = between(l, a, r)` and `q = between(l, b, r)` follow a similar form. However, without knowing how `between` is defined, it is not trivial to deduce that both `p` and `q` share a common prefix. On the other hand, if both were expanded based on the inline definition, it is trivial to see that both parsers have a common prefix and suffix of `l` and `r`, respectively.

Summary

This chapter demonstrates how the raw TypeScript* source code is transformed in order to support numerous language features, including the user-defined operators introduced in chapter 3. The additions to TypeScript are also presented in the context of parser combinators and how a user may find them beneficial. A deeper dive was also made into the underlying design of the preprocessor itself as well as how the parsed structure is traversed and simplified for optimisations.

However, the theory and implementation of the optimisation steps have been omitted for brevity. Instead, they are further explored in chapter 5, which dives deeper into the techniques applied and how they transform the grammar.

Chapter 5

Optimisation and Analysis

The optimisation stage, introduced in chapter 4, forms the crux of the preprocessor. This chapter provides a deeper insight into each of the various optimisations and how they can remedy common pitfalls present in parser combinators. These optimisations include simple changes to the parsers via parser laws to simplify the combinator tree, as well as deeper optimisations which can alter the semantics such as rewriting a grammar to support left recursion. Multiple optimisations that aid in improving performance from backtracking are also explored, including a specific case for alternative strings as well as a more generalised case on combinators.

5.1 Simple Optimisations

5.1.1 Choice Reduction

A common pattern in parsers and grammars are sequences of alternatives, or disjunctions. Since choice (`<|>`) is an associative operator, it can often be useful to treat a tree of alternatives as a sequence, which can be reconstructed in a fully left-associative manner. Note that the order should generally be maintained, as it is not a commutative operator.

While this property is primarily used in subsequent optimisations, it can enable a simple optimisation - removing duplicate disjunctions. For example, if a parser was `p <|> q <|> p`, it could be simply reduced to `p <|> q`. However, by trimming down disjunctions, it reduces the amount of processing required in subsequent steps, most of which rely on dealing with sequences of disjunctions.

5.1.2 Parser Law Optimisation

In order to simplify the parsers further, before more complex optimisations are applied, basic destructive transformations can be performed. Note that this step does not apply all optimisation rules, even if it may allow for further optimisation - it only makes changes that will reduce the size of the parser. Consider the size of the parser as the number of nodes in the combinator tree. Generally, this refers to the neutral, catch, and absorption laws of alternatives, applicatives, and monoidals (previously discussed in subsection 2.2.4). However, this step also applies the definition of 'fmap' when applicable - by performing this reduction, it allows for subsequent preprocessing stages to easily determine whether something is a parser or not.

5.2 Left Recursion Analysis

A common pitfall of recursive descent parsers, and by extension parser combinators, is the inability to handle left-recursion. While techniques exist to manually manipulate a grammar, they can often be mechanical and slow. Additionally, parser combinators typically contain significantly more information than just the parsing structure, often including semantics on how subtrees (of parsers) should be combined. This section introduces how left recursion can automatically be detected and rewritten in a form that can terminate via iteration [6], while preserving the desired, user-defined semantics.

5.2.1 First Sets

In order to begin analysing left recursion, the first step is to determine a programmatic method of checking if a production is left-recursive in the first place. More importantly, the alternatives (if any) that are not left-recursive also need to be deduced.

The first set of a production can be intuitively thought of as anything that can begin the derivation for a particular rule. In a traditional grammar, this is simply a union over the first sets of all disjunctions (alternatives) of a rule. The first set of a rule without disjunctions is the first set of the first element in the rule, which can then be recursively computed. Finally, the first set of a terminal is a set containing only itself. The same intuition can be carried forwards to parser combinators.

Let $\mathcal{F}$ represent the **global** first set and $\mathcal{L}$ represent the **local** first set. The local first set is defined as follows on combinators (note that `i₀` denotes the first character of the string literal that `i` represents (which can be an escape sequence) and `<X>` denotes any arbitrary combinator):

$$\begin{aligned}
\mathcal{L}(\texttt{p <|> q}) &= \mathcal{L}(\texttt{p}) \cup \mathcal{L}(\texttt{q}) && \text{union for choice}\\
\mathcal{L}(\texttt{f <\$ p}) \mid \mathcal{L}(\texttt{f <\$> p}) &= \mathcal{L}(\texttt{p}) && \text{non-parser on left}\\
\mathcal{L}(\texttt{p \$> f}) \mid \mathcal{L}(\texttt{p <\&> f}) &= \mathcal{L}(\texttt{p}) && \text{non-parser on right}\\
\mathcal{L}(\texttt{pure x <X> p}) &= \mathcal{L}(\texttt{p}) && \texttt{pure}\ (\varepsilon)\ \text{first}\\
\mathcal{L}(\texttt{p <X> q}) &= \begin{cases} \mathcal{L}(p) & \text{if } \mathcal{L}(p) \neq \varnothing \\ \mathcal{L}(q) & \text{otherwise} \end{cases} && \text{any other combinator}\\
\mathcal{L}(\texttt{pure x}) \mid \mathcal{L}(\texttt{empty}) &= \varnothing && \text{empty set}\\
\mathcal{L}(\texttt{i: Id}) &= \{\texttt{i}\} && \text{`terminal'}\\
\mathcal{L}(\texttt{a: Atom}) &= \{\texttt{a}\} && \text{`terminal'}\\
\mathcal{L}(\texttt{i: IRLiteral}) &= \begin{cases} \{\texttt{i}_0, \texttt{i}\} & \text{if string / char} \\ \{\texttt{i}\} & \text{otherwise} \end{cases} && \text{`terminal'}
\end{aligned}$$

The computation of the local first set is done via the use of pattern matching in Scala, with recursion when required. Also note that the results of the first set computation is memoised in the shared mutable state, preventing a possibly expensive recomputation at the cost of space.

Note that the computation for the first set of a function is similar, by looking at the first parser in the function's arguments. One important distinction is that the local first set considers identifiers, which are typically references to other parsers (productions), as a terminal, which is incorrect. While this alone allows for some simple left-recursive productions to be detected, it does not yet fully account for indirect left recursion. As the preprocessor has a wider view of the program, it is feasible to obtain the global first set of all productions.

Consider all lexical declarations that can occur in the program. Looking at only parsers (other declarations will have a first set computed, but are meaningless), a declaration refers to any statement that declares a value (reassignments are not supported), such as `p = q`. In this case, `p` is the parser of interest, which is an identifier assignment. The value of the assignment, `q`, can be in any of the forms listed above, including disjunctions.

A relation L, where $\langle p, a \rangle$ denotes a (a parser) as being in the **local** first set of p (an identifier), is defined as the following:

$$L = \{\langle p, a \rangle \mid a \in \mathcal{L}(p)\}$$

The global first set can then be computed as the *transitive closure* of L, such that $\mathcal{F} = L^+$. Note that the transitive closure L^+ refers to the smallest transitive set containing L. A binary relation R on `IR` is defined as transitive if for all $p, q, r \in \texttt{IR}$ if $\langle p, q \rangle$ and $\langle q, r \rangle$ both exist, then $\langle p, r \rangle$ must exist, or formally:

$$\forall p, q, r \in \texttt{IR}\ [\langle p, q \rangle \in R \wedge \langle q, r \rangle \in R \Rightarrow \langle p, r \rangle \in R]$$

This can be computed by adding 'missing' tuples by repeatedly performing relation composition until the relation is transitive. As each step of adding missing tuples is required for transitivity, this creates the minimal set - the transitive closure. In practice, this computation is done by first scanning the entire program's AST for declarations and computing the local first set, creating a mapping from `String` (the identifier) to `Set[IR]`. The transitive closure is then computed by iteratively expanding any identifiers found in a first set to the identifier's respective first set until all identifiers (in the expanded first set) have been visited.

Consider the following example of a parser that contains indirect left recursion:

```
val a  = '1' $> 1;
lazy p = f <$> q <*> a <|> a;
lazy q = p <* '+';
```

The following result is obtained, using the rules for **local** first sets and the subsequent construction of $\mathcal{F}$. Note that expansion is an iterative process, as seen in $\mathcal{F}(\texttt{q})$, where the expansion of `p` leads to further expansion on `a`.

$$\begin{aligned}
\mathcal{L}(\texttt{a}) &= \{\texttt{'1'}\} \\
\mathcal{L}(\texttt{p}) &= \{\texttt{q}, \texttt{a}\} \\
\mathcal{L}(\texttt{q}) &= \{\texttt{p}\} \\
\mathcal{F}(\texttt{a}) &= \{\texttt{'1'}\} && \text{no identifiers} \\
\mathcal{F}(\texttt{p}) &= \{\texttt{p}, \texttt{q}, \texttt{a}, \texttt{'1'}\} && \texttt{q} \mapsto \{\texttt{q}, \texttt{p}\}, \texttt{a} \mapsto \{\texttt{a}, \texttt{'1'}\} \\
\mathcal{F}(\texttt{q}) &= \{\texttt{p}, \texttt{q}, \texttt{a}, \texttt{'1'}\} && \texttt{p} \mapsto \{\texttt{p}, \texttt{q}, \texttt{a}\} \mapsto \{\texttt{p}, \texttt{q}, \texttt{a}, \texttt{'1'}\}
\end{aligned}$$

5.2.2 Detection

By computing the first sets $\mathcal{F}$, it is now possible to detect all three forms of left recursion; direct (from the local first set), indirect (via the transitive closure on local first sets), and hidden (via the rules in place to deal with ε). Recall that a parser is left-recursive when it is able to derive some form, after some number of substitutions (or none), with itself as the first parser [24]. Intuitively, this means left recursion occurs when a parser contains a rule where the derivation begins with itself. As such, a parser `p` is left-recursive if it is contained within its own first set; $\texttt{p} \in \mathcal{F}(\texttt{p})$.

However, simply detecting a parser is left-recursive is not enough to begin rewriting it. The more common form is as follows, where some rules ($\texttt{r}_\texttt{i}$) are left-recursive, but others ($\texttt{q}_\texttt{j}$) are not:

```
lazy p = r_1 <|> ... <|> r_m <|> q_1 <|> ... <|> q_n;
```

Listing 5.1: Example of parser `p` in a general form

In order to rewrite the structure of `p`, the disjunctions must first be partitioned into those which are locally left-recursive (`R`) and those which are not (`Q`). Note that the use of disjunctions(`p`) refers to the set of alternatives for `p`.

$$\begin{aligned}\text{disjunctions}(\texttt{p}) &= \{\texttt{r}_1, \dots, \texttt{r}_\texttt{m}, \texttt{q}_1, \dots, \texttt{q}_\texttt{n}\} && \text{from example above}\\ \texttt{R} &= \{\pi \in \text{disjunctions}(\texttt{p}) \mid p \in \mathcal{L}(\pi)\}\\ \texttt{Q} &= \{\pi \in \text{disjunctions}(\texttt{p}) \mid p \notin \mathcal{L}(\pi)\}\end{aligned}$$

The use of local left recursion (rather than global) allows for safer restructuring - hoisting in declarations may lead to unintended consequences as well as generally more verbose, thus less legible, generated code. In order to allow for declarations that can be safely hoisted in, the lexical declaration `inline` has been implemented, as mentioned in subsection 4.4.4. As such, all three forms of left recursion can be detected, however, only direct left recursion can be rewritten. The other forms are quite infrequent in most grammars, as mentioned in Parr et al. (2014) [4].

5.2.3 Rewriting

It is important to note that by rewriting the productions, the preprocessor will fundamentally alter the semantics of these parsers. Parsers which previously would not terminate due to infinite recursion are now modified to parse iteratively, with termination being possible.

This is done in three steps; normalisation, reduction, and finally, rewriting. The underlying idea behind rewriting left recursion is to convert a ‘standard’ production into a parser that utilises `postfix`. Note that `postfix` was chosen (rather than `infixl`, or similar [6]) as it provided the most ‘general’ form - it is able to automatically handle infix operators as well as postfix operators. This property is desirable, especially in the context of automatic rewriting; while the generated program may be less concise, it allows for more forms of detection, as well as a reduced likelihood of errors, thus requiring less manual intervention.

Consider the following (direct) left-recursive production, similar to the previous declaration of `p`;

$$P \to \underbrace{\overbrace{Ps_1}^{r_1} \mid \dots \mid \overbrace{Ps_m}^{r_m}}_{\texttt{R}} \mid \underbrace{q_1 \mid \dots \mid q_n}_{\texttt{Q}}$$

On a translation to a PEG, the recursive productions simply become repetition operators as stated in Ford (2004) [16], thus the following result is obtained (CFG to PEG);

$$P \leftarrow (q_1 \ / \ \dots \ / \ q_n)(s_1 \ / \ \dots \ / \ s_m)*$$

This result is similar to that of Hill (1994) [25]. Note that P may appear in s_j, such as in the case of addition ($E \to E$ `'+'` E) where $P = E$ and $s_j =$ `'+'` E. However, the rewriting remains valid as long as $P \notin \mathcal{F}(s_j)$. Note that the rewrite still occurs if $P \in \mathcal{F}(s_j)$ but $P \notin \mathcal{L}(s_j)$, however, the preprocessor will raise a warning regarding a possible indirect or hidden left recursion.

However, this translation of a CFG to a PEG must be mirrored in terms of parser combinators, which not only carries syntactic information in terms of the parse structure, but has underlying semantics in terms of how the parsed results are used (or not used). This raises a number of challenges: the same grammar can exist in multiple forms (lack of normalisation) and combinators carry semantic information as well as syntactic information.

The first of which is that parser combinators can exist in a number of forms which represent the same underlying grammar. For example, the four following combinators all represent $P_i \to P_i S \mid Q$:

```
lazy p_0 = f_0 <$> p_0 <*> s <|> q;
lazy p_1 = p_1 <**> pure(f_1) <*> s <|> q;
lazy p_2 = f_2 <$> p_2 <~> s <|> q;
lazy p_3 = lift2(f_3, p_3, s) <|> q;
```

The second problem is preserving the semantics of the parse - note that in the example above, there is an additional transformation that is applied to each of the $P_i S$ disjunctions. While it may be quite intuitive to reason about the behaviour of the function at a glance, any changes to the parser, which may be numerous, must be accurately reflected in how the function is transformed (recall the rewriting of certain functions from subsection 4.3.1). Without this constraint, a rewritten parser would simply verify if the structure is correct and give the sequence of parsers applied - this is only sufficient for recognisers. While parser generators will still require tree reassociation, the semantics are provided 'externally' (outside the parser generator) over the parse tree, which is not the case for parser combinators.

Normalisation

The first problem is addressed via normalisation and reduction. In order to aid the subsequent steps, any locally left-recursive parsers are matched against a series of patterns in order to extract the required elements. The chosen normal form for the majority of operators is `p` = `lift2(f, q, r)` - this allows for a clear separation between three key components; `f` (an uncurried function), `q` (the left recursion), and `r` (the 'remainder' or suffix). Notice that the left recursion does not necessarily have to be `p` - just that $\mathtt{p} \in \mathcal{L}(\mathtt{q})$.

Note that any patterns using 'reverse fmap' (`<&>`) will be omitted for brevity; simply replace a use of `f <$> p` with `p <&> f`, additionally patterns using 'const fmap' such as `x <$ p` can be substituted with `p $> x`. Some normalisation patterns are as follows:

```
p match {
    case (f <$> q) <*> r          => Lift2(Uncurry(f), q, r)
    case (f <$> (q <~> r))        => Lift2(ArgsToTuple(f), q, r)
    case (q <**> Pure(f)) <*> r   => Lift2(Uncurry(f), q, r)
    case q <**> (f <$> r)         => Lift2(Uncurry(Flip(f)), q, r)
    case (q <**> (f <$ op)) <*> r => Lift2(Uncurry(f), q, op *> r)
    // ...
}
```

Listing 5.2: Examples of normalisation steps for common patterns

While these patterns cannot account for every possible combination, when coupled with the later reduction steps, they can deal with a variety of parsers in various common forms. The validity of these normalisation steps is detailed in section 6.4.

Reduction

Recall that the recursive component of `p` = `lift2(...)` does not necessarily have to be `p`. However, the end goal is to 'reduce' this component down to just be `p`. This is done via a sequence of reduction steps, which monotonically simplifies the recursive component; the complexity of a component can be quantified as the size of the combinator tree representing it. By requiring a constraint where the structure is monotonically simplified, termination is guaranteed - resulting in a successful reduction or a failed reduction (which causes the preprocessor to roll back any transformations made on a particular disjunction).

Similar to the normalisation step, some patterns will be omitted for brevity if they are equivalent to another pattern, albeit with flipped arguments. Additionally, `<*` and `<~` are equivalent and thus have the same reduction rules. Function shorthand will also contain `UFA(f)` and `MFA(f)` to represent `UnpairFirstArg(f)` and `MapFirstArg(f)`, respectively.

```
p match {
    case Lift2(f, r <* s, q)   => _Lift2(f, r, s *> q)
    case Lift2(f, g <$> r, q)  => _Lift2(MFA(g, f), r, q)
    case Lift2(f, c <$ r, q)   => _Lift2(MFA(Const(c), f), r, q)
    case Lift2(f, r <~> s, q)  => _Lift2(UFA(f), r, s <~> q)
    case Lift2(f, r <*> s, q)  => _Lift2(f, AFM <$> (r <~> s), q) // (1)
    case Lift2(f, r <**> s, q) => _Lift2(f, PFM <$> (r <~> s), q) // (1)
    case Lift2(f, r <:> s, q)  => _Lift2(f, ((:) <$> r) <*> s, q) // (2)
    // ...
}
```

Listing 5.3: Some `lift2` reduction cases, note that `_Lift2` is a 'smart constructor' that performs further reductions

The validity of these reductions is proven via equivalences in section 6.3. Note that the rules for `<*>` and `<**>` bootstrap off of the existing reduction steps for `<~>` and `<$>`, based on the idea of recovering applicative from monoidal, as stated by McBride & Paterson (2008) [26]. As such, the reduction is implemented in a way that adds complexity to begin with but reduces in a fixed, finite number of steps to a simpler form, thus maintaining the monotonically decreasing constraint. The same idea is applied for `<:>` (lifted cons), however this also builds on the use of `<*>`.

Postfix

Prior to this point, the parsers were still left-recursive. Nothing has been done to change the semantics yet, however left-recursive disjunctions have been 'reconditioned'; the aforementioned problems (the lack of a normal form and the preservation of parse semantics) have been dealt with. Recall the earlier example of `p`, shown in listing 5.1. In the ideal case, this parser would have become reconditioned to the following;

```
lazy p = lift2(f_1, p, s_1) <|> ... <|> lift2(f_m, p, s_m) <|>
         q_1 <|> ... <|> q_n;
```

Listing 5.4: Example of parser `p`, after normalisation and reduction

Using the previous partitioning of `Q` and `R`, where the latter represents left-recursive disjunctions, it follows that the newly created `lift2`s fall into the `R` partition. Note that `postfix` and `lift2` have the following types - additionally, in the case of `lift2` there is the constraint that `C` = `A`, as `p` has the type `P<A>`.

```
function postfix(q: P<A>, op: P<(a: A) => A>): P<A>;
function lift2(f: (a: A, b: B) => C, p: P<A>, q: P<B>): P<C>;
```

Listing 5.5: Types involved for postfix conversion

Using these types as guide, the next natural step is to populate the parameters of `postfix` as required, resulting in a parser that replaces recursion with iteration. The semantics of `postfix` are that it parses one `q` and then zero or more occurrences of `op`, applying the result to some accumulating value. Naturally, `q` must be the non-recursive cases found in the `Q` partition. However, `Q` represents a collection of parsers, which can trivially be changed into a parser by reducing as alternatives (`<|>`).

The remaining work lies in converting the recursive cases into `ops` (which can then be reduced in the same way). Recall that work was done to ensure that `lift2` followed the same structure, where `p` is isolated in all alternatives. Consider a single recursive disjunction, `lift2(f, p, s)` $\in$ `R`. If the value for `b` (the second argument) were to be pre-populated in `f`, the resulting function would be ideal for `postfix`.

This can be seen in listing 5.6, where the parser simply subtracts 1 from the number parsed. In this example, the argument `b` is represented by `y`. However, it is clear that this will be 1: the result of the second 'parser' (`pure(1)`). If this value were to be populated, the resultant function would resemble `(x: number) => x - 1`, which is the desired effect. Of course, this would have to be done to accommodate an arbitrary parsed result, not just a constant.

```
let sub1 = lift2((x: number, y: number) => x - y, nat, pure(1));
```

Listing 5.6: Simple example of `lift2` to demonstrate pre-populating an argument

This is done by first currying `f`, which has the type `(a: A) => (b: B) => A`. By flipping this curried function, the resultant type is `(b: B) => (a: A) => A`. Finally, this can be partially applied with 'fmap' (`<$>`), with the result (`flip(curry(f)) <$> r`) being of the desired type. This transformation is performed on all alternatives in `R`, which is then reduced. Validity of this transformation is further discussed in section 6.2.

The earlier example, shown in listing 5.1 and listing 5.4, is finally rewritten into:

```
lazy p = postfix(q_1 <|> ... <|> q_n,
    flip(curry(f_1)) <$> s_1 <|> ... <|>
    flip(curry(f_m)) <$> s_m
);
```

Listing 5.7: Example of parser `p`, after rewriting

5.2.4 Worked Example

Recall the example of addition first introduced in listing 2.1 and the subsequent IR in listing 4.1. The code matches the pattern `q <**> (f <$ op) <*> r` exactly, with no requirement for reduction. In this case (let `C('+')` denote `Chr(IRLiteral("'+'"))` for brevity):

$$\begin{aligned} \texttt{q} &= \texttt{Id("expr")} & \qquad \texttt{op} &= \texttt{C('+')} \\ \texttt{f} &= \texttt{Atom(0)} & \qquad \texttt{r} &= \texttt{Id("nat")} \end{aligned}$$

Trivially, this is converted into `lift2(u(f), Id("expr"), C('+') *> Id("nat"))`. As this is already in the desired shape, where the left recursion is the first parser, it can be converted into a `postfix` operation. This is then rewritten into the following (the violet component is the result of the function after converting from the IR):

$$\underbrace{\texttt{flip(curry(u(f)))}}_{\texttt{(y) => (x) => x + y}}\ \texttt{<\$> (C('+') *> Id("nat"))}$$

Notice how the first application, which would be to the result of `Id("nat")`, is the argument `y`, thus pre-populating the function as desired.

5.3 String Trie

A frequently observed pattern in parsing, especially in programming languages, is a number of alternative strings, namely keywords. However, this pattern can often lead to two pitfalls; the first revolves around the ordering of words and the second revolves around the performance penalty caused by backtracking. The execution of disjunctions in parser combinators follows the behaviour of PEGs. As such, the 'dangling else ambiguity' can be easily resolved by a reordering of disjunctions, with the longest pattern first. This also means that disjunctions are not commutative (therefore, this optimisation changes the semantics of the parser). Consider an example consisting of a subset of TypeScript keywords:

```
kw ::= 'as' | 'async' | 'break' | 'case' | 'const' | 'continue'
```

In this order, if the word `'async'` were to be parsed, the parser would terminate on a successful parse of `'as'`, with a remainder of `'ync'`. On the other hand, if the order were to be flipped, with `'as'` being after `'async'`, the parser would fail due to the input being partially consumed - backtracking is therefore required to reset the state back to where the first parser began. However, backtracking operations are inherently expensive, possibly requiring work to be redone. Ideally, a parser would have minimal (if any) backtracking (implemented via the use of `attempt`), and any parsers that contain backtracking would be as 'small' as possible.

As noted in Swierstra (2000) [27], constructing a 'trie' structure allows for possible productions to be grouped by common prefixes, thus eliminating the need for backtracking - ambiguities are resolved as they would be separated into choices that have no conflict. Without any backtracking, the parser is able to complete in linear time, after this data structure has been constructed. Rather than having this pattern detection and construction being performed at runtime (in TypeScript), this can be done during transpile time (in Scala, within the preprocessor).

While this approach can limit the effectiveness due to its inability to inspect values only known at runtime, most common patterns (such as the motivating example) leverage string literals as keywords, which can trivially be accessed and analysed by the preprocessor. By moving the cost of the analysis to the preprocessor, it removes the performance penalty of constructing the trie on the initial run of the parser, which would likely cause an overall performance degradation if the parser is not commonly used or is sufficiently small.

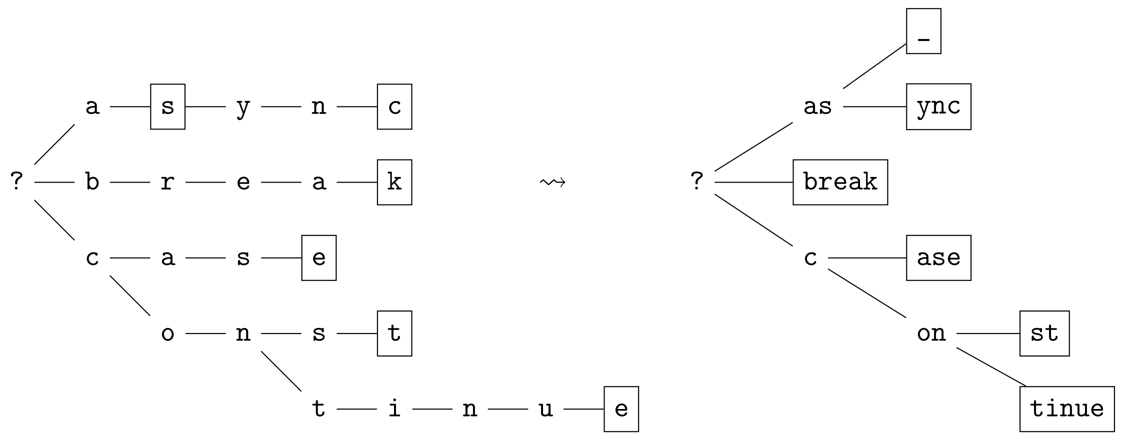


Figure 5.1: Left: uncompressed (naïve) trie, right: compressed (radix) trie. The initial `?` denotes the root, a box denotes a 'complete' word, and `_` denotes the empty (ε) string, `pure("")`.

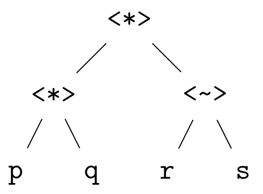
The preprocessor implements this optimisation by first detecting chains of `Str` or `Chr` (in the IR) and attempting to extract a list of string or character literals. These literals are then inserted into an uncompressed trie (one character at a time) in order to allow for escape sequences to be easily handled. Once all words have been inserted, the trie is compressed, as seen in Figure 5.1. This structure is then converted back into a parser bottom-up; the order of alternatives does not matter as long as the empty string (if present) is at the end. Note that a bottom-up (leaf to root) construction is only valid if the combinators or operations to join the parsers are associative, or are naturally right-associative. Fortunately, in the case of string concatenation, this holds.

5.4 Backtracking Reduction

As mentioned in section 5.3, there is a significant performance penalty incurred by backtracking. However, the technique used for strings does not generalise well to arbitrary sequences of combinators due to the lack of right-associativity. This section explores alternative techniques and methods to extract 'common' terms in an attempt to reduce backtracking[1].

5.4.1 Flat IR and Naïve Fusion

While the current tree-based IR allows for effective analysis on the localised **structure** of the parse, it can often become cumbersome when reasoning about the **sequence** of the parse; the order in which parsers are applied (and more importantly, consume input). This becomes particularly tedious when inspection needs to be done on the whole parser tree, when the leftmost and rightmost leaves need to be inspected. Due to this limitation, inspection in this stage is done primarily using the `FlatParser` type, which is a sequence of `FlatIR`s. Note that `FlatIR` is a trait which contains two notable concrete implementations; `P` and `C` - the former wraps an `IR` and the latter represents a `Combinator`. The conversion is done respecting left-associativity; therefore traversal is only performed on the left-hand side of a combinator, as seen in Figure 5.2.



```
⇝   [P(p), C(<*>), P(q), C(<*>), P(r <~> s)]
```

Figure 5.2: Left: tree-based (original) IR, right: list-based (flat) IR. Both represent the parser `p <*> q <*> (r <~> s)`.

Since left-associativity is respected when converting from the combinator tree to the flat representation, recovery can be done without any concerns of changing associativity. However, this approach limits the amount of inspection that can be performed, especially when considering common terms. For example, in Figure 5.2, inspection cannot be done within `<~>`, as it is considered a single parser.

Certain combinators, namely the label operation (`<?>`) and choice (`<|>`) are not flattened. Any other nodes in the IR, including functions, are kept as 'singleton' parsers.

This representation allows for a naïve implementation of factoring. Two disjunctions can be 'fused' when everything, other than the last element in the list, is equal between both parsers. This element can then be combined by utilising the distributive property of alternatives on applicatives. Note that this form of fusion is omitted from the pipeline, as another technique discussed later is generally more effective.

[1]No pun intended

Also note that the left-distributive property[2] is only applicable for backtracking parsers and parsers with non-biased choice (the *Parsec* family is left-biased). However, as all disjunctions require backtracking in order to benefit from this optimisation, it can be assumed to hold within this optimisation. The right-distributive[3] property does not hold for parsers [28], as demonstrated in listing 5.8 with an input of `'+++'`. In this scenario, `c` will only be able to succeed if `b` succeeds (if `a` succeeds, there will be insufficient `'+'`s for `c`) - this case is possible in `p`, as the first disjunction will fail. However, if `q` was to be used, the first conjunction would succeed with `a` succeeding, however `c` will then fail, causing the entire parser to fail. Furthermore, factoring on the right does not offer a performance benefit - the same amount of backtracking would still be performed, however code size may be reduced.

```
let a = str('++');
let b = str('+');
let c = str('++');
let p = attempt(a *> c) <|> attempt(b *> c);
let q = (attempt(a) <|> attempt(b)) *> c;
```

Listing 5.8: Example showing failure of right factoring, `p` is the original parser and `q` is the attempted optimisation

5.4.2 Left-Associative Normalisation

The crux of this technique is to 'fuse' similar structures with alternative parsers. In order to maximise the efficacy of this technique, a normal form should be established as it permits parser structures that are semantically equivalent that otherwise wouldn't match based on structure to be fused.

Similar to before, cases for `<~` and `~>` are interchangeable with `<*` and `*>`, respectively. The following associativity and reassociation equivalences are applied in order to obtain a normal form:

$$\begin{aligned} \texttt{p <* (q <* r)} &\Rightarrow \texttt{p <* q <* r} \\ \texttt{p *> (q *> r)} &\Rightarrow \texttt{p *> q *> r} \\ \texttt{p *> (q <*> r)} &\Rightarrow \texttt{p *> q <*> r} \\ \texttt{p <*> (q <* r)} &\Rightarrow \texttt{p <*> q <* r} \end{aligned}$$

While this alone does not guarantee everything is converted into a normal form, it increases the amount of left-associated operators. By performing this step, further steps that traverse the structure of the combinator tree are able to progress further without requiring additional logic for matching equivalent structures.

5.4.3 Fusion with Merkle Trees

As mentioned earlier, a key limitation of the flattened structure is the inability to analyse the right subtree of a combinator, when it is not atomic (when the parser within `P` is actually a combinator). An example of this can be seen in Figure 5.3; which would otherwise be fused to `(r <~> s)/(t <~> s)`. Another key limitation is that fusion on the flattened IR requires both to have the same number of elements.

[2] `a <*> (b <|> c) = (a <*> b) <|> (a <*> c)`
[3] `(a <|> b) <*> c = (a <*> c) <|> (b <*> c)`

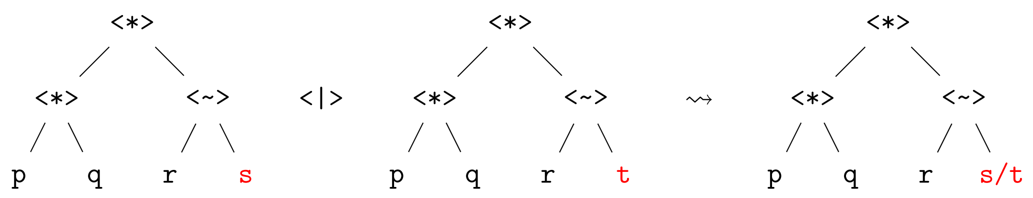


Figure 5.3: Left: two parser trees before fusion, right: resulting parser tree after fusion. Nodes in `red` are fused.

However, the problem of efficiently detecting small differences between two trees and performing some aggregation or synchronisation mirrors that of large-scale data management systems such as *DynamoDB*, as described by DeCandia et al. (2007) [29]. One of the data structures used for 'anti-entropy' is a Merkle tree [30].

Each node contains a hash which consists of the hashes of its children - for the combinator tree, this also needs to account for the operator itself. Another modification for traversal on combinator trees is that the termination condition is not only on the leaf but rather when a mismatch is detected that prevents further traversal.

The algorithm consists of two components; 'matching', where two trees are checked for compatibility and the wrapper, which performs the fusion across disjunctions. The following describes the matching step. Assuming that the nodes represent the same combinator and are not equal, the general rule is to first determine whether the right side of the combinator can be fused. If the left side is equal, it is traversed further until it finds the first 'difference' (the combinators are not equal or only one of the two arguments are combinators), at which point a fusion occurs. For example, if a parser is factorable on the left and the left sides are equal, then the fusion function is called with the two right-hand sides. Note that if this is the first call of the function, no fusion is performed. These rules are detailed further in algorithm 1.

Algorithm 1 Matching over Merkle trees, LF denotes left-factorable

```
Require: $\text{Type}(a) = \text{Type}(b)$
1: function $\text{Match}(a, b, f)$                              ▷ $a$ originates from the candidate set
2:     if $a = b$ return $\text{Some}(a)$
3:     $\text{sc} \leftarrow \text{SameCombinator}(a, b)$              ▷ also checks if both are combinators
4:     if sc and LF and $a.l = b.l$ return $\text{Match}(a.r, b.r, \bot)$ map $\lambda r \to \text{Op}(a.l, r)$
5:     if $\text{ToFuse}(irs) \leftarrow a$ return $\text{Some}(\text{ToFuse}(b :: irs))$
6:     if $f$ return None
7:     return $\text{Some}(\text{ToFuse}([a, b]))$
8: end function
```

The fusion stage wraps the matching stage. Initially, the set of candidates is initialised to the empty set. For each disjunction, a match is attempted across each of the candidate trees, with the first match being taken (if any). If a match is successful, the original tree (from the candidate set) is replaced with the new tree (containing the fusion with the disjunction). Otherwise, the disjunction is added directly to the candidate set. The purpose of the candidate set is to maintain existing matches as an accumulator, preventing redundant computation and additional parsers from being created. In practice, this fusion is recursive (the newly combined parser is further optimised) - however, this is done at the end. These steps are detailed in algorithm 2.

Algorithm 2 Wrapper function for fusion using Merkle trees.

```
Require: $\exists T\ [\forall ir \in irs\ [\textsc{Type}(ir) = T]]$          ▷ all have same type
 1: function FuseTrees($irs$)                                                  ▷ primary function
 2:     $candidates \leftarrow \varnothing$
 3:     for all $ir \in irs$ do
 4:         if $\exists c \in candidates\ [\textsc{Some}(f) \leftarrow \textsc{Match}(c, ir, \top)]$ then
 5:             $candidates \leftarrow (candidates - \{c\}) \cup \{f\}$
 6:         else
 7:             $candidates \leftarrow candidates \cup \{ir\}$
 8:         end if
 9:     end for
10:     return $candidates$ map Expand                                          ▷ recursive expand
11: end function
```

Recall that this analysis occurs over disjunctions on a parser. This provides a crucial invariant; the types of each disjunction must be the same. As long as this invariant is maintained, the fusions are type-safe. For example, consider two parsers, `p <X> q` and `p <X> r` - if they are of the same type, then `q` and `r` must also have the same type. A recursive step on `q` and `r` will also therefore maintain this invariant. The same argument is made for the other side of the combinator.

5.4.4 Ordering with Tries

Tries also provide another benefit; they can be used to determine the ideal ordering of parsers to prevent the 'dangling else' ambiguity. This property is used for the 'string tries' optimisation discussed in section 5.3. However, in order to generalise this to parsers, the preprocessor must first establish the sequence in which parsers execute within the disjunction. A sequence can be obtained using the existing flattened IR, however, expansion should also be done on the right-hand side. Recall that the purpose of only expanding on the left-hand side was to allow for easier reconstruction due to left-associativity - a property that is no longer required as this parse sequence will not be reconstructed.

Again, consider the example presented in Figure 5.3. Suppose no fusion had occurred, thus only looking at the two disjunctions. The two combinator trees would then be flattened into `[P(p), C(<*>), P(q), C(<*>), P(r), C(<~>), P(s)]` and `[P(p), C(<*>), P(q), C(<*>), P(r), C(<~>), P(t)]` respectively. However, when considering the order in which parsers are applied, the combinators are no longer required and can therefore be stripped out, leaving the sequence to be `[p, q, r, s]` and `[p, q, r, t]`, respectively. Note the process of retaining only parsers in the example above is overly simplified. In reality, larger portions of the flattened parser are observed at a time, similar to peephole optimisation [31]. This filters out 'non-parsers', such as functions when used with `<$>`.

While this method of determining the parser sequence allows for an order to be established, it does not allow for parsers to be recovered since the combinators have been discarded. In order to remedy this, the structure of a trie is modified; rather than marking a node as being complete, it needs to contain a collection of parsers, which is initially empty. On completion of an insert, when the sequence of parsers is empty, the original parser (before flattening) must be added to the collection of complete parsers in a given node. A post-order traversal can then be performed on the trie, where the completed collection is appended to the end of the results of its child nodes. Since each of the child nodes represents different parsers (with ambiguity being accounted for), adding the result at the end allows for the 'shortest' parser to be performed last.

5.4.5 Overall Process

The previous sections discussed multiple techniques to deal with an abundance of backtracking as well as poorly ordered disjunctions. However, the actual implementation relies on all three methods: normalisation, fusion with Merkle trees, and trie ordering.

As a `PipelineStage`, the entry point of this optimisation exists when processing an `IR`. Note that this is only run when it detects a sequence of disjunctions, all of which are wrapped in an `attempt`, as this permits for backtracking - a property that many of the optimisations rely on for fusion. Once this is detected and collected as a sequence of disjunctions (with the `attempts` stripped), normalisation is performed in order to maximise the efficacy of later analysis. Next, fusion is performed using the Merkle tree - note that any fusion that occurs in this step is recursive and will undergo the same process. At this stage, the result is still a collection of disjunctions, hopefully containing fewer elements than the original due to fusion.

The final step is to combine the disjunctions. However, recall that backtracking was removed for the sake of processing. If the parsers were naïvely combined, with no regard to backtracking, there would likely be ambiguities between disjunctions, however, if all parsers were wrapped and combined, the primary purpose of this optimisation would be lost. ‘Ambiguous’ parsers are first wrapped with an `attempt` and then fed into the trie to perform an ordering, whereas parsers with no ambiguity are directly inserted into the tree - note that the sequence of parsers for a wrapped parser and unwrapped parser is the same. Not only does this process account for ambiguity, redundant uses of `attempt` are also removed.

An ambiguity between two parsers `p` and `q` occurs when there is some element that exists in the first set of both parsers (defined in subsection 5.2.1);

$$\text{ambiguous}(\texttt{p}, \texttt{q}) = \exists r\ [r \in \mathcal{F}(\texttt{p}) \land r \in \mathcal{F}(\texttt{q})]$$

This is a minor simplification - additional logic is in place to prevent two ‘different’ parsers from not being a match. For example, two `satisfy` parsers may have two functions that are different but can be satisfied by the same character, thus consuming input. The same can be said for `Atoms` or identifiers, which cannot be easily inspected - this further motivates the use of the ‘inlining’ functionality introduced in subsection 4.4.4. In these cases, where it is non-trivial to inspect, the parsers are marked as ambiguous.

Finally, the trie is traversed, which gives the desired ordering (with ambiguous parsers being wrapped). This can then be reduced by combining all terms with the choice operator.

Summary

Throughout this chapter, a number of techniques for improving the performance or usability of parsers have been explored. These techniques range from ‘lawful’ optimisations based on destructive parser laws, to automated grammar refactoring which fundamentally changes the semantics of a parser, to optimisations that reduce the amount of backtracking performed. The efficacy of these optimisation stages is analysed further in section 7.1.

In the case of grammar refactoring, a number of rules were stated with little to no evidence, aside from intuition. These equivalences and conversions are explored in greater detail in chapter 6, where the soundness of the aforementioned transformations are shown.

Chapter 6

Soundness

In section 5.2, a number of transformations were introduced in order to rewrite left-recursive productions into an iterative form, utilising `postfix`. This chapter dives deeper into these transformations to prove equivalences in the normalisation (section 6.4) and reduction (section 6.3) steps, as well as verifying the correctness of rewriting a recursive production in an iterative manner (section 6.2).

6.1 Equivalences

Throughout these proofs, the following applicative equivalences are used;

$$\texttt{p <**> q} = \texttt{pure (flip (\$)) <*> p <*> q} \tag{I}$$
$$\texttt{p <*> pure f} = \texttt{pure (\textbackslash g -> g f) <*> p} \tag{II}$$
$$\texttt{pure f <*> pure g} = \texttt{pure (f g)} \tag{III}$$
$$\texttt{p <*> (q <*> r)} = \texttt{pure (.) <*> p <*> q <*> r} \tag{IV}$$
$$\texttt{f <\$> p} = \texttt{pure f <*> p} \tag{V}$$
$$\texttt{p <*> (q <* r)} = \texttt{p <*> q <* r} \tag{VI}$$
$$\texttt{pure f <*> (pure g <*> p)} = \texttt{pure f.g <*> p} \tag{VII}$$
$$\texttt{pure f <*> p <*> pure g} = \texttt{pure (\textbackslash h -> h g).f <*> p} \tag{VIII}$$
$$\texttt{p <* q} = \texttt{pure const <*> p <*> q} \tag{IX}$$
$$\texttt{p *> q} = \texttt{pure (flip const) <*> p <*> q} \tag{X}$$
$$\texttt{p <\textasciitilde> q} = \texttt{pure (,) <*> p <*> q} \tag{XI}$$
$$\texttt{lift2(f}_{\texttt{u}}\texttt{, p, q)} = \texttt{pure f <*> p <*> q} \tag{XII}$$
$$\texttt{f <\$ p} = \texttt{pure (const f) <*> p} \tag{XIII}$$
$$\texttt{p <*> q} = \texttt{(\textbackslash(p,q) -> p q) <\$> (p <\textasciitilde> q)} \tag{XIV}$$
$$\texttt{flip (\$)} = \lambda xg \rightarrow g\ x \tag{XV}$$
$$\texttt{flip} = \lambda fxy \rightarrow f\ y\ x \tag{XVI}$$
$$\texttt{const} = \lambda pq \rightarrow p \tag{XVII}$$
$$(f \cdot g)\ x = f\ (g\ x) \tag{XVIII}$$

(I) definition of derived combinator 'reverse ap' (`<**>`)

(II) interchange law

(III) homomorphism law

(IV) composition law

(V) definition of derived combinator 'fmap' (`<$>`)

(VI) 'reassociation' law, can be derived using (IX)

(VII) frequently used pattern, derived from (III) and (IV)

(VIII) frequently used pattern, derived from (II), (III), and (IV)

(IX) definition of derived combinator 'ap left' or 'then-discard' (`<*`)

(X) definition of derived combinator 'ap right' or 'then' (`*>`)

(XI) construction of monoidal from applicative

(XII) definition of 'lift2'

(XIII) definition of derived combinator 'const fmap' (`<$`)

(XIV) recovery of applicative from monoidal

Equivalences (II), (III), (IV), (XI), and (XIV) are noted by McBride & Paterson (2008) [26]

6.2 Rewriting Left Recursion

Let two parsers be defined as 'similar' if the intended result are the same given the same input. The goal is to show that the parser `lift2(`$\mathtt{f_u}$`, p, q) <|> a` can be written as `postfix a (flip f <$> q)` (similar). Correctness of this is established when an input sentence $\mathcal{S}$ gives the expected result on both parsers. However, equivalence cannot be directly established since the behaviour is fundamentally different; the first parser will infinitely recurse due to the left-recursive production, whereas the goal of the refactoring is to obtain a parser that terminates via iteration.

Note that the parser results in the following grammar; $p \to pq \mid a$. However, this is clearly left-recursive, following the basic rewriting steps, the resultant productions are $p \to ap'$ and $p' \to qp' \mid \varepsilon$. For brevity, let this be denoted as $p \to aq*$, where $q*$ denotes zero or more occurrences of q.

The following definition, adapted from Willis & Wu (2021) [6], is used for `postfix`:

```
postfix a op = a <**> rest
  where
    rest = flip (.) <$> op <*> rest <|>
                  pure id
```

Prior to performing induction over the structure of $\mathcal{S}$, certain cases can be analysed and reasoned over. The primary cases are when the sentence does not match at all, the sentence partially matches (when the sentence begins with a valid production but does not entirely match), and finally, when the entire sentence matches.

In the case that $\mathcal{S} = s$, where s does not start with a (hence an invalid production), both parsers would fail; thus they are trivially 'similar'. Similarly, consider the case where $\mathcal{S} = ps$, where s does not start with q; after p is parsed, the next step for both parsers would be an attempt to parse q on s, which will fail. In this case, the parsers are similar for $\mathcal{S} = ps$ as long as the parsers are similar for $\mathcal{S} = p$.

Consider the non-trivial case where the input string exactly follows the production of p (such that $\mathcal{S} = p$). The proof follows induction over the structure of p.

6.2.1 Base Case

The goal in this case is to prove that `p` = `postfix a (flip f <$> q)` results in `a` (similar), when the input is $\mathcal{S} = a$.

$$\texttt{p} = \texttt{a <**> pure id} \qquad (1)$$

$= \texttt{flip (\$) <\$> a <*> pure id}$ (I)

$= \texttt{pure (flip (\$)) <*> a <*> pure id}$ (V)

$= \texttt{pure (\textbackslash f -> f id).(flip (\$)) <*> a}$ (VIII)

$= \texttt{pure id <*> a}$ (2)

$= \texttt{a}$

(1) Since $\mathcal{S} = a$, the first use of `rest` takes the second branch (`pure id`) as the first branch will fail.

(2) Remaining work is to prove that `g` = `(\f -> f id).(flip ($))` is equivalent to the identity function.

$$\begin{aligned}
\texttt{id} &= \lambda x \to x \\
\texttt{g} &= (\lambda f \to f\ (\lambda x \to x)) \cdot (\lambda xy \to y\ x) \\
&= \lambda x \to (\lambda f \to f\ (\lambda x \to x))((\lambda xy \to y\ x)\ x) \\
&= \lambda x \to (\lambda y \to y\ x)(\lambda x \to x) \\
&= \lambda x \to (\lambda x \to x)\ x \\
&= \lambda x \to x
\end{aligned}$$

6.2.2 Inductive Case

For brevity, let $r = aq*$, and assume that this property holds for $\mathcal{S} = r$. The goal in this case is to prove that `postfix a (flip f <$> q)` results in $\texttt{lift2(f}_\texttt{u}\texttt{, r, q)}$, when the input is $\mathcal{S} = (aq*)q$.

Auxiliary Result on postfix

In order to prove this, the first step is to show that `postfix a (flip f <$> q)` is similar to `postfix r (flip f <$> q)` on $\mathcal{S} = (aq*)q$. To formalise this, take the input as $\mathcal{S} = aq^n q = aq^{n+1}$ (arbitrary $n \geq 0$), where q^k denotes k occurrences (successful parses) of q.

In the base case, let $m = 0$. This can easily be shown to hold, as $aq^0 = a$, therefore `postfix a (flip f <$> q)` is trivially equivalent to `postfix a (flip f <$> q)`. In the inductive case, assume that this holds for $m = k - 1$. The assumption states that `postfix a (flip f <$> q)` is similar to $\texttt{postfix aq}^{\texttt{k-1}}\texttt{ (flip f <\$> q)}$.
Using this assumption, it is sufficient to prove that $\texttt{postfix aq}^{\texttt{k-1}}\texttt{ (flip f <\$> q)}$ is similar to $\texttt{postfix aq}^{\texttt{k}}\texttt{ (flip f <\$> q)}$. First, $\texttt{aq}^\texttt{k}$ is defined as;

$$\begin{aligned}
\texttt{aq}^0 &= \texttt{a} \\
\texttt{aq}^\texttt{k} &= \texttt{lift2(f}_\texttt{u}\texttt{, aq}^{\texttt{k-1}}\texttt{, q)} \\
&= \texttt{aq}^{\texttt{k-1}}\texttt{ <**> (pure (flip f) <*> q)}
\end{aligned}$$

Let $\texttt{b} = \texttt{aq}^{\texttt{k-1}}\texttt{ <**> (pure (flip f) <*> q)}$, in order to verify the equivalence stated above;

$\texttt{b} = \texttt{pure (flip (\$)) <*> aq}^{\texttt{k-1}}\texttt{ <*> (pure (flip f) <*> q)}$ (I)

$= \texttt{pure (.) <*> (pure (flip (\$)) <*> aq}^{\texttt{k-1}}\texttt{) <*>}$ (IV)
$\texttt{pure (flip f) <*> q}$

$= \texttt{pure (.).(flip (\$)) <*> aq}^{\texttt{k-1}}\texttt{ <*> pure (flip f) <*> q}$ (VII)

$= \texttt{pure } \overbrace{\texttt{(\textbackslash g -> g (flip f)).((pure (.).(flip (\$)))}}^{\beta}\texttt{ <*>}$ (VIII)
$\texttt{aq}^{\texttt{k-1}}\texttt{ <*> q}$

$$
\begin{aligned}
&= \texttt{pure f <*> aq}^{\texttt{k-1}}\ \texttt{<*> q} \\
&= \texttt{lift2(f}_{\texttt{u}}\texttt{, p, q)} && (\text{XII}) \\
\beta &= \lambda x \to (\lambda g \to g\ (\texttt{flip}\ f))(((\cdot) \cdot (\lambda xg \to g\ x))\ x) \\
&= \lambda x \to (\lambda g \to g\ (\texttt{flip}\ f))((\lambda g \to g\ x) \cdot) \\
&= \lambda x \to (\lambda g \to g\ x) \cdot (\texttt{flip}\ f) \\
&= \lambda xy \to (\lambda g \to g\ x)((\texttt{flip}\ f)\ y) \\
&= \lambda xy \to (\texttt{flip}\ f)\ y\ x \\
&= \lambda xy \to f\ x\ y
\end{aligned}
$$

The goal is to now show that $\texttt{c} = \texttt{postfix aq}^{\texttt{k-1}}\ \texttt{(flip f <\$> q)}$ is similar (for a sentence $\mathcal{S} = aq^n$ where $n \geq m$) to $\texttt{d} = \texttt{postfix aq}^{\texttt{k}}\ \texttt{(flip f <\$> q)}$.

$$
\begin{aligned}
\texttt{R} &= \begin{cases} \texttt{pure id} & \text{if } n = m \\ \texttt{flip (.) <\$> (flip f <\$> q) <*> rest} & \text{if } n > m \end{cases} \\
\texttt{Q} &= \texttt{flip f <\$> q} \\
\texttt{c} &= \texttt{aq}^{\texttt{k-1}}\ \texttt{<**> (flip (.) <*> Q <*> R)} \\
&= \texttt{pure (flip (\$)) <*> aq}^{\texttt{k-1}}\ \texttt{<*> (flip (.) <*> Q <*> R)} && (\text{I}) \\
&= \texttt{pure (.) <*> (pure (flip (\$)) <*> aq}^{\texttt{k-1}}\texttt{) <*>} && (\text{IV}) \\
&\quad \texttt{(pure (flip (.)) <*> Q) <*> R} \\
&= \texttt{pure (.).(flip (\$)) <*> aq}^{\texttt{k-1}}\ \texttt{<*>} && (\text{VII}) \\
&\quad \texttt{(pure (flip (.)) <*> Q) <*> R} \\
&= \texttt{pure (.) <*> (pure (.).(flip (\$)) <*> aq}^{\texttt{k-1}}\ \texttt{<*>} && (\text{IV}) \\
&\quad \texttt{pure (flip (.)) <*> Q <*> R} \\
&= \texttt{pure (.).((.).(flip (\$))) <*> aq}^{\texttt{k-1}}\ \texttt{<*>} && (\text{VII}) \\
&\quad \texttt{pure (flip (.)) <*> Q <*> R} \\
&= \texttt{pure}\ \overbrace{\texttt{(\textbackslash h -> h (flip (.))).((.).((.).(flip (\$))))}}^{\gamma}\ \texttt{<*>} && (\text{VIII}) \\
&\quad \texttt{aq}^{\texttt{k-1}}\ \texttt{<*> Q <*> R} \\
\texttt{d} &= \texttt{aq}^{\texttt{k}}\ \texttt{<**> R} && (1) \\
&= \texttt{pure (flip (\$)) <*> aq}^{\texttt{k}}\ \texttt{<*> R} && (\text{I}) \\
&= \texttt{pure (flip (\$)) <*> (aq}^{\texttt{k-1}}\ \texttt{<**> Q) <*> R} && (2) \\
&= \texttt{pure (flip (\$)) <*>} && (\text{I}) \\
&\quad \texttt{(pure (flip (\$)) <*> aq}^{\texttt{k-1}}\ \texttt{<*> Q) <*> R} \\
&= \texttt{pure (.)(flip (\$)) <*> (pure (flip (\$)) <*> aq}^{\texttt{k-1}}\texttt{) <*>} && (\text{IV}) \\
&\quad \texttt{Q <*> R} \\
&= \texttt{pure}\ \underbrace{\texttt{((.)(flip (\$))).(flip (\$))}}_{\delta}\ \texttt{<*> aq}^{\texttt{k-1}}\ \texttt{<*> Q <*> R} && (\text{VII})
\end{aligned}
$$

$$
\begin{aligned}
\gamma &= (\lambda h \to h\ (\texttt{flip}\ (\cdot))) \cdot ((\cdot) \cdot ((\cdot) \cdot (\lambda xf \to f\ x))) \\
&= \lambda x \to (\lambda h \to h\ (\texttt{flip}\ (\cdot)))(((\cdot) \cdot ((\cdot) \cdot (\lambda xf \to f\ x)))\ x) \\
&= \lambda x \to (\lambda h \to h\ (\texttt{flip}\ (\cdot)))((\cdot)(((\cdot) \cdot (\lambda xf \to f\ x))\ x)) \\
&= \lambda x \to (\lambda h \to h\ (\texttt{flip}\ (\cdot)))((\cdot)((\cdot)((\lambda xf \to f\ x)\ x))) \\
&= \lambda x \to (\lambda h \to h\ (\texttt{flip}\ (\cdot)))((\cdot)((\cdot)(\lambda f \to f\ x))) \\
&= \lambda x \to ((\cdot)(\lambda f \to f\ x)) \cdot (\texttt{flip}\ (\cdot)) \\
&= \lambda xg \to (\lambda f \to f\ x) \cdot ((\texttt{flip}\ (\cdot))\ g)
\end{aligned}
$$

$$
\begin{aligned}
&= \lambda xgf \to (\lambda f \to f\ x)((\texttt{flip}\ (\cdot))\ g\ f)\\
&= \lambda xgf \to (f \cdot g)\ x\\
&= \lambda xgf \to f\ (g\ x)\\
\delta &= ((\cdot)(\lambda xf \to f\ x)) \cdot (\lambda xf \to f\ x)\\
&= \lambda x \to (\lambda xf \to f\ x) \cdot ((\lambda xf \to f\ x)\ x)\\
&= \lambda xg \to (\lambda xf \to f\ x)(((\lambda xf \to f\ x)\ x)\ g)\\
&= \lambda xg \to (\lambda xf \to f\ x)(g\ x)\\
&= \lambda xg \to (\lambda f \to f\ (g\ x))\\
&= \lambda xgf \to f\ (g\ x)
\end{aligned}
$$

(1) Note that there is one less occurrence of `q`, as it moves to the 'atom' part of `postfix`.

(2) Using the definition of $\texttt{aq}^{\texttt{k}}$ from earlier.

As $\gamma = \delta$, it follows that `c` = `d`. This result shows that it is sound to substitute $\texttt{lift2(f}_\texttt{u}\texttt{, a, q)}$ with $\texttt{lift2(f}_\texttt{u}\texttt{, a}^\texttt{m}\texttt{, q)}$ on the input $\mathcal{S} = aq^n$, so long as $m \leq n$. By obtaining this auxiliary result, the problem is reduced to proving similarity on a single case.

Proving Similarity

Recall that the input sequence is $\mathcal{S} = aq^{n+1} = rq$. The goal in this final step is to show that `postfix a (flip f <$> q)` is similar to $\texttt{lift2(f}_\texttt{u}\texttt{, r, q)}$, where `r` is $\texttt{a}^\texttt{n}$. Once this result is verified, the inductive step is complete.

```
  postfix a (flip f <$> q)
= postfix r (flip f <$> q)                                    (1)
= r <**> rest
= r <**> (flip (.) <$> (flip f <$> q) <*> pure id)            (2)
= r <**> (pure (flip (.)) <*> (pure (flip f) <*> q) <*>)      (V)
          pure id)
= r <**> (pure (flip (.)).(flip f) <*> q <*> pure id)         (VII)
= r <**> (pure (\g -> g id).((flip (.)).(flip f)) <*> q       (VIII)
= r <**> (pure (flip f) <*> q)                                (3)
= pure (flip ($)) <*> r <*> (pure (flip f) <*> q)             (I)
= pure (.) <*> (pure (flip ($)) <*> r) <*>                    (IV)
  pure (flip f) <*> q
= pure (.).(flip ($)) <*> r <*> pure (flip f) <*> q           (VII)
= pure (\g -> (flip f)).((.).(flip ($))) <*> r <*> q          (VIII)
= pure f <*> r <*> q                                          (4)
= lift2(fu, r, q)                                             (XII)
```

(1) Using the result from before, as $\texttt{r} = \texttt{aq}^\texttt{n}$.

(2) Since the input sequence has more q, on the first use of `rest` the first branch is taken, but the second branch is taken on the second (recursive) use of `rest`.

(3) Obtained as follows;

$$(\lambda g \to g\ \texttt{id}) \cdot ((\texttt{flip}\ (\cdot)) \cdot (\texttt{flip}\ f))$$

$$
\begin{aligned}
&= \lambda x \to (\lambda g \to g\ \texttt{id})(((\texttt{flip}\ (\cdot)) \cdot (\texttt{flip}\ f))\ x) \\
&= \lambda x \to (\lambda g \to g\ \texttt{id})((\texttt{flip}\ (\cdot))((\texttt{flip}\ f)\ x)) \\
&= \lambda x \to (\texttt{flip}\ (\cdot))((\texttt{flip}\ f)\ x)\ \texttt{id} \\
&= \lambda x \to \texttt{id} \cdot ((\texttt{flip}\ f)\ x) \\
&= \lambda x \to (\texttt{flip}\ f)\ x
\end{aligned}
$$

(4) Obtained as follows;

$$
\begin{aligned}
&\quad (\lambda g \to g\ (\texttt{flip}\ f)) \cdot ((\cdot) \cdot (\lambda xf \to f\ x)) \\
&= \lambda x \to (\lambda g \to g\ (\texttt{flip}\ f))(((\cdot) \cdot (\lambda xf \to f\ x))\ x) \\
&= \lambda x \to (\lambda g \to g\ (\texttt{flip}\ f))((\cdot)((\lambda xf \to f\ x)\ x)) \\
&= \lambda x \to (\lambda g \to g\ (\texttt{flip}\ f))((\lambda f \to f\ x)\cdot) \\
&= \lambda x \to (\lambda f \to f\ x) \cdot (\texttt{flip}\ f) \\
&= \lambda xy \to (\lambda f \to f\ x)((\texttt{flip}\ f)\ y) \\
&= \lambda xy \to (\texttt{flip}\ f)\ y\ x \\
&= \lambda xy \to f\ x\ y
\end{aligned}
$$

As similarity is shown when $\mathcal{S} = a$ and assuming similarity when $\mathcal{S} = aq^k$ implies similarity for $\mathcal{S} = aq^{k+1}$, the property holds for all $\mathcal{S}$ following the production of p by induction. Since this rewrite maintains the desired semantics - the conversion of a set of disjunctions, some of which are directly left recursive, to `postfix` is sound.

6.3 Reductions

Reductions, introduced in section 5.2, form part of the rewriting process for left-recursive productions. This section proves the validity of these reductions, as well as describes the transformations, if any, that are required for the function component of `lift2`.

6.3.1 Reduction of <*

The goal is to show that $\texttt{a} = \texttt{lift2(f}_\texttt{u}\texttt{, p <* q, r)}$ and $\texttt{b} = \texttt{lift2(f}_\texttt{u}\texttt{, p, q *> r)}$ are equivalent, thus permitting the reduction.

$$
\begin{aligned}
\texttt{a} &= \texttt{pure f <*> (p <* q) <*> r} && \text{(XII)} \\
&= \texttt{(pure f <*> p) <* q <*> r} && \text{(VI)} \\
&= \texttt{pure const <*> (pure f <*> p) <*> q <*> r} && \text{(IX)} \\
&= \texttt{pure } \underbrace{\texttt{const.f}}_{\alpha} \texttt{ <*> p <*> q <*> r} && \text{(VII)} \\
\texttt{FC} &= \texttt{flip const} && (1) \\
\texttt{b} &= \texttt{pure f <*> p <*> (q *> r)} && \text{(XII)} \\
&= \texttt{pure f <*> p <*> (pure FC <*> q <*> r)} && \text{(X)} \\
&= \texttt{pure (.) <*> (pure f <*> p) <*> (pure FC <*> q) <*> r} && \text{(IV)} \\
&= \texttt{pure (.).f <*> p <*> (pure FC <*> q) <*> r} && \text{(VII)} \\
&= \texttt{pure (.) <*> (pure (.).f <*> p) <*> pure FC <*>} && \text{(IV)} \\
&\quad\ \texttt{q <*> r} \\
&= \texttt{pure (.).((.).f) <*> p <*> pure FC <*> q <*> r} && \text{(VII)}
\end{aligned}
$$

$$= \texttt{pure } \underbrace{\texttt{(\textbackslash g -> g FC).((.).((.).f))}}_{\beta}\ \texttt{<*> p <*> q <*> r} \qquad \text{(VIII)}$$

Since both `a` and `b` are now in the same ‘shape’, the remaining work is to prove equivalence of the two functions α and β.

$$\begin{aligned}
\texttt{flip const} &= (\lambda fpq \to f\ q\ p)(\lambda pq \to p) \\
&= \lambda pq \to (\lambda pq \to p)\ q\ p \\
&= \lambda pq \to q \\
\alpha &= (\lambda pq \to p) \cdot f \\
&= \lambda x \to (\lambda pq \to p)(f\ x) \\
&= \lambda x \to (\lambda q \to f\ x) \\
&= \lambda xy \to f\ x \\
&= \lambda xyz \to f\ x\ z \\
\beta &= (\lambda g \to g\ (\lambda pq \to q)) \cdot ((\cdot) \cdot ((\cdot) \cdot f)) \\
&= \lambda x \to (\lambda g \to g\ (\lambda pq \to q))(((\cdot) \cdot ((\cdot) \cdot f))\ x) \\
&= \lambda x \to (((\cdot) \cdot ((\cdot) \cdot f))\ x)(\lambda pq \to q) \\
&= \lambda x \to ((\cdot)((((\cdot) \cdot f))\ x))(\lambda pq \to q) \\
&= \lambda x \to ((\cdot)((\cdot)(f\ x)))(\lambda pq \to q) \\
&= \lambda x \to ((f\ x)\cdot) \cdot (\lambda pq \to q) \\
&= \lambda xy \to ((f\ x)\cdot)((\lambda pq \to q)\ y) \\
&= \lambda xy \to (f\ x) \cdot (\lambda q \to q) && (2) \\
&= \lambda xyz \to f\ x\ z
\end{aligned}$$

(1) for brevity

(2) $f \cdot id = f$

Since $\alpha = \beta$, it follows that $\texttt{a} = \texttt{b}$, hence the reduction is sound. By equivalence of `*>` and `~>`, the same argument can be made for the latter case.

6.3.2 Reduction of `<~>`

Intuitively, this case moves the ‘pair’ from one argument to the other. The goal is to show that reducing from $\texttt{a} = \texttt{lift2(f}_\texttt{u}\texttt{, p <~> q, r)}$ to $\texttt{b} = \texttt{lift2(g}_\texttt{u}\texttt{, p, q <~> r)}$ is sound.

$$\begin{aligned}
\texttt{a} &= \texttt{pure f <*> (p <~> q) <*> r} && \text{(XII)} \\
&= \texttt{pure f <*> (pure (,) <*> p <*> q) <*> r} && \text{(XI)} \\
&= \texttt{pure (.) <*> pure f <*> (pure (,) <*> p) <*> q <*> r} && \text{(IV)} \\
&= \texttt{pure (f.) <*> (pure (,) <*> p) <*> q <*> r} && \text{(III)} \\
&= \texttt{pure } \underbrace{\texttt{(f.).(,)}}_{\alpha}\ \texttt{<*> p <*> q <*> r} && \text{(VII)}
\end{aligned}$$

$$\begin{aligned}
\texttt{b} &= \texttt{pure g <*> p <*> (q <~> r)} && \text{(XII)} \\
&= \texttt{pure g <*> p <*> (pure (,) <*> q <*> r)} && \text{(XI)} \\
&= \texttt{pure (.) <*> (pure g <*> p) <*> (pure (,) <*> q) <*> r} && \text{(IV)} \\
&= \texttt{pure (.).g <*> p <*> (pure (,) <*> q) <*> r} && \text{(VII)} \\
&= \texttt{pure (.) <*> (pure (.).g <*> p) <*> pure (,) <*> q <*> r} && \text{(IV)} \\
&= \texttt{pure (.).((.).g) <*> p <*> pure (,) <*> q <*> r} && \text{(VII)}
\end{aligned}$$

$$= \texttt{pure}\ \underbrace{\texttt{(\textbackslash f -> f (,)).((.).((.).g))}}_{\beta}\ \texttt{<*> p <*> q <*> r} \qquad \text{(VIII)}$$

Similar to before, `a` and `b` are in the same form. However, the goal is not to show that the two functions α and β are exactly equivalent, but rather show that a trivial transformation of `f` can result in `g`. This can be done by introducing an intermediate function h, where $f\ (x, y)\ z = h\ x\ y\ z = g\ x\ (y, z)$.

$$\begin{aligned}
\alpha &= (f\cdot)\cdot(,) \\
&= \lambda x \to f \cdot (x,) \\
&= \lambda xy \to f\ (x, y) \\
&= \lambda xyz \to f\ (x, y)\ z \\
\beta &= (\lambda f \to f\ (,)) \cdot ((\cdot)\cdot((\cdot)\cdot g)) \\
&= \lambda x \to (\lambda f \to f\ (,))(((\cdot)\cdot((\cdot)\cdot g))\ x) \\
&= \lambda x \to (\lambda f \to f\ (,))(((\cdot)((\cdot)\cdot g)\ x)) \\
&= \lambda x \to (\lambda f \to f\ (,))((\cdot)((\cdot)\ (g\ x))) \\
&= \lambda x \to ((\cdot)\ (g\ x)) \cdot (,) \\
&= \lambda xy \to (g\ x) \cdot (y,) \\
&= \lambda xyz \to g\ x\ (y, z)
\end{aligned}$$

This result demonstrates that g can be constructed to wrap f by taking in two arguments, with the second being a tuple. The second argument can then be destructured, forming a tuple with the first argument and the first component of the tuple, and passing it into f.

6.3.3 Reduction of `<$>`

The goal is to show the soundness of a reduction from `a` = `lift2(f`$_\texttt{u}$`, g <$> p, q)` to `b` = `lift2(h`$_\texttt{u}$`, p, q)`.

$$\begin{aligned}
\texttt{a} &= \texttt{pure f <*> (g <\$> p) <*> q} && \text{(XII)} \\
&= \texttt{pure f <*> (pure g <*> q)} && \text{(V)} \\
&= \texttt{pure f.g <*> p <*> q} && \text{(VII)} \\
&= \texttt{lift2((f.g)}_\texttt{u}\texttt{, p, q)} && \text{(XII)} \\
&= \texttt{b} && \text{where } \texttt{h} = \texttt{f.g}
\end{aligned}$$

The behaviour of `h` is simply `f` with `g` applied to the first argument (when uncurried);

$$\begin{aligned}
f \cdot g &= \lambda x \to f\ (g\ x) \\
&= \lambda xy \to f\ (g\ x)\ y
\end{aligned}$$

This result also provides validity for reduction steps on the derived combinators, `<$`, `$>`, and `<&>`. The former two can be reduced by specifying `g` to be a constant function. Alongside the case for `<~>`, this also provides validity for reduction cases on `<*>` and `<**>`.

6.3.4 Reduction of `<*>` and `<**>`

Using the previous results for `<~>` and `<$>`, it is possible to construct reduction steps for `<*>` by equivalence (XIV). Additionally, using (I), it is also possible to create a similar rule for `<**>`.

Rather than showing a direct conversion between two `lift2`s, the goal is to verify that `a` = `p <**> q` is equivalent to `b` = `(\(p,q) -> q p) <$> (p <~> q)`. Verifying this result allows for bootstrapping off of the aforementioned results.

$$
\begin{aligned}
\texttt{a} &= \texttt{pure (flip (\$)) <*> p <*> q} && \text{(I)}\\
&= \texttt{pure (\textbackslash(p,q) -> p q) <*> (pure (flip (\$)) <\textasciitilde> p) <*> q} && \text{(XIV)}\\
&= \texttt{pure (\textbackslash(p,q) -> p q) <*>} && \text{(IX)}\\
&\quad \texttt{(pure (,) <*> pure (flip (\$)) <*> p) <*> q} &&\\
&= \texttt{pure (\textbackslash(p,q) -> p q) <*>} && \text{(III)}\\
&\quad \texttt{(pure (,)(flip (\$)) <*> p) <*> q} &&\\
&= \texttt{pure } \underbrace{\texttt{(\textbackslash(p,q) -> p q).((,)(flip (\$)))}}_{\alpha} \texttt{ <*> p <*> q} && \text{(VII)}\\
\texttt{b} &= \texttt{pure (\textbackslash(p,q) -> q p) <*> (p <\textasciitilde> q)} && \text{(V)}\\
&= \texttt{pure (\textbackslash(p,q) -> q p) <*> (pure (,) <*> p <*> q)} && \text{(IX)}\\
&= \texttt{pure (.)(\textbackslash(p,q) -> q p) <*> (pure (,) <*> p) <*> q} && \text{(III, IV)}\\
&= \texttt{pure } \underbrace{\texttt{((.)(\textbackslash(p,q) -> q p)).(,)}}_{\beta} \texttt{ <*> p <*> q} && \text{(VII)}
\end{aligned}
$$

$$
\begin{aligned}
\alpha &= (\lambda(p,q) \to p\ q) \cdot ((,)(\lambda xy \to y\ x))\\
&= \lambda x \to (\lambda(p,q) \to p\ q)((\lambda xy \to y\ x), x)\\
&= \lambda x \to (\lambda xy \to y\ x)\ x\\
&= \lambda x \to (\lambda y \to y\ x)\\
&= \lambda xy \to y\ x\\
\beta &= ((\cdot)(\lambda(p,q) \to q\ p)) \cdot (,)\\
&= \lambda x \to (\lambda(p,q) \to q\ p) \cdot (x,)\\
&= \lambda xy \to (\lambda(p,q) \to q\ p)(x,y)\\
&= \lambda xy \to y\ x
\end{aligned}
$$

As before, since $\alpha = \beta$, it follows that `a` = `b`, hence the two forms are equivalent. However, since it is now in a form `f <$> (p <~> q)`, it can be reduced using existing reduction rules. Additionally, this also permits for further bootstrapping for reducing the `<:>` case.

6.4 Normalisation

Similar to the reduction rules, the normalisation rules are used within the left recursion rewriting process. Validity is shown for the normalisation steps, as well as for any function transformations that occur.

6.4.1 Normalisation of `p <**> (f <$ q) <*> r`

Note that unlike the other cases, this case contains an additional discarded parser (`q`). The goal is to show that the normalisation of `a` = `p <**> (f <$ q) <*> r` to `b` = `lift2(`$\texttt{f}_\texttt{u}$`, p, q *> r)` is sound.

$$
\begin{aligned}
\texttt{a} &= \texttt{pure (flip (\$)) <*> p <*> (f <\$ q) <*> r} && \text{(I)}\\
&= \texttt{pure (flip (\$)) <*> p <*> (pure (const f) <*> q) <*> r} && \text{(XIII)}\\
&= \texttt{pure (.) <*> (pure (flip (\$)) <*> p) <*>} && \text{(IV)}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{pure (const f) <*> q <*> r} \\
&= \texttt{pure (.).(flip (\$)) <*> p <*> pure (const f) <*> q <*> r} && \text{(VII)} \\
&= \texttt{pure } \overbrace{\texttt{(\textbackslash g -> g (const f)).((.).(flip (\$)))}}^{\alpha} \texttt{ <*> p <*>} && \text{(VIII)} \\
&\quad \texttt{q <*> r} \\
\texttt{b} &= \texttt{pure } \underbrace{\texttt{(\textbackslash g -> g FC).((.).((.).f))}}_{\beta} \texttt{ <*> p <*> q <*> r} && (*) \\
\alpha &= (\lambda g \to g\ ((\lambda pq \to p)\ f)) \cdot ((\cdot) \cdot (\lambda xh \to h\ x)) \\
&= (\lambda g \to g\ (\lambda q \to f)) \cdot ((\cdot) \cdot (\lambda xh \to h\ x)) \\
&= \lambda x \to (\lambda g \to g\ (\lambda q \to f))(((\cdot) \cdot (\lambda xh \to h\ x))\ x) \\
&= \lambda x \to (\lambda g \to g\ (\lambda q \to f))((\cdot)(\lambda h \to h\ x)) \\
&= \lambda x \to (\lambda h \to h\ x) \cdot (\lambda q \to f) \\
&= \lambda xy \to (\lambda h \to h\ x)\ f \\
&= \lambda xy \to f\ x \\
&= \lambda xyz \to f\ x\ z \\
\beta &= \lambda xyz \to f\ x\ z && (*)
\end{aligned}
$$

Note that the results for `b` and β are reused from subsection 6.3.1 for brevity. Using these results, $\alpha = \beta$, therefore `a` = `b` - hence the normalisation is sound.

6.4.2 Normalisation of `f <$> (p <~> q)`

The goal is to show that normalising `a` = `f <$> (p <~> q)` to `b` = `lift2(`$\texttt{g}_\texttt{u}$`, p, q)` is sound.

$$
\begin{aligned}
\texttt{a} &= \texttt{pure f <*> (p <\textasciitilde> q)} && \text{(V)} \\
&= \texttt{pure f <*> (pure (,) <*> p <*> q)} && \text{(XI)} \\
&= \texttt{pure (.) <*> pure f <*> (pure (,) <*> p) <*> q} && \text{(IV)} \\
&= \texttt{pure f. <*> (pure (,) <*> p) <*> q} && \text{(III)} \\
&= \texttt{pure } \underbrace{\texttt{(f.).(,)}}_{\alpha} \texttt{ <*> p <*> q} && \text{(VII)} \\
\alpha &= (f\cdot) \cdot (,) \\
&= \lambda x \to (f\cdot)(x,) \\
&= \lambda xy \to f\ (x, y)
\end{aligned}
$$

The function `g` can trivially be constructed as one that takes in two arguments and combines them into a tuple, which is then passed into `f`, as shown in α.

6.4.3 Normalisation of `p <**> pure f <*> q`

The goal is to show that a normalisation step from `a` = `p <**> pure f <*> q` to `b` = `lift2(`$\texttt{f}_\texttt{u}$`, p, q)` is sound.

$$
\begin{aligned}
\texttt{a} &= \texttt{pure (flip (\$)) <*> p <*> pure f <*> q} && \text{(I)} \\
&= \texttt{pure } \underbrace{\texttt{(\textbackslash g -> g f).(flip (\$))}}_{\alpha} \texttt{ <*> p <*> q} && \text{(VIII)} \\
\alpha &= (\lambda g \to g\ f) \cdot (\lambda xf \to f\ x) \\
&= \lambda x \to (\lambda g \to g\ f)(\lambda f \to f\ x)
\end{aligned}
$$

$$= \lambda x \rightarrow (\lambda f \rightarrow f\ x)\ f$$
$$= \lambda x \rightarrow f\ x$$

As α can be shown to be equivalent to `f`, it follows that `a` = `b`, hence the normalisation step is valid.

Summary

This chapter verifies the operations performed to support grammar refactoring, ensuring that the desired semantics are maintained throughout an arbitrary number of transformations. By validating the transformations via equivalences and structural induction over a single grammar rule, left recursion analysis and the subsequent refactoring are shown to be sound.

Chapter 7

Evaluation

While chapter 6 verified the theoretical soundness of the transformations performed, it is equally important to consider the efficacy of the optimisations discussed in chapter 5 when applied to real examples. This chapter begins by exploring how these optimisations perform when applied to actual parsers, some approximations for optimisation complexity, and finishes by analysing the performance of the library introduced in chapter 3 against existing solutions.

For the purpose of consistency, all performance tests are run on a machine equipped with an i7-10750H[1] and 16GB of DDR4 at 3200 MT/s. TypeScript-based experiments are all run on version 10.8.0 of *ts-node*[2].

7.1 Efficacy of Optimisation and Analysis

7.1.1 Left Recursion Results

While it is difficult to quantify the effectiveness of rewriting a left-recursive production, it is beneficial to see how real examples of parsers may be transformed. Note that prior to analysis (if the operators were converted into functions, but no further processing was done), a left-recursive production will still infinitely run until it exceeds the capacity of the call stack. The same behaviour can be seen in other parser combinator libraries, such as *ts-parsec* and *parsimmon* (TypeScript / JavaScript based libraries, which are compared further in section 7.3).

Consider a simple calculator, which allows for left-associative addition and subtraction of numbers. The raw (TypeScript*) code is shown in listing 7.1, where left recursion is present in two of the three disjunctions for `calc` and the non-left-recursive disjunction is `nat`. It is also important to note that the left-recursive disjunctions (for addition and subtraction) are in different forms. Note that numerous other forms can be processed due to the normalisation and reduction steps highlighted in subsection 5.2.3

```
val nat   = ((s: string) => parseInt(s)) <$> /[0-9]+/;
lazy calc = (x => y => x + y) <$> (calc <* '+') <*> nat <|>
            calc <**> ((x => y => x - y) <$ '-') <*> nat <|>
            nat;
```

Listing 7.1: Simple addition and subtraction calculator, prior to processing

[1] https://ark.intel.com/content/www/us/en/ark/products/201837/intel-core-i710750h-processor-12m-cache-up-to-5-00-ghz.html

[2] https://www.npmjs.com/package/ts-node

This code shown in listing 7.2 is generated by the preprocessor from the previous example of the left-recursive calculator. The preprocessor correctly identifies the disjunction without left recursion and uses it as the first parameter for `postfix`. On the other hand, the two left-recursive disjunctions were identified, normalised, and reduced, thus leading to the two existing in the same 'shape' (as seen in lines 4 and 5).

Prior to the refactoring, attempting to parse `'1-2-3+4'` would recurse until the call stack was exhausted (as expected). However, with the rewritten parser, the computation completes with the expected outcome: giving `0` as the result.

```
1  const nat  = fmap((s: string) => parseInt(s), re(/[0-9]+/));
2  const calc = lazy(() => postfix(nat,
3      choice(
4          pamf(apR(chr('-'), nat), (y) => (x) => x - y),
5          pamf(apR(chr('+'), nat), (y) => (x) => x + y)
6      )
7  ));
```

Listing 7.2: Simple addition and subtraction calculator, after processing

However, the example above only demonstrated cases with direct left recursion (which is significantly more common in actual use [4]). Despite the rarity of the other two cases, discussed in subsection 2.3.1, it is still important to be able to detect them, even if they cannot be easily fixed. This can be seen in Figure 7.1, where `p` and `q` are indirectly (and mutually) left-recursive, and `r` contains hidden left recursion. The case of mutual recursion is not handled by *ANTLR4* (detailed in subsection 8.3.1), where the grammar will not be accepted, whereas the preprocessor will still perform the refactoring but provides a warning to the user.

```
P ::= Q x
Q ::= P
R ::= ε R
```

$\rightsquigarrow$

```
lazy p = q *> x;
lazy q = p;
lazy r = pure("") <* r;
```

Figure 7.1: Left: grammar with indirect and hidden left recursion, right: syntactically equivalent parser

The preprocessor is able to detect both of these cases, noting that `p` and `q` have disjunctions with 'non-local' left recursion. On the other hand, `r` is treated as if it had local left recursion, based on the definitions provided in subsection 5.2.1, thus being trivially detected. This demonstrates the preprocessor's ability to refactor grammars that carry semantic information regarding the parse structure, as well as handle cases that cannot be resolved by some parser generators (especially when the `inline` declaration is used). However, it is still important to acknowledge that this technique cannot cover every case: soundness is verified (chapter 6), not completeness.

7.1.2 String Trie Results

In order to quantify the performance improvements granted by the optimisation, a parser is generated that accepts n words, from a shuffled set of the top 1,000 most English common words [32]. The aim is to demonstrate the performance penalties caused by backtracking; the original parser is simply a number of alternative strings, each wrapped in `attempt`. Ordering was performed to prevent the parser from consuming just a prefix (for example, successfully parsing `in` when the input is `include`). On the other hand, the optimised parser is generated from a sequence of strings with the optimisation enabled, thus creating a string parser that requires no backtracking. A single run for n words consists of constructing a parser that accepts all n words and then parsing each of the words. For each n, 105 runs are completed on each parser with the results of the initial 5 runs being discarded to account for caching or warming up.

The benefits of the optimisation can be seen most clearly on 'larger' parsers (ones with more alternatives). While an increase is expected in the total execution time of these tests (simply due to more words being tested), a more interesting result is visible in Figure 7.2, where the changes to average (per word) execution time can be seen. For the unoptimised parser, the trend is roughly linear with respect to the number of alternatives. On the other hand, the time to parse each word in the optimised parser is roughly constant.

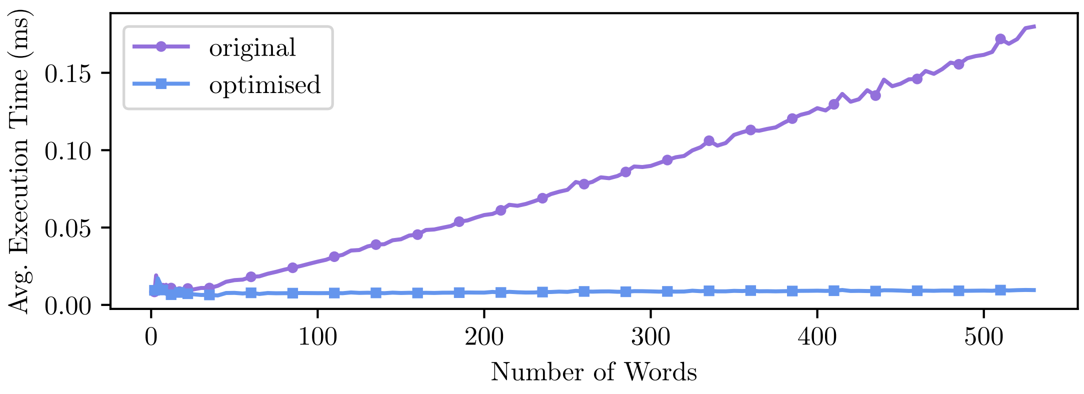


Figure 7.2: Average execution time of parsing a single word on each parser

The same result can be seen with the parser's performance on an invalid word (one that is not in the alternatives), as shown in Figure 7.3. As expected, the time to fail is longer than the time to successfully parse a valid word as it must reach the end of all the alternatives. However, with the optimised parser, it can 'short-circuit' and fail quite quickly as soon as any parse fails, whereas the unoptimised parser requires going through each `attempt` disjunction.

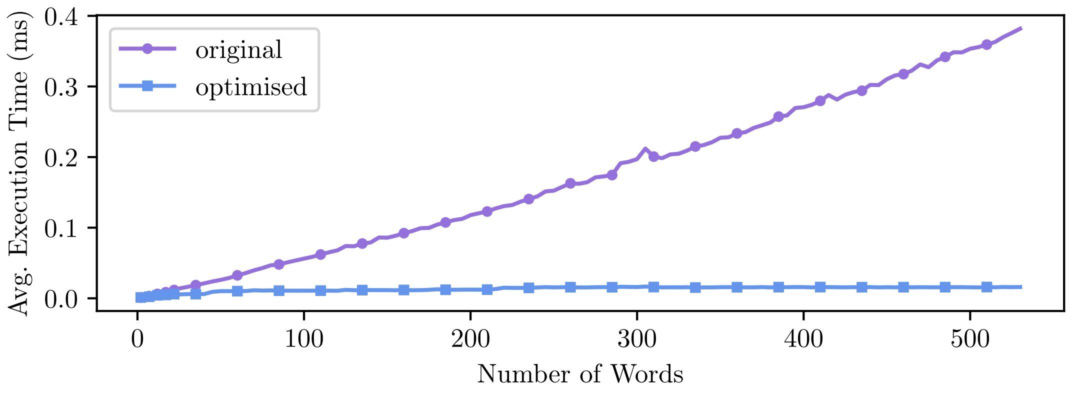


Figure 7.3: Average execution time of parsing a single invalid word on each parser

7.1.3 Backtracking Reduction Results

A similar result to subsection 7.1.2 can be observed for generalised backtracking. In order to test a pathological case of how backtracking reduction can be hugely beneficial, parsers were generated to accept n periods (`'.'`s) followed by a single character from `'a'` to `'j'` -

a total of 10 parser disjunctions are generated for each n (shown as parser `p` in listing 7.3). A single run executes each of the 10 strings against the generated parser. Similar to the previous experiment, a total of 105 runs are performed, with the first 5 being discarded. For example, when $n = 3$, the following parser is created:

```
val p = attempt('.' *> '.' *> '.' *> 'a') <|> ... <|>
        attempt('.' *> '.' *> '.' *> 'j');
val q = '.' *> '.' *> '.' *> ('a' <|> ... <|> 'j');
```

Listing 7.3: Pathological of a parser that requires backtracking for $n = 3$ (`p`) and the 'optimised' result (`q`)

It is clear that the parser `p` described in listing 7.3 requires backtracking: all disjunctions have a (large) shared prefix. The only differentiating factor between the disjunctions is the final character, however, work is repeatedly being done to parse the first n periods, then backtracking when the final character does not match. As there is a common prefix, it is acceptable to factor it out (due to the explicit use of backtracking through `attempt`s) into the parser `q`.

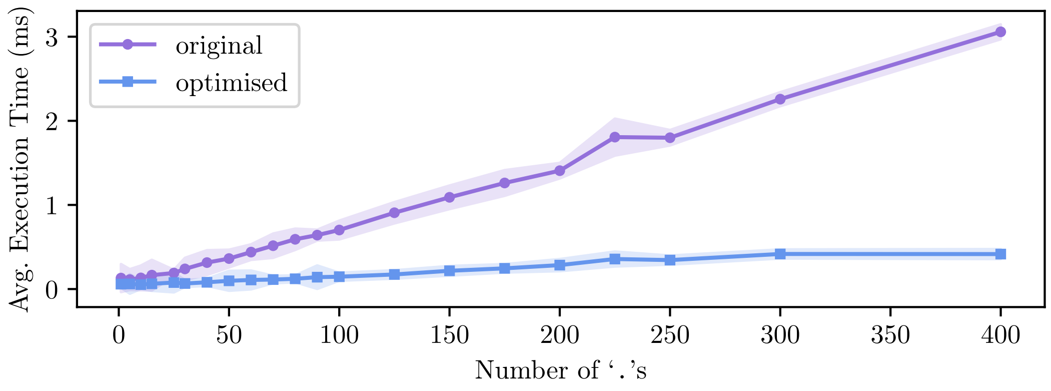


Figure 7.4: Average execution time of parsing all 10 strings

As expected, a greater benefit is noted when more backtracking is performed, as seen in Figure 7.4. While there is an increase in the parsing time for the optimised parser, it is expected as the strings are longer (therefore, more parsers need to be applied). However, with a larger string, significantly more work needs to be redone when backtracking. While this result is shown on an unrealistic parser, it is completely feasible that a similar pattern can be seen in real-world applications (albeit to a significantly lesser extent).

7.2 Preprocessor

It is inherently difficult to accurately reason about the performance of the preprocessor. First and foremost, the amount of time spent in each pipeline stage is extremely dependent on the parsers defined - as such, it is challenging to generate example programs that can stress all aspects of the optimiser.

However, it may still prove valuable to reason about the complexity of certain aspects of the optimiser. It is also important to note that certain steps, such as computing the first sets (both global and local), are subject to memoisation in an effort to avoid repeated computation at the cost of space. Therefore, it is reasonable to approximate it as a constant time operation once it is computed.

For a single production (lexical declaration) consisting of m disjunctions, each with a 'size' (number of combinators) of n, the time spent in the left recursion analysis stage is approximately $O(mn)$. Considering a single disjunction (therefore, this has to be done m times), the worst-case scenario is when the parser is detected to be left-recursive ($\approx$ constant time, as mentioned before), and the entire combinator falls into the first parser argument after normalisation; thus it must be entirely reduced. This will take n steps (almost exactly that, other than the cases which bootstrap off existing cases), each of which are constant time, and a final rewrite step, which is also constant time. Similarly, the worst case (completely unbalanced tree) for a single fusion step is also $O(mn)$ - without even accounting for the recursive fusion steps.

However, it is important to remember that the preprocessor essentially acts as an optimising compiler to some extent, the complexity of which is non-polynomial [33].

7.3 Comparison to Other Libraries

While it is important to consider the efficacy of the optimisations and the preprocessor itself, the performance and features of the underlying parsing library should also be analysed. Instead of comparing against parser generators and handwritten solutions as done by *Chevrotain* [34], the performance is only compared against *ts-parsec* [35] and *parsimmon* [36], both of which are parser combinator libraries.

7.3.1 Feature Parity

The idea of feature parity, in terms of what is provided by the library, should be taken with a grain of salt as one of the major draws of parser combinators is the ability for the user to define derived combinators. However, it can still be beneficial to consider the differences in what is provided directly by the library.

All three libraries support the use of regular expressions for performing parsing, which can often perform better than concatenating a collection of individual string parsers. However, *Teaspoon* and *parsimmon* provide this functionality as a parsing primitive, whereas *ts-parsec* requires this for the lexing stage - this step is not required by the former two and can often lead to degraded performance (observed in subsection 7.3.2).

Both *parsimmon* and *ts-parsec* perform backtracking by default, whereas *Teaspoon* maintains the semantics of the *Parsec* family. However, *Teaspoon* can achieve the same semantics by wrapping alternatives in `attempt`, whereas neither of the other libraries can obtain *Parsec* semantics as neither provide operators to prevent backtracking. This functionality can be desirable to allow for code portability, as *Attoparsec* (Haskell) provides these semantics if required, despite always backtracking by default. Additionally, the lack of `cut` (to stop backtracking) prevents these libraries from being optimised to avoid unnecessary backtracking (thus being unsuitable for practical use).

However, *parsimmon* and *ts-parsec* both provide a number of niceties, such as commonly used primitive parsers such as `digit`, `whitespace`, and similar, alongside predefined functions for repetition or optional parsers. On the other hand, these are quite trivial to implement with the primitives and combinators provided in *Teaspoon*.

7.3.2 Performance

In order to create as even a comparison as possible, the goal was to create a parser for JSON, with the structure of all the parsers being as close as possible in order to eliminate variances from certain library features. JSON objects of various lengths are then generated in order to test scaling. For each generated object, 55 runs are performed, with the first 5 being discarded. Within each run, the object is parsed 5 times.

A JSON parser was chosen for two main reasons, the first of which is the lack of left recursion and the simplicity of the grammar (alongside its ability to use various combinators). While the ability to address left-recursive productions was a key feature of the preprocessor, it required features from the library that may be too specific. Despite all three libraries being able to handle left recursion in one way or another, the implementations could have become quite different, thus leading to differences in the testing methodology. This was unacceptable, as the primary purpose of this experiment was to determine the relative performance of combinators. The second reason for using JSON was the ability to generate large amounts of realistic (thus demonstrating real-world application) data with tools such as *json-generator*[3].

Note that the end-to-end flow (which would reconstruct a JSON object) is not measured. This decision was made to prevent adding a fairly fixed time operation, which could result in the relative performance between the libraries being skewed.

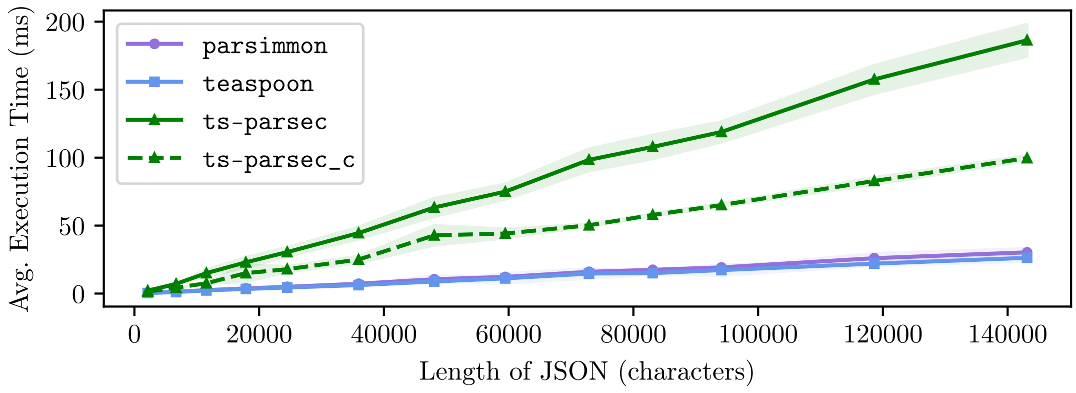


Figure 7.5: Average execution time of parsing JSON files of various sizes on TypeScript combinator libraries

While *Teaspoon* and *parsimmon* have extremely similar performance, *ts-parsec* falls behind by a factor of $\approx 7\times$ to complete a parse. This is most likely due to the lexing stage, which utilises regular expressions to obtain tokens. As seen in Figure 7.5, when the lexing stage is pre-computed (`ts-parsec_c`) the performance is notably better, however it still performs measurably worse than the others. An interesting observation is the significant decrease in variance when the lexing stage is excluded. Furthermore, the variance seems to be notably higher from using regular expressions.

On the other hand, *Teaspoon* and *parsimmon* are almost close enough to be considered to be within a margin of error of each other, however, the former performs the parse slightly faster, by a very narrow margin. Comparing the two libraries, both have fairly similar underlying implementations for the combinators, with the primary difference being the former's use of parsers as functions and the latter's use of parsers as objects. However, there are fundamental differences in how backtracking is performed: the former requires the use of `attempt` to perform it explicitly, whereas the latter performs backtracking by default. In the case of JSON, where there is very little ambiguity, the impact is negligible.

However, it is important to acknowledge that the testing methodology has slight flaws. For example, this only tests performance on the overall grammar; it does not account for the performance of each combinator. This could skew the results if a computationally expensive combinator is not utilised much in a given grammar. Another source of unfairness can stem from a lack of feature parity, such as *ts-parsec*'s use of an expensive lexing stage or how the state is tracked differently.

[3] https://json-generator.com/

Chapter 8

Conclusion

8.1 Contributions

This project achieves all of the aims set out in chapter 1, and more. The first contribution of the project is a lightweight parser combinator library built entirely in TypeScript (*Teaspoon*) with no external dependencies. It adheres closely to the semantics of the *Parsec* family of parsers. The library itself performs as well as, or better than, existing parser combinator libraries as demonstrated in section 7.3.

While the library itself can function alone, it is best coupled with the preprocessor where the bulk of the contributions lie. The preprocessor alone provides a highly extensible pipeline for performing analysis over the structure of an arbitrary TypeScript program. It also contributes numerous language features which can be used outside of writing parser combinators, such as the ability for the user to define infix operations, Rust-like macros, and lazy evaluation.

The primary contributions lie in the optimisation and analysis performed in the pipeline: namely, the ability to analyse left-recursive productions, as well as an optimisation to reduce unnecessary backtracking. This project provides an algorithm and a set of equivalences to automate the mechanical process of normalising and reducing common left-recursive productions into a form that can be trivially rewritten into a homogenous chain, thus removing left recursion, as described throughout section 5.2. The soundness of this method (alongside the equivalences) is verified and proven in chapter 6. Furthermore, a definition of first sets and left-recursion is described on parser combinators, building on the existing intuition for grammars. An algorithm for recursively fusing disjunctions utilising Merkle trees is also provided, thus reducing the amount of backtracking that needs to be performed and improving the performance of the generated parser. Evidence of the efficacy of these optimisations is shown in section 7.1.

8.2 Ethical Issues

Due to the nature of this project, there are no reasonable, direct ethical concerns. However, as with all contributions, dual use may still apply. The same argument would be made for any parser, compiler, or programming tool in general - it is not out of the question that the utility could be used for malicious purposes. Similarly, it is possible that the tool could be used to produce code that may need to follow export laws. One could argue that with both of these scenarios, the nature of what is produced is unchanged from what is fed into it: if malicious code is processed, then malicious code is generated.

A possible concern may lie in the code generation, as the semantics of parsers can be changed after some optimisations. As such, this could lead to unintended side effects. This is also important from the use of the preprocessor as part of an automated pipeline - if a malicious actor were to modify how the code is generated in the preprocessor, the input program could be tampered with to cause unintended, harmful side effects without the user being aware.

This project utilises two libraries: *Parsley* [1] for parsing and *scopt* [21] for command-line options. As such, this project needs to adhere to the licences these libraries fall under, BSD 3-clause and MIT, respectively. In this situation, both licences allow for private and commercial use, as well as distribution and modification. It comes with no warranty nor liability, and the name cannot be used without consent. Additionally, the same disclaimer must be provided.

8.3 Related Works

8.3.1 ANTLR4

One change between *ANTLR3* and *ANTLR4* was the type of parser used. Previously, an LL($*$) parser was used in *ANTLR3* and therefore suffered the same limitation with left-recursive rules. *ANTLRWorks* [37], a development environment for *ANTLR3* grammars, supports direct left-recursion removal [38]. However, this only existed as a tool outside of the parser generator and therefore restricted the editing environment, which may not be ideal for all users. Additionally, this approach only supported rewriting direct left-recursion, where the parser is defined with itself as the first parser in a sequence. For example, the grammar $A \to A\alpha$ (as mentioned previously) would be rewritten, but an indirect left-recursion would not be.

Included with *ANTLR4*'s ALL($*$) (adaptive LL($*$)) parsing, the preprocessor rewrites the grammar for **direct** left-recursion, before the parser generation stage begins [4]. However, the rewriting is not performed for indirect or hidden left-recursion cases; the generated trees would be far larger, and these cases are not commonly found in real grammars.

Part of the justification for this is a larger grammar (after the transformation) as well as a more complex parse tree. However, the latter may not be as much of a concern in the case of combinator parsing, as it could be possible to automatically transform and combine the parse tree. On the other hand, the former may be taken as an argument against rewriting the grammar to some extent; one of the benefits of using combinator parsing is the inherent readability which may be lost if too much processing is performed.

The idea of grammar refactoring is used heavily throughout section 5.2. However, parser generators do not carry the user-defined semantics directly in the grammar (this is provided externally, over the parse tree), whereas parser combinators do. As such, additional work is required on top of a grammar rewrite to ensure semantics are retained.

8.3.2 Counting Left Recursion

Frost & Hafiz (2006) [39] explore the use of memoisation to reduce the complexity of LL parsing from exponential to polynomial, following the work of Norvig (1991) [40]. However, this paper also noted that it would be possible to support left-recursive grammars by counting the number of applications of a parser to a given index during the memoisation process. The paper notes that this count is either zero or one for productions that have no left-recursion. On the other hand, the maximum number of applications of a parser to an index is one more than the size of the remaining input - at which point the parse will fail. It is important to note that the naïve approach of counting the number of parsers applied to an index is insufficient since some parsers may directly refer to another parser without additional requirements.

The work is continued in Frost et al. (2008) [41], where the aforementioned method of counting left-recursion is used to accommodate direct left-recursion (detailed in subsection 2.3.1). The ability to handle indirect left-recursion is extended in this paper by adding additional data when propagating results back up the memoisation, including the context of the left-recursion. This would prevent an empty (note, failed) result from being stored for a parser that is able to continue a parse at a given index in a different context. Instead, the context of the left-recursion is compared when checking the memoisation table (when the parsing is stopped by exceeding an application count).

As the second approach by Frost builds on the first, the discussion will only revolve around the latter. This solution has a number of advantages, namely the lack of a preprocessing step. By avoiding a preprocessing step, with everything being in the host language, changes are easier to make, and the code is likely to be more intuitive to read. Additionally, the grammar is not altered or modified in any way, which also aids in readability. However, this approach requires additional bookkeeping and causes a state to be maintained for the parsers, which can introduce additional costs in terms of space, especially with storing the left-recursion context. Additionally, this method will still continue to execute the parsers, which may be computationally expensive in the case of a large parse input.

Note that with this project, a preprocessor is necessary to introduce user-defined operators in TypeScript regardless. As such, this method was not as suitable since it would not be able to leverage the global view that the preprocessor has. Additionally, it adds significant complexity to the parsers themselves, which could prevent some users from implementing the parsers they may require.

8.3.3 Generalising Monads to Arrows

Swierstra & Duponcheel (1996) [42] introduce the idea of having two components in a parser; a fast static component which recognises whether the parser can succeed alongside a slower dynamic component which performs the actual parse. The static parser contains both the list of symbols (typically `Char`s), which can begin the derivation, as well as whether the parser can accept ε. The use of a static and dynamic component, however, does not fit into `Monad` interface.

An `Arrow`, described by Hughes (2000) [43], allows for both static effects (much like `Applicatives`) as well as dynamic effects (similar to `Monads`) in the same context [44]. As such, they allow for more granular control as to how effects can be combined. For example, `first` takes in a function and a pair of inputs but only applies a function to the first, and vice versa for `second`. This allows for a chain of composed arrows to keep effects separate as well as perform multiple computations in the same step. For Swierstra & Duponcheel's parsers, `Arrows` allow for both the static and dynamic components of two parsers to be manipulated or combined as required. The resultant parser can then still be executed in the same manner.

Similar to the approach detailed in subsection 8.3.2, this requires additional bookkeeping and does not utilise the global scope provided by the preprocessor. The additional bookkeeping heavily complicates the interface and may prevent users from adapting the combinators for their own use, however, the idea of maintaining a static component may prove interesting for future performance optimisations on the library.

8.3.4 Self Inspecting Code

Baars & Swierstra (2004) [45] present a deep embedding for grammars. The deep embedding preserves the structure, thus allowing for analysis over the structure of the grammar. This technique is also present in both *Parsley* (Scala) [1] and *Parsley* (Haskell) [46], where inspection is performed over the grammar. Parsers are initially stored in an environment where parsers are represented as grammar productions. Once a compilation step is done on the environment, the environment is converted into one containing parsers. However, since the environment provides a global view of productions, left recursion can be removed.

The approach taken by Baars & Swierstra is similar to the one described in section 5.2 where a partition is made on a rule's disjunctions. Similarly, the originally (infinitely) recursive production is rewritten into an iterative approach (albeit using a left fold rather than `postfix`). However, this approach only applies to a limited number of combinators, whereas the previously described implementation supports a wider range of commonly used combinators and 'shapes' due to the normalisation and reduction steps.

8.3.5 Existing TypeScript Libraries

ts-parsec is an open source library, maintained by *Microsoft* for performing combinator parsing in TypeScript [35]. The library provides the ability to tokenise based on regular expressions, combinators, and support for recursive syntax, either with laziness or with `chains`. However, it lacks the ability to perform context-sensitive application or context-sensitive tokenisation. In its documentation, conflicting token definitions are resolved by first taking the longest token, and any further conflicts are resolved by the order they are passed into the lexer. However, this lexing stage greedily generates tokens and loses information regarding the context of the token. Ambiguities will then have to be resolved by the parser [6]. However, it is possible for a manual tokeniser to be implemented to support this. Unlike *ts-parsec*, *Teaspoon* does not require such a tokeniser.

Another similar library is *parsimmon*, which is written for JavaScript [36]. This follows the functional nature of combinator parsing closer than *ts-parsec* with compatibility with *fantasy-land*[1], an unofficial specification for algebraic JavaScript.

Neither of these libraries directly support left-recursive grammars (as no analysis is performed at runtime), and neither have classical operators. The former uses function calls that take in parameters, whereas the latter involves parsers that provide methods to combine with other parsers. While the implementation for the parsing library is primarily derived from *Parsec* (see subsection 8.3.6), the use of standard regular expressions as parsers is quite interesting and is something that is implemented as an additional 'primitive'.

Comparisons between *Teaspoon* and the aforementioned libraries are explored in further detail in section 7.3.

8.3.6 Parsec

Throughout the implementation, the work of Leijen & Meijer (2001) [13] is used heavily. *Parsec* aims to be a combinator library for use in the real world. It does so by efficiently generating error messages, including the position and possible productions that may have been expected at the illegal position (the first set). Note that intuitively, the first set for a rule is anything that can begin the derivation for that particular rule. This set can be recursively constructed; the first set of a terminal is itself, and if the production's rule starts with a non-terminal, the first set of the first production includes the first set of the left-most rule in the production.

Parsec has arbitrary lookahead, with support for backtracking via the use of `try`. This is done by using a consumer-based approach, where the result of the parser is either `Consumed` or `Empty`, depending on whether input has been consumed or not, respectively. Within the response, the result is also captured as `Ok` or `Error`, which captures whether the parse was successful or not.

Note that the use of `try` allows the parser to become effectively LL(∞) with full backtracking. This occurs when every parser is wrapped with `try`. However, this can come at the

[1]https://github.com/fantasyland/fantasy-land

penalty of an exponential worst-case complexity. Even if the grammar is not ambiguous, backtracking may still be required to look further into the future.

8.3.7 Parsley

Parsley for Scala [1] builds on a previous port of *Parsec* and improves on the latter's performance pitfalls. The latter's pitfalls can largely be attributed to inherent differences between Haskell and the JVM. The performance is achieved with the use of a stack machine, which fully supports applicative parsers. Note that `bind` has issues with performance and optimisation as the parser is unknown until the result of the first parser has been computed.

While the aim of this project was not to reach the best performance in TypeScript, these design considerations may be interesting to consider for future performance improvements. Furthermore, the idea of lawful optimisation on parsers is also utilised, albeit in the preprocessing pipeline (described in subsection 5.1.2). *Parsley* itself is also utilised as the parser for the preprocessor, as mentioned in section 4.2.

8.3.8 Khepri

One of the key features that this project aims to implement is the ability to use combinators in a natural way, as operators in a DSL, rather than nested function calls or method calls. However, TypeScript does not support natively support this. A modified version of ECMAScript, *Khepri* [47], supports this in a similar way to how this project implements user-defined operators. In general, *Khepri* aimed to be more concise than ECMAScript and have better support for functional programming. However, the project has since been abandoned, with the domain expiring and lacking updates as of 2015.

The author of the project has detailed the preprocessor in a 2014 blog post [48]. However, it also supports arbitrary (within limits) operators - this is something that is not needed in this project as it intends to specifically parse for combinators in order to build up a 'combinator tree', which can later be optimised over and modified. Additionally, adding operators dynamically can cause difficulties as expressions may need to be parsed differently depending on the local context of the operator. With static (fixed) operators, the rules are written into the grammar as if they were any other operator, similar to mathematical operators. Furthermore, the parsing is done in JavaScript, whereas the preprocessor for this project is implemented in Scala in order to leverage the pattern matching abilities that come from it for the optimisation stage, as detailed in section 4.2. However, the idea of arbitrary operators is interesting and is implemented as a minor feature where a user-defined function can be wrapped in angle brackets to be processed as an infix operator, as detailed in section 4.4.

8.4 Future Work

While the preprocessor and library (in their current state) achieve the desired, successful results, there still remain numerous avenues in which the project can be extended further.

Optimising the Library As library performance was not the primary goal of this project, it stands to reason that a future improvement is to optimise the library. This would involve changing the underlying architecture, as described by Willis & Wu (2018) [1], rather than closely adhering to *Parsec*'s implementation. The majority of parser combinator libraries currently available for TypeScript or JavaScript use a similar underlying implementation rather than anything similar to the stack machine used in *Parsley*.

More Cases for Analysis An obvious next step would be to attempt to consider even more cases for analysis, particularly for the left recursion optimisation step. Similarly, the current implementation of inspecting first sets is limited by a number of cases which cannot trivially be inspected, namely `satisfy` and `re`. The latter is a significantly easier case, compared to the former which would require much deeper inspection into the predicate function. However both require the ability for the preprocessor to construct a set of characters which would fit the required pattern.

Further Static Analysis Since the preprocessor has a global scope, it would be feasible to perform further analysis on the program. One example of this would be inspecting arbitrary function calls when the definition is available. Recall the limitation first stated in subsection 4.4.4, where simple functions cannot be trivially inspected - if the preprocessor was able to determine the result of the function and whether it had any side effects, it would be feasible to hoist the definition out to perform further optimisations. The same could be done for declarations; if the value of a declaration is verified to not be changed through any execution path, then it could safely be inlined. Additionally, performing inspection into functions that are used in `<$>` can also be beneficial for optimisations - for example, if `f <$> p <*> q` was known to be equivalent to `p <* q`, a simple substitution could occur to allow for more effective optimisations later on.

General Optimisations for TypeScript While the preprocessor was designed for the optimisation of parser combinators, it is still capable of performing optimisations over the AST of a program regardless of the presence of combinators. This would be prior to any compilation to JavaScript, as the TypeScript compiler does not perform any optimisation. Simple examples of optimisations that could be performed include loop-invariant code motion, where code that has no side effects and exists within a loop can be lifted out. Performing these optimisations could also benefit the performance of the parsers. For example, section 3.3 highlights the performance uplift from replacing recursive calls with iteration - it would be possible to detect cases of tail recursion and replace them with semantically equivalent loops.

Bibliography

[1] Jamie Willis and Nicolas Wu. Garnishing parsec with parsley. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, page 24–34, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358361. doi: 10.1145/3241653.3241656. URL https://doi.org/10.1145/3241653.3241656.

[2] Paolo Martini and Daan Leijen. megaparsec: Monadic parser combinators, September 2015. URL https://hackage.haskell.org/package/megaparsec-5.3.0. Accessed: 2022-06-15.

[3] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, AT&T Bell Laboratories, 1975. URL http://dinosaur.compilertools.net/yacc/.

[4] Terence John Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: the power of dynamic analysis. In *OOPSLA 2014*, 2014.

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321486811.

[6] Jamie Willis and Nicolas Wu. Design patterns for parser combinators (functional pearl). In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, page 71–84, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386159. doi: 10.1145/3471874.3472984. URL https://doi.org/10.1145/3471874.3472984.

[7] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8, September 1999. doi: 10.1017/S0956796898003050.

[8] C. Allen, J. Moronuki, and S. Syrek. *Haskell Programming from First Principles*. Lorepub LLC, 2016. ISBN 9781945388033.

[9] Philip Wadler. How to replace failure by a list of successes. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*, page 113–128, Berlin, Heidelberg, 1985. Springer-Verlag. ISBN 3387159754.

[10] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, page 1–23, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3540594515.

[11] Graham Hutton. *Programming in Haskell*. Cambridge University Press, USA, 2nd edition, 2016. ISBN 1316626229. doi: 10.5555/3092752.

[12] Nicolas Wu. zenzike/yoda: A simple combinator library, November 2018. URL https://github.com/zenzike/yoda. Accessed: 2022-01-26.

[13] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, July 2001. URL https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007.

[14] Jeffrey Kegler. Parsing left recursions, June 2018. URL http://jeffreykegler.github.io/Ocean-of-Awareness-blog/individual/2018/05/lrecursion.html. Accessed: 2022-01-25.

[15] Mark-Jan Nederhof and Janos J. Sarbo. Increasing the applicability of LR parsing. In *Proceedings of the Third International Workshop on Parsing Technologies*, pages 187–202, Tilburg, Netherlands and Durbuy, Belgium, August 1993. Association for Computational Linguistics. URL https://aclanthology.org/1993.iwpt-1.16.

[16] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 111–122, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113729X. doi: 10.1145/964001.964011. URL https://doi.org/10.1145/964001.964011.

[17] Alexander Birman. *The TMG Recognition Schema*. PhD thesis, Princeton University, USA, 1970. AAI7101582.

[18] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., USA, 1972. ISBN 0139145567.

[19] Andrew Partridge and David Wright. Predictive parser combinators need four values to report errors. *Journal of Functional Programming*, 6(2):355–364, 1996. doi: 10.1017/S0956796800001714.

[20] Ecma International. Ecmascript 2015 language specification, 2015. URL https://www.ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf. Accessed: 2022-01-04.

[21] Eugene Yokota. scopt/scopt - command line options parsing for Scala, January 2022. URL https://github.com/scopt/scopt. Accessed: 2022-05-31.

[22] Prettier. Prettier - opinionated code formatter, Unknown. URL https://prettier.io/. Accessed: 2022-05-31.

[23] Microsoft. Typescript language specification, 2016. URL https://raw.githubusercontent.com/microsoft/TypeScript/main/doc/TypeScript%20Language%20Specification%20-%20ARCHIVED.pdf. Accessed: 2022-01-04.

[24] James Power. Notes on formal language theory and parsing. October 1999. doi: 10.1.1.43.2472.

[25] Steve Hill. Continuation passing combinators for parsing precedence grammars. Technical report, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, November 1994. URL https://kar.kent.ac.uk/21168/.

[26] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18:1–13, 01 2008. doi: 10.1017/S0956796807006326.

[27] Doaitse Swierstra. Combinator parsers: From toys to tools. *Electronic Notes in Theoretical Computer Science*, 41:38–59, 01 2000. doi: 10.1016/S1571-0661(05)80545-6.

[28] Brent Yorgey. Typeclassopedia - haskellwiki, 2009. URL https://wiki.haskell.org/Typeclassopedia. Accessed: 2022-06-06.

[29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935915. doi: 10.1145/1294261.1294281. URL https://doi.org/10.1145/1294261.1294281.

[30] Ralph Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO '87*, volume 293, pages 369–378, 08 1987. ISBN 978-3-540-18796-7. doi: 10.1007/3-540-48184-2_32.

[31] W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, jul 1965. ISSN 0001-0782. doi: 10.1145/364995.365000. URL https://doi.org/10.1145/364995.365000.

[32] Education First. 1000 most common words in English, Unknown. URL https://www.ef.co.uk/english-resources/english-vocabulary/top-1000-words/. Accessed: 2022-05-20.

[33] Clinton F. Goss. Machine code optimization - improving executable object code. *CoRR*, abs/1308.4815, 2013. URL http://arxiv.org/abs/1308.4815.

[34] Shahar Soel. https://chevrotain.io/performance/, June 2015. URL https://chevrotain.io/performance/. Accessed: 2022-01-26.

[35] Microsoft Open Source. Github - microsoft/ts-parsec, August 2019. URL https://github.com/microsoft/ts-parsec. Accessed: 2022-01-26.

[36] Jeanie Adkisson, Han Seoul-Oh, and Brian Mock. Github - jneen/parsimmon, December 2021. URL https://github.com/jneen/parsimmon. Accessed: 2022-01-26.

[37] Jean Bovet and Terence Parr. Antlrworks: The antlr gui development environment, October 2013. URL https://www.antlr3.org/works/. Accessed: 2022-01-25.

[38] Unknown. Left-recursion removal - antlr 3 - confluence, May 2008. URL https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687334/Left-Recursion+Removal. Accessed: 2022-01-25.

[39] Richard A. Frost and Rahmatullah Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *SIGPLAN Not.*, 41 (5):46–54, may 2006. ISSN 0362-1340. doi: 10.1145/1149982.1149988. URL https://doi.org/10.1145/1149982.1149988.

[40] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics - COLI*, 17, January 1991.

[41] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *PADL*, 2008.

[42] S. Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, volume 1129, July 1996. ISBN 978-3-540-61628-3. doi: 10.1007/3-540-61628-4_7.

[43] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1):67–111, May 2000. ISSN 0167-6423. doi: https://doi.org/10.1016/S0167-6423(99)00023-4. URL https://www.sciencedirect.com/science/article/pii/S0167642399000234.

[44] Ross Paterson. Arrows: A general interface to computation, Unknown. URL https://www.haskell.org/arrows/. Accessed: 2022-06-16.

[45] Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self inspecting code. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 69–79, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138504. doi: 10.1145/1017472.1017485. URL https://doi.org/10.1145/1017472.1017485.

[46] Jamie Willis, Nicolas Wu, and Matthew Pickering. Staged selective parser combinators. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi: 10.1145/3409002. URL https://doi.org/10.1145/3409002.

[47] Matt Bierner. Home mattbierner/khepri wiki, September 2014. URL https://github.com/mattbierner/khepri/wiki. Accessed: 2022-01-26.

[48] Matt Bierner. Parsing user defined operator expressions in a scripting language - uwtb, November 2014. URL https://blog.mattbierner.com/user-defined-operators-in-a-dynamic-scripting-language/. Accessed: 2022-01-26.

Appendix A

Supplementary Material

Combinator Types (*Teaspoon*)

function	**operator**	**LHS type**	**RHS type**	**combinator type**
`ap`	`<*>`	`P<(a: A) => B>`	`P<A>`	`P<B>`
`pa`	`<**>`	`P<A>`	`P<(a: A) => B>`	`P<B>`
`apL`	`<*`	`P<A>`	`P<B>`	`P<A>`
`apR`	`*>`	`P<A>`	`P<B>`	`P<B>`
`choice`	`<|>`	`P<A>`	`P<A>`	`P<A>`
`mult`	`<~>`	`P<A>`	`P<B>`	`P<[A, B]>`
`multL`	`<~`	`P<A>`	`P<B>`	`P<A>`
`multR`	`~>`	`P<A>`	`P<B>`	`P<B>`
`fmap`	`<$>`	`(a: A) => B`	`P<A>`	`P<B>`
`pamf`	`<&>`	`P<A>`	`(a: A) => B`	`P<B>`
`constFmapL`	`<$`	`A`	`P<B>`	`P<A>`
`constFmapR`	`$>`	`P<A>`	`B`	`P<B>`
`liftCons`	`<:>`	`P<A>`	`P<A[]>`	`P<A[]>`
`label`	`<?>`	`P<A>`	`string`	`P<A>`

Table A.1: Implemented combinators and their respective types, `P` denotes `Parser` for brevity

CLI Options

abbreviation	**type**	**description**
`-h`	×	Displays the help menu
`-o`	file	(**required**) Path to target (new) file
`-p`	toggle	Reformats the generated code (via the use of *prettier*)
`-a`	toggle	Enable all optimisations
`-icc`	toggle	Implicitly convert chars / strings when used directly with a combinator
`-cr`	toggle	Remove redundant alternatives
`-lra`	toggle	Analyse and attempt remediation of left-recursive productions
`-st`	toggle	Analyse string choice chains and optimise
`-br`	toggle	Analyse attempt chains and optimise to reduce backtracking by factoring
`-plo`	toggle	Basic optimisations using parser laws
`-ngfs`	toggle	Restrict first set analysis to local declaration (global by default)

Table A.2: Options and flags available for the CLI

Example AST

```
BinaryOpExpr(
  BinaryOpExpr(
    BinaryOpExpr(
      Identifier("expr"),
      BinaryOpExpr(
        ArrowFunctionIdentifier(
          Identifier("x"),
          ArrowFunctionIdentifier(
            Identifier("y"),
            BinaryOpExpr(Identifier("x"), Identifier("y"), "+")
          )
        ),
        MemberCall(
          Identifier("chr"),
          Arguments(None, List(Literal("'+'")))
        ),
        "<$"
      ),
      "<**>"
    ),
    Identifier("nat"),
    "<*>"
  ),
  Identifier("nat"),
  "<|>"
)
```

Listing A.1: Parsed AST for running example of simple addition calculator