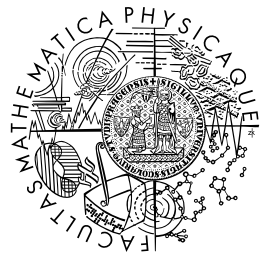

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Michal Tichý

ParsecCore: A parser combinator library in the C# language

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Filip Kliber
Study programme:	Computer Science
Study branch:	Programming and software development

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor Mgr. Filip Kliber for his time and for the advice he gave me. I would also like to give special thanks to my family for supporting me during my studies and my friends for their emotional support.

Title: ParsecCore: A parser combinator library in the C# language

Author: Michal Tichý

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Filip Kliber, Department of Distributed and Dependable Systems

Abstract: In this thesis, we implement a parser combinator library in C# inspired by Parsec. Parser combinators refer to a style of parsing where a parser is modeled as function from string to some structured result. Afterward, higher-order functions called combinators are used to combine simple parsers into more complex ones. Other such libraries exist, but in our estimation, they contain various deficiencies. We create an implementation without these faults and also introduce two new modules for parsing indentation-sensitive languages and permutations. We additionally implement two example parsers to showcase our library.

Keywords: C#, csharp, parser, combinator, syntax analysis

Contents

Introduction

One of the essential parts of computer science and programming is *parsing* — the process of analyzing a sequence of symbols conforming to some *formal grammar* and transforming them into a structured result that can be further used.

One popular approach to parsing in functional programming languages is using *parser combinators*. *Parsers* in functional languages can be defined as functions that take a string of tokens as input and output a structured result. Parser combinators then refer to higher-order functions that take several simpler parsers and create a new, more complex parser out of them [1].

This approach to parsing is quite popular due to the ease with which new parsers can be constructed. The parser can be written in the very same language we are programming the rest of our application in. This means we have the bonuses of the original language, such as type checking and the development environment. Furthermore, since the parsers are first-class values in the language we are programming in, it is easy to extend the set of the already available parsers with a custom parser [1] [2].

The parser combinator approach is commonplace in functional programming languages but not very common in imperative languages. One of the most popular imperative programming languages is C# [3], which is, compared to other languages, particularly well suited to have a parser combinator library written in it. A few solutions which introduce parser combinators into C# exist. However, all of them have issues regarding their implementation; They do not work with large inputs, have missing features, or have missing documentation. We aim to fix these deficiencies and create a well-documented, fully-featured parser combinator library for the C# language.

One well-known parsing library that uses parser combinators is *Parsec* [4]. Parsec is, as already stated, a parser combinator library written in Haskell [5] and created by Daan Leijen in 2001. It can parse context-sensitive, infinite look-ahead grammars [2].

In the next chapter, we will show that Parsec has a sound formal background and justify its usage as a basis for a new parsing library. The popularity of Parsec also shows that its general structure is easy to work with and makes for a good

foundation for our own parser combinator library if we are able to translate its structure into C# effectively.

Goals

We have three primary goals for our thesis.

1. Implement a parser combinator library in C# based on Parsec with all basic combinators in Parsec also present in ours.
2. Create examples that are able to parse some non-trivial grammar. In this way, we showcase our library and prove it can be used to implement programs that work with complicated data.
3. Extend our library to implement non-trivial combinators not present in other C# parser combinator libraries.

Thesis Layout

In the first chapter, we introduce needed definitions related to parsing and situate parser combinators into a broader context of parsing. We also show how a parser combinator library works by showing and describing examples of Parsec. Finally, we introduce The LINQ syntax of the C# language.

We analyze how we will implement our library in the second chapter. We show the relation of the LINQ syntax to Haskell do notation. We review current implementations of parser combinator libraries in C#, note their shortcomings, and try to think of ways to improve them. We also describe and propose solutions to problems related to implementing an architecture initially used in functional languages in an imperative one. Finally, we discuss how to extend our library compared to other solutions.

The third chapter talks about the implementation of our library and discusses any interesting issues that occurred. It also describes how to modify and extend the library.

The fourth chapter is the user's documentation. We describe the structure of the library in detail from the point of view of a programmer merely using our library. Using examples, we will describe the library's features, explain the semantics of different parsers and advise on using the existing parsers and parser combinators.

In the fifth chapter, we discuss the examples that come with the library. Using these examples, we show that our library is usable even for quite complex grammars.

The sixth chapter focuses on evaluating the performance of our library. We benchmark our library and compare it with other solutions for creating parsers. We also describe the process of improving the library's performance.

Finally, in the conclusion, we discuss how successful we were in achieving our goals. In addition, we mention possible improvements to the library.

Chapter 1

Theoretical background

In the first part of this chapter, we situate our library in a larger context of parsing. We define terms used later in the thesis. Furthermore, we introduce aspects of parsing that explain the semantics of our combinators.

In the second part, we look at Parsec and what a parser combinator library looks like using examples.

Finally, in the third part, we introduce C# Language integrated query, or LINQ for short, that we will use extensively in our library.

1.1 Grammars

As stated in the introduction, parsing is the process of analyzing a sequence of symbols conforming to some *formal grammar*. Let us define formal grammars and describe some of the related terms. The definitions and descriptions for this section as well as the following section are taken from *Compilers: Principles, Techniques, and Tools (2nd Edition)* [6] and lecture notes from Stanford University [7] [8].

Definition 1 (Formal grammars)**.** *Grammars are used to formally describe the syntax of data formats and programming languages. A formal grammar has four components:*

1. *A set of* nonterminals. *A variable symbol that can be replaced or expanded to a sequence of grammar symbols*
2. *A set of* terminal *symbols. They are the elementary parts of the language defined by the grammar*
3. *A set of* productions*, rules in the grammar that describes how to replace symbols.*

4. A start symbol. A single nonterminal is designated to be the starting symbol.

Definition 2 (Productions)**.** *Productions have the general shape*

$$u \to v$$

where both u and v are a sequence of grammar symbols. Such a production tells us that we can substitute a sequence of symbols u for the sequence v.

Depending on the form of the productions we distinguish between four types of grammars. In this thesis, we are only concerned with two:

- Type 2 grammars, also known as *context-free* grammars
- Type 1 grammars, also known as *context-sensitive* grammars

The productions in context-free grammars are of the form

$$N \to u$$

where N is a single nonterminal and u is an arbitrary sequence of terminal and nonterminal symbols. The production signifies a rule in the grammar that we can substitute u for the nonterminal N. The grammars are called context-free because there is a single nonterminal on the left-hand side of the production; This is in contrast to context-sensitive grammars.

One nonterminal can be on the left-hand side of multiple productions. In such a case, we can write all such productions with a single left-hand side and a sequence of the differing right-hand sides separated by '|'. If a nonterminal N appears on the left-side of n productions with right-hand sides $r_1, \ldots, r_n$ then we can write all of the productions like so:

$$N \to r_1 \mid r_2 \mid \ \ldots \ \mid r_n$$

Context-sensitive grammars contain productions of the form

$$aNb \to aub$$

where a, b and u are arbitrary sequences of terminal and nonterminal symbols and N is a single nonterminal symbol. This production tells us that we can substitute u for the nonterminal N if N is surrounded by the sequences a and b from the sides. We can consider a and b as the context of the production and thus the name context-sensitive grammars.

Similar to context-free grammars, if the left-hand sides of multiple productions are the same, then we can write them with the aforementioned shorthand.

1.1.1 Derivations

A grammar generates a set of *sentences*. A sentence is a string consisting entirely of terminal symbols of the grammar. We generate a sentence by starting with the start symbol and repeatedly applying rules for rewriting nonterminals, productions, until no nonterminals are left. This process is called a *derivation*, also known as a *parse*.

The set of sentences a grammar can generate is called the *language* of that grammar. All languages that a context-free grammar can generate can also be generated by context-sensitive languages but not the other way around.

Derivation, more formally, is a sequence of applications of the productions that generates a finished string of terminals. If it is possible to apply more than one production, we choose any one. In this way, we define the semantic meaning of the aforementioned separator '|' as the operator for *unordered choice*; We nondeterministically choose any of the right-hand sides in the production. It is important to note that we care not only about the final string of terminals but also about the order in which the productions were applied.

We thus define *leftmost derivation*, a type of derivation where we always substitute for the leftmost nonterminal. We still have to choose from multiple productions if the leftmost nonterminal is on the left-hand side of multiple productions, but we limit the choice in terms of which nonterminal to choose.

Another way we can understand a derivation is as the sequence of how a sentence was parsed. From this point of view, if there exists more than one derivation which generates the same sentence, then there is more than one way to parse this sentence. If such two derivations exist in a grammar, then we say that grammar is *ambiguous*.

Ambiguity is unacceptable in a grammar meant to parse, for instance, a programming language. If such a grammar was ambiguous, then, for example, different implementations of compilers could interpret arithmetic expressions differently, and such expressions would have differing results. Thankfully we will side-step the problem of ambiguity in our library later in this chapter.

1.2 Top-down parsers

With formal grammars defined, we can next examine parsers through the lens of grammars. We can look at parsers as an implementation of a grammar. Instead of talking about generating sentences, however, we talk about recognizing, or *parsing*, an input string of terminals, *symbols*, of the grammar. In summary, a language has a grammar and the grammar is implemented by a parser.

There are two major parsing approaches: *top-down* and *bottom-up*. Though

bottom-up parsers are an important part of parsing theory, we will not discuss them in this thesis as they are not relevant to our topic.

In top-down parsing, we start with the start symbol and apply productions until we reach the desired string. We will showcase this process using an example. We have the following grammar:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E\ Op\ E \mid (\ E\ ) \mid int \\ Op &\rightarrow + \mid * \end{aligned}$$

The nonterminals are S, E, and Op with S being the start symbols. The terminals are '(', ')', '$+$', '$*$', and 'int'.

The parse for the string "$int + int + int$" is as follows. We start with the start symbol. We are applying the production on the right side to the sequence on the left side.

S	$S \rightarrow E$
E	$E \rightarrow E\ Op\ E$
$E\ Op\ E$	$E \rightarrow E\ Op\ E$
$E\ Op\ E\ Op\ E$	$E \rightarrow int$
$int\ Op\ E\ Op\ E$	$Op \rightarrow +$
$int\ +\ E\ Op\ E$	$E \rightarrow int$
$int\ +\ int\ Op\ E$	$Op \rightarrow +$
$int\ +\ int\ +\ E$	$E \rightarrow int$
$int\ +\ int\ +\ int$	

We thus derived or parsed the input string "$int + int + int$".

In a sense, the top-down parser guesses the structure of the input and afterward checks with the input string. In this small example, it is easy to see which production to use as we see the entire input string. However, in real-world parsers, we do not see the entire input string, and as such, the problem of which production to choose becomes much harder.

1.2.1 Predictive parsers

A possible implementation of top-down parsing is a *predictive parser*. A predictive parser chooses the next productions based solely on the following k symbols in the input, where k is some fixed constant. For this to be possible, a grammar has to be of a particular form.

These parsers are also called *LL*, *left-to-right*, parsers, and the grammars that they parse are called LL grammars. As the name suggests, they parse the input string from left to right and always perform the leftmost derivation in doing so.

The number of upcoming symbols in the input the parser can see is called the *lookahead*. The LL parser is usually parametrized with the natural number k, $LL(k)$, signifying its lookahead.

A grammar is an $LL(k)$ grammar if there exists an $LL(k)$ parser that parses it. The class of grammars for which there exists an $LL(k)$ parser is also called the $LL(k)$ class.

Any $LL(k+1)$ class is a strict superset of the $LL(k)$ class.

1.2.2 Backtracking parsers

Another possible implementation of top-down parsing would be using *backtracking*. We can think of backtracking as an implementation of the guessing mentioned previously.

Based on the available symbols, the parser makes a decision about which production to go with. If this decision leads to a dead-end, we must return to the decision point and try a different production. This involves going back in the input. We try every production at the decision point until one succeeds. If all productions fail, then the production we are in right now also fails.

Backtracking parsers are sometimes thought of as predictive parsers with unlimited lookahead. Thus they are sometimes informally denoted as LL(∞) parsers.

Backtracking parsers are, therefore, more powerful than predictive parsers. However, this greater ability comes with the cost of having to potentially retry the parsing of a section many times.

1.2.3 Left recursion

Productions are often defined in terms of themselves. For illustration, in our example grammar, the nonterminal E appears on both sides in the production $E \to E\ Op\ E$. We call such productions *recoursive*.

If the recursive nonterminal is the leftmost symbol in the production, such as in our example production $E \to E\ Op\ E$, then we call that production *left-recoursive*. Left-recursive grammars are problematic for top-down parsers as we can end up in an infinite loop of always choosing the left-recursive production.

Left recursion can even occur across multiple productions, such as in this example:

$$A \to B\,a \mid a$$
$$B \to A\,b \mid b$$

If, for example, in a backtracking parser we always first try the first production, then we end up in an infinite loop.

Thankfully left recursion can be wholly eliminated from a grammar with a simple algorithm that does not change the language it generates. However, it is still an issue that must be accounted for.

1.3 Parsing expression grammars

In this section, we will introduce a formalism more similar to top-down parsers and, by extension, Parsec. We introduce the concept of *prioritized (ordered) choice*. We also discuss the class of languages that Parsec, and therefore our future library, can parse. The following definitions and descriptions are taken from "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation" [9] and "Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking" [10].

Although we have so far described top-down parsers in terms of context-free grammars, actual real-world implementations of top-down parsers are closer to a formalism called *Parsing expression grammars*, also PEGs from now on.

PEGs are a recognition-based formal foundation for language syntax inspired by context-free grammars and regular expressions. Recognition-based in this context means that instead of generating sentences, we talk about recognizing or parsing input strings and either accepting or rejecting them. The most significant departure of PEGs from context-free grammars is the introduction of *prioritized choice*. Instead of choosing any one production when multiple substitutions are possible, we choose the one with the highest precedence that we did not try yet.

Parsing expression grammars consist of rules. Rules are of the form

$$N \leftarrow E$$

where N is a nonterminal and E is a *parsing expression*. A parsing expression is a list of sequences consisting of terminals, nonterminals, and operators separated by the prioritized choice operator '/'. The precedence is given by the position in the list; the earlier in the list, the higher the precedence. Therefore, for example, in the rule:

$$N \leftarrow \text{ab} \ / \ \text{a}N$$

we first try to parse the string 'ab', and afterward, if the parse of 'ab' fails, we try to parse 'aN'.

Since the precedence is always clear and we are using only leftmost derivations, there is no ambiguity in the parse. Thus we remove a major problem in the construction of parsers. Of course, instead of ambiguity, we move the problem onto a different aspect of the grammar as the order of the alternatives in a '/' expression matters.

The definition of PEGs is similar to formal grammars. We again have terminals and nonterminals, but instead of productions, we have rules and parsing expressions. A constraint on rules is that a nonterminal can, at most, appear on the left-hand side of one rule. Finally, along with a sequence of terminals and nonterminals and prioritized choice, parsing expressions can also contain the following operators:

- e^*, zero-or-more repetitions
- $!e$, a not-predicate

We can add more operators taken from regular expressions, such as $e?$ or $e+$, as syntax sugar on top of already defined operators.

All of the expressions are defined in terms of backtracking. The sequence $e_1 e_2$ tries to parse expression e_1 immediately, followed by e_2. If any of them fails, the entire expression fails, and we backtrack in the input. The prioritized choice first tries the expression e_1, and if it fails, it backtracks to the beginning and then tries the expression e_2.

The repetition operator is greedy, meaning it will keep applying the expression e as long as possible and thus consume as much input as possible.

The parsing expression $!e$ is the same as the expression e except with an inverted result. If the parsing expression e accepts the input, then $!e$ rejects it and vice-versa. Whether the expression $!e$ succeeded or failed, we backtrack to where we started before trying this expression.

Most conventional syntax descriptions are split into two parts:

- A context-free grammar to specify the hierarchy of the syntax elements.
- Regular expressions to recognize the lexical elements in the syntax that serve as terminals in the CFG.

CFGs are not suitable for lexical analysis because they lack some idioms of regular expressions, such as the greedy rule that applies to numbers or the "negative" rule often used in strings. However, neither of these problems exists in PEGs, and thus PEGs can be used simultaneously for both lexical and syntax analysis.

As with other top-down parsers, left recursion is not possible in PEGs as it represents a degenerative loop. However, since recursive productions are often used in context-free grammars to represent iteration and PEGs have the repetition operator e^*, the lack of left recursion is not as restrictive, even without rewriting the grammar to eliminate left recursion.

1.3.1 Context-free grammar versus a PEG

It is an open question whether PEGs can recognize all languages that a context-free grammar can generate. We do know, however, that PEGs can recognize some languages only recognizable by context-sensitive languages. And although we do not know if all context-free grammars have their equivalence in PEGs, we can safely assume that any reasonable language specified by a context-free grammar can also be recognized by a PEG.

1.4 Parser combinators and Parsec

In this section, we will describe an example of a parser combinator library in Haskell. This library will be a simplified version of Parsec. Through it, we will describe how the grammar constructions from PEGs — such as sequencing, choice, and repetition — are translated into combinators. The library will thus be able to recognize all languages that PEGs can. The structure of the library, as well as some of the descriptions, are taken from "Monadic parser combinators" [1], "Parsec: direct style monadic parser combinators for the real world" [11], and "Parsec, a fast combinator parser" [2].

We model a simple parser as a function that takes input token types, usually strings, and outputs either the structured result of the parse or an error message signifying failure. Along with the result, the function also outputs the remaining input that it did not parse.

```
1  newtype Parser a = ParserCstr String -> (String, Either ParseError a)
```

`Parser` is the name of the type. The `Parser` can be parametrized by any type, here represented by the type variable `a`. This type variable signifies the structured result that our parser outputs. `ParserCstr` is the constructor of the type. As we can see, the Parser takes a string as input. Afterward, it outputs a pair of the remaining input that we did not parse and either a `ParseError` or the structured result we parsed.

1.4.1 Basic parsers

We build a parser combinator library from the simplest parsers and combinators. The simplest parser usually included is the *satisfy* parser. Satisfy parser is constructed using a predicate that takes one input symbol. Afterward, during parsing, it reads one input symbol and either accepts it or rejects it based on the predicate. Here we can see the type of the satisfy function which constructs our `Parser`.

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

In Haskell, the identifier before the pair of colons is the name we assign to an expression. After the pair of colons is the type of the expression. This type has two parts: a function from `Char` to `Bool` as input and `Parser Char` as output. The operator `->` signifies a function in Haskell. If there are multiple `->` operators in a row without parenthesis, we can understand it as a function that takes all but the last type as input and outputs the last type.

From this parser, we can construct any number of parsers. For example, we can construct parser *char* that accepts only a specific symbol, a parser that accepts only if the symbol is in a given set, a parser that accepts any symbol, or many more.

Another basic parser is the *return* parser. This parser always succeeds and returns whatever structure we pass to it during its creation.

The last basic parser we will mention is the *fail* parser that always fails.

1.4.2 Sequencing and 'do' notation

So far, we can parse a single symbol. That is our basic building block. Nevertheless, we need a way to create new parsers which parse a sequence of symbols. There are two ways this is usually accomplished. The first is the *arrow-style* combinator, and the second is the *bind* combinator.

```
1  arrow :: Parser a -> Parser (a -> b) -> Parser b
2  bind :: Parser a -> (a -> Parser b) -> Parser b
```

The arrow-style combinator receives and sequences two parsers to be applied after each other. The first parser returns a generic type `a`, and the second one returns a function, `a -> b`, that takes the result of the first parser and creates the final result. Since the function is returned by the second parser, the function cannot affect the second parser. Therefore, the second parser cannot depend on the runtime result of the first parser. The arrow-style combinator can thus only parse context-free grammars.

On the other hand, the second argument in the bind combinator is a function that takes the result of the first parser and only afterward returns the parser to apply. Therefore the second parser can depend on the runtime result of the first parser. This means we can parse some context-sensitive grammars. We can imagine that, for example, we parse a format version number, and depending on it, we choose which parser to use further.

Both combinators are present in Parsec, although the bind combinator is used much more often.

It is customary to provide both combinators as binary infix operators. The combinators are aliased as the following operators.

```
(<*>) = arrow
(>>=) = bind
```

The following code sequences two parsers that parse the character ‘a’ and ‘b’ respectively and returns the string “ab”. Note that a string in Haskell is simply a list of characters, and list literals are written between square brackets.

```
ab :: Parser String
ab =
    (char 'a') >>= \parsedA ->
    (char 'b') >>= \parsedB ->
    return [parsedA, parsedB]
```

The parser bind combinator is a powerful concept and is a specific instance of a much more general idea that is the bind function of a *monad*. However, if we needed to sequence multiple parsers, then the syntax would be cumbersome. Therefore Haskell provides syntactic sugar that makes writing bind combinators much easier. This syntactic sugar is called the *do notation* and looks the following way.

```
ab :: Parser String
ab = do
    parsedA <- char 'a'
    parsedB <- char 'b'
    return [parsedA, parsedB]
```

This code is equivalent to the previous example. The parsers `char 'a'` and `char 'b'` both put their results into variables `parsedA` and `parsedB`, respectively. We can use the `parsedA` variable everywhere after it has been defined, including in the invocation of the second parser. If any of the parsers fail, the sequencing fails as well.

1.4.3 More combinators

We will quickly go over other essential combinators. Most of these are implementations of the operators from parsing expressions of PEGs. With these added, we can finally recognize all languages that a PEG can.

Choice

Another important combinator is the *choice* combinator. We define the binary infix operator <|> that implements choice. It takes two parsers and creates a parser that tries first the left parser and optionally the right one.

The choice combinator has similar semantics to the prioritized choice in PEGs. However, whereas the prioritized choice in PEGs backtracks if the first parser fails,

in Parsec, the second parser is attempted only if the first parser does not consume any input. If the first parser fails while consuming input, then the entire parser fails outright, and the second one is not even attempted. Parsec is, therefore, a predictive LL(1) parser by default. There are ways to add backtracking to the parser, but the user has to explicitly state where.

Try and backtracking

The way to add backtracking is the *try* combinator. The try combinator takes a single parser and adds backtracking to it.

The parser returned by the try combinator tries to parse the same as the original parser. If the original parser succeeds, then the entire parser succeeds. If the original parser fails, then the parsing fails as well. However, it also backtracks in the input to the position with which it started.

Having a separate combinator only for adding backtracking allows for more modularity. It, for example, also allows us to include lexical as well as syntax analysis in a single parser by wrapping the parser of each lexical token in the try combinator. In practice, the try combinator is only needed for grammar construction that requires lookahead.

Left recursion

Parsec cannot deal with left recursion, the same as PEGs. Nevertheless, this is a problem that is solvable since, as stated previously, every left-recursive grammar can be rewritten into a right-recursive one that we are able to parse. It is also possible to create a combinator *chainl* that parses left-associative binary operators that are often the source of left recursion in the grammar.

Repetition

Repetitions are added using the `Many` combinator. This combinator takes a parser, we will call it the *item parser*, and tries to apply it as many times as possible until it fails. The results are then collected into a list and returned.

```
1 many :: Parser a -> Parser [a]
```

The only difference from the repetition PEG operator is that `Many` fails if the item parser fails while consuming input. We can solve this discrepancy by adding the `Try` combinator.

Not predicate

The last combinator we need to implement all of the operators of PEGs is the `NotFollowedBy` combinator. This combinator has the same function as the not-predicate of the PEGs. It succeeds if the original parser fails and vice-versa. Again, if we want the combinator to behave exactly like the PEG operator, we need to add the `Try` combinator.

1.4.4 Errors

Errors in Parsec report the location at which the error occurred and the cause of the error. Additionally, the error can contain messages about all possible productions that would have been legal at that point in the input.

Parsec is able to report on all possible productions at a point in the input by dynamically computing the messages during parsing. Thanks to Haskell's laziness, such a computation is not very expensive as this calculation only happens when an error occurs.

The computation starts at the satisfy parser. In case of failure, the satisfy parser reports the current position of the error and the token that was unsuccessfully parsed.

The second important step of the computation occurs in the choice combinator. The error messages that tell us about the expected productions are combined if they occurred at the same place in the input. If they did not occur at the same position, then the error message that occurred later in the input is preferred.

1.5 LINQ

In this section, we will discuss the *Language integrated query* or *LINQ* for short. LINQ adds queries as first-class language constructs to C#. Thus LINQ adds type-checking and IntelliSense support to queries in the C# language. LINQ also serves as an abstraction over multiple query languages. LINQ is implemented in the standard C# library for querying over objects, relational databases, and XML [12].

The LINQ syntax is similar to SQL, except that the order of the clauses is reversed. A LINQ query consists of commands about what information to retrieve and how the information should be shaped afterward. The most basic query has a `from` clause and a `select` clause [13].

```
1 var data = int[] { 0, 1, 2, 3, 4, 5 };
2 var query =
3     from datum in data
4     select 2 * datum;
```

The `from` clause signifies what data we are retrieving. The `select` clause sets what data we are selecting and also how we are mapping it. This query will thus return the contents of `data` multiplied by two.

Additionally, the `where` clause provides filtering of the data.

```
1 var data = int[] { 0, 1, 2, 3, 4, 5 };
2 var query =
3     from datum in data
4     where (datum % 2) == 0
5     select 2 * datum;
```

This query will only select even numbers from the data and multiply them by two.

There are more clauses in LINQ. However, we will not use them in our library. As such, we will leave them without discussion.

The creation of the query itself does not create any new data; The execution of the query is distinct from the query itself. The data retrieval is lazy and happens only after query execution, such as in a `foreach` statement [13].

1.5.1 Creating our own LINQ

So far, all queries we have shown are in the declarative *query syntax*. The queries, nevertheless, have to be translated during the compilation into method calls. We can even call these methods directly ourselves; this is called the *method syntax*. These methods have a name similar to that of the clauses which are translated into them. LINQ is simply syntax sugar for the extension methods used in method syntax [14].

For illustration, the previous example would be translated into the following method calls.

```
1 var data = int[] { 0, 1, 2, 3, 4, 5 };
2 var query = data
3     .Where(datum => datum % 2 == 0)
4     .Select(datum => 2 * datum);
```

Extension methods are methods that "extend" an existing type. They can be called as if they were instance methods on the type, even though they are static methods defined outside the original class. We can add new extension methods to any type. Thus we can add LINQ query operators to a type we want to add query syntax to.

The most basic LINQ extension methods are the `Select` and `SelectMany` extension methods. These allow us to create queries with a single and also multiple `from` clauses and combine the retrieved data in the final `select` clause.

In summary, we can create extension methods for types to which we want to add LINQ syntax. What extension methods we implement determines what clauses we can use. Whenever we use the LINQ syntax with the given type, presupposing we have our extension methods imported, the LINQ syntax is translated into method calls of the extension methods that we have provided.

Chapter 2

Analysis

We have explained terms related to parsing that we will be using throughout the thesis. We have also displayed the basics of a parser combinator library. Now it is time to analyze how we will translate Parsec into C#.

We will first explain how to use the LINQ notation as an approximation of the do notation from Haskell. Afterward, we will look at existing implementations of a parser combinator library in C#. From this, we will formulate concrete goals that we want to achieve. Lastly, we will explain the design decisions that we have made to accomplish the set-out goals.

2.1 LINQ as do notation

We have already talked about LINQ (see 1.5). We have also discussed the do notation in Haskell in section 1.4.2. Now it is time to put these two concepts together.

`SelectMany` is one of the extension methods used in LINQ. If we look at its signature, we can see it is very similar to the monadic bind in Haskell. It is especially similar once we substitute the type `Parser` to the `SelectMany` signature. Note that the type `Func` in C# represents a function that takes all except the last type parameter as inputs and outputs the last type parameter.

```
1 public static Parser<TResult> SelectMany<TSource, TSecond,
    TResult>(
2     this Parser<TSource> source,
3     Func<TSource, Parser<TSecond>> collectionSelector,
4     Func<TSource, TSecond, TResult> returnSelector
5 );
```

```
1 bind :: Parser a -> (a -> Parser b) -> Parser b
```

In fact, the functions are almost the same, and the original report on the proposal for LINQ [15] calls `SelectMany` "monadic bind".

As noted by Luke Hoban [16], LINQ queries can express the monadic computation of parsing. We simply need to implement the extension methods `Select` and `SelectMany`. Additionally, we can also implement the `Where` extension method to add functionality.

The extension method `SelectMany` will serve for sequencing two parsers. If the first parser succeeds, then the second parser is also applied. If any of the parsers fail, then the entire sequence fails. Thus, as already mentioned, it is the equivalent of the bind combinator.

The method `Select` will map the result of a parser if it succeeds.

The method `Where` will test the result of a previous parser. If the test fails, then the entire query fails.

Finally, there is a fourth clause, the `let` clause. This clause allows us to define local variables in a query expression that we can use thereafter. For this clause, we do not have to implement any extension method. We can use it after adding the `Select` and `SelectMany` extension methods.

As an aside, the LINQ syntax does not have the full power of the do notation. We cannot, for example, implement loops and any other control flow directly in LINQ queries. We can implement external functions that output a parser and use control flow inside them. However, using this method frequently becomes cumbersome. Overall the LINQ syntax should be expressive enough to implement all usual operations used in parsing. [17]

The following example will be the same as in section 1.4.2, only translated into C#. First, we will try to parse the character 'a' using the parser `Char('a')`. Next, we will try to parse the character 'b' using the parser `Char('b')`. Finally, we will create a string using the parsed characters.

```
1  Parser<string> ab =
2      from a in Char('a')
3      from b in Char('b')
4      select new string(new char[] {a, b});
```

The next example will use all four clauses. We will parse a lowercase letter using the parser `Lowercase` and check that the parsed character is not 'z'. Subsequently, we will define a new character that is one greater than the parsed letter and try to parse it. Lastly, we will again construct and return a string.

```
1  Parser<string> ascendingLetters =
2      from firstChar in Lowercase
3      where firstChar != 'z'
4      let plusOne = firstChar + 1
5      from secondChar in Char(plusOne)
6      select new string(new char[] {firstChar, secondChar});
```

This translation of the do notation into LINQ is the reason we chose C#. It provides us with nice declarative syntax for combining parsers that, to our knowledge, are missing from other object-oriented languages.

2.2 Other implementations

Now we will examine other implementations of a parser combinator library in C#. We will look at their strengths and their deficiencies. This overview will help us decide which features to add to our implementation. It will also show us what we can improve.

There are several implementations. However, only a few are more widely used. It is primarily these upon which we will focus. We measured the popularity of these libraries by the Nuget [18] download counts.

2.2.1 Sprache

The most well-known parser combinator library in C# is *Sprache* [19]. Sprache promotes itself as a "simple, lightweight library for constructing parsers directly in C# code".

Sprache uses the LINQ syntax, explained above, for the purpose of constructing new parsers. It has all three `Select`, `SelectMany`, and `Where` extension methods, and thus clauses, defined.

Another positive is that Sprache also contains most of the basic parsers and combinators that are in Parsec. Including, for example, some of the more complex combinators such as *chain*.

Let us define what we mean by *basic Parsec parsers and combinators*. We mean the parsers and combinators that are located in the top Parsec module [20] under the 'Combinator' and 'Character Parsing' headings. Other headings document lower-level constructs tied to the specific implementation of Parsec. These parsers and combinators make the foundation for the entire library and constitute most of what Sprache implements. They also contain all combinators needed for the library to recognize all the same languages as PEGs. We can define them as a mandatory minimum that a parser combinator needs to implement to be usable.

Additionally, Sprache contains new primitive parsers for regex. These make writing parsers for numbers and other literals much easier for the user. It also has infrastructure for writing comment parsers. However, this help is, in our opinion, not that significant as comment parsers are usually easy to write even without this support.

The first negative we encounter is that there is no standalone backtracking combinator such as try or its equivalent. Now, Sprache still has the functionality

to backtrack. However, this ability is directly built into combinators. For most combinators, there is a normal variant and an 'X' variant. The normal variant has backtracking; the 'X' variant acts as an LL(1) predictive parser. This causes three problems. First, many combinators must be implemented twice, duplicating code. This means that any bug is duplicated. The two implementations can also get out of sync and behave differently. Second, parsers are less reusable, and adding and removing backtracking is more complicated. The last problem is that if a user wants to add another primitive combinator that supports backtracking, then he has to have more knowledge than what would be ordinarily necessary, as he needs to know how to implement a backtracking combinator.

Another issue is that the parsers cannot accept input tokens other than characters. While this is not a problem in the usual usage scenario, it can cause issues if the user wants to use a separate lexical analysis step. The user may want to do this because it reduces the number of times we need lookahead during parsing. There is currently no way to accomplish this in Sprache and significant additions to a large portion of the code would be needed to add this functionality.

Another pitfall we want to point out is that there is no support for reading characters from a stream. If we want to parse from a file, then the only way to do this is to load the entire file into memory as a string. Afterward, we can parse this string.

We would also welcome a more robust error system than what Sprache currently supports. The error system as it is now is functional and is inspired by the original Parsec error system (see section 1.4.4). However, the errors are built directly into the result of parsing instead of being their separate type. This means that a user can only add their own errors through string messages. Therefore, the errors are inflexible to change and make it hard to track that they remain consistent across the entire parser.

The last problem we want to discuss is documentation. Most parsers and combinators have comments. Nevertheless, most of these comments are very simple and short. The semantics of some of the parsers are non-trivial, but the comments, in our opinion, do not explain them sufficiently. For example, the `ChainOperator` is not a trivial combinator that parses a sequence of left-associative binary operators. However, the comment for the combinator only states "Chain a left-associative operator.". It does not mention how the combinator behaves if the operand parser fails or any other nuance.

In conclusion, Sprache is a fine library. Nevertheless, we believe there are many aspects that can be improved upon as well as features that can be added.

2.2.2 Superpower

Superpower is another popular parser combinator library in C#. [21] It is similar to Sprache. As such, some of the same issues remain while others are fixed.

Superpower has support for parsing tokens other than characters. It even has the infrastructure to write simple lexical tokenizers quickly. The tokenizers, ultimately, are written in the same style as other parsers. Thus this functionality can be emulated without such infrastructure. However, the additional combinators make this task easier.

Superpower has a separate backtracking combinator in contrast with Sprache. It thus avoids the duplication of combinators and gains in composability.

It also has better documentation than Sprache. Its parsers and combinators are well documented, and it also includes an introductory tutorial.

For all its improvements, there are still a few issues. Superpower does not support reading from a file or any other stream. Additionally, errors are still built into the result of the parsing. As such, the errors have to be completely written in strings, and any additional user errors are harder to implement.

Covariance

One new complaint we have is that the parser type is not covariant with respect to its result type. Variance describes how the subtyping of complex types should relate to the subtyping between their component types. Suppose that we have two types, `A` and `B`, and we have a more complex type `T<U>`. For the complex type `T<U>` to be covariant with respect to `U` means that if type `A` is more generic than `B`, then so is T<A> more generic than `T<B>`. [22]

Covariance in C# manifests itself in being able to assign a variable of the less general type to a variable with the more general type. For example: [23]

```
1 IEnumerable<string> strings = new List<string>();
2
3 // An object that is instantiated with a more
4 // derived type argument is assigned to an object
5 // instantiated with a less derived type argument.
6 IEnumerable<object> objects = strings;
```

It is sensible to make a function covariant with respect to its result type. [22] Parsers in the formulation of functional languages are functions from input tokens to a structured result. It thus makes sense to designate parsers as covariant with respect to their result types. Making parsers covariant causes them to be easier to combine. This is because we do not need to cast their results unnecessarily.

2.2.3 ParsecSharp

The last example of a parser combinator library we will look at is *ParsecSharp*. [24] It is not as popular as Sprache and Superpower. Nonetheless, we want to examine it because this library supports reading characters from a file.

ParsecSharp is again similar to Sprache in its basic form. It has all of the basic Parsec parsers and combinators. It also contains many additional primitive parsers.

There are two problems we want to point out first. The first is that parsers are again not covariant The second problem is that documentation is severely lacking. And even the little documentation that is present is, in large part, in Japanese.

Parser input

Now we will discuss the aforementioned reading from files. One admirable property of ParsecSharp is that it tries to be entirely immutable, including the input. Having an immutable input makes it easier to reason about. Thus it is also easier to extend. ParsecSharp accomplishes having immutable input while also reading from a stream by using buffers.

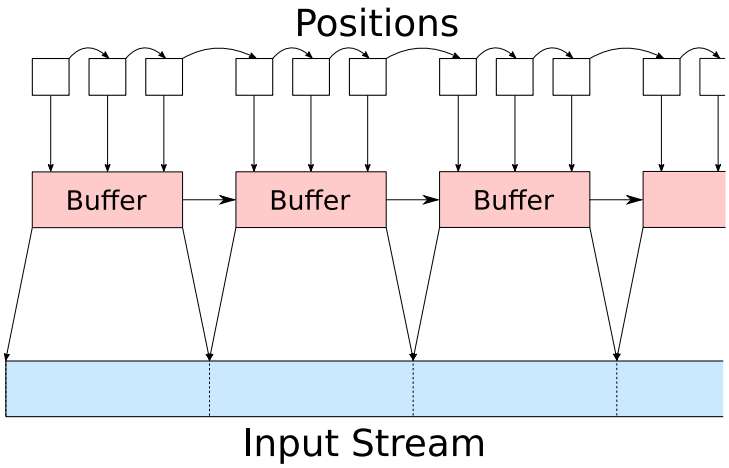

Figure 2.1 ParsecSharp's input implementation

Each position in the input is a separate object. Each of these objects points to the buffer that contains its place in the input. When we want the next position, we call a method on the current position. This method will give us the next position in the input. This method has two possible behaviors. The first occurs when the next position is also contained in the same buffer. We simply create a new

position object and point to the same buffer. The second behavior occurs when our position is on the edge of a buffer. Calling for the next position requires a new buffer to be constructed.

Buffers have a lazy reference to the next buffer in line. A lazy reference in this context means that a buffer is constructed only when we first try to access this reference. When we want to advance to the next position in the input, we construct a buffer and, afterward, create the position itself.

There are two problems with this approach. The first is that immutability is implemented using functions and continuations. However, C# does not optimize the calls, and as such, we run out of stack space on any larger input — about 200 lines of JSON.

The second problem is that, as far as we can see, this does not save any space compared to loading the entire input into memory. When the first position is used, the reference to it is located on the stack. When we ask for further positions, the reference to the first one is still located on the stack in the first activation record. Because of this, the buffer that holds the beginning of the input is held in memory and not garbage collected. And since every buffer has a reference to the next one, all of the buffers remain in memory. By the end of the parse, the entire input will thus be in memory.

As opposed to ParsecSharp, we will attempt to have the ability to parse directly from a stream without storing the entire input in memory.

2.3 Goals

We will now list concrete goals we have for our library. If we achieve these goals, then the starting goals we stated in the introduction will also be accomplished.

1. Implement the basic Parsec parsers and combinators
2. Provide the LINQ syntax for combining parsers
3. Support parsing of symbols other than characters
4. Implement the ability to read from files without the need to load the entire input into memory
5. Make the input interface immutable
6. Have the parsers be covariant with respect to their result type
7. Robust and extendable error system
8. Create an example parser to showcase our core library

9. Implement an extension to the library that provides ability to parse permutations

10. Add an extension to the library that allows easy parsing of indentation-sensitive grammars

11. Write an example parser for a simplified version of Python to demonstrate the ability to parse indentation-sensitive languages

2.4 Design principles

We will examine the strengths of parser combinators compared to other parsing solutions. We will also discuss how this will inform our design decisions.

Parser generators are tools that create a parser from a formal description of a grammar. The input of a parser generator is usually a grammar file. The output is the source code of a parser, written in the language targeted by the given parser generator [6].

Parser combinator libraries have the advantage of a simple setup compared to parser generators. Another advantage is that parsers are easy to extend. New parser combinators are also easy to implement.

In contrast, parser generators commonly have a performance advantage. Parsers created by parser generators are usually bottom-up, in contrast to our top-down parsers, and are generally faster than parsers written in the parser combinator style.

We should primarily focus on our strengths when designing a library. Thus we will focus on ease of use and extendability instead of optimizing for speed in our library. We will still want our library to be acceptably fast, but it will not be the main priority.

2.5 Core library

Let us now explain the ways we will achieve the stated goals. Our goals and their solutions will give us the design of the core of our library.

2.5.1 Parsec parsers and combinators

The first goal (1) is for our library to implement all basic Parsec parsers and combinators. The implementation of this goal is straightforward once we have the design of the primitives that we will use in our library. These primitives

include the parser type, the input our parsers will read from, and the result that parsers will return.

In considering the design of the parsing primitives, we will also accomplish goals (4) and (5) and also touch upon goals (3) and (6).

Parser type

There are two possible ways to implement the parser type. Since we are modeling parsers as functions, we can use the C# types that represent functions. These types are called delegates. Alternatively, we can define the parser as a class.

Delegates represent references to methods. They are defined by the method's parameter list and return type. We can thus assign to them any method with compatible signatures [25].

Delegates have the advantage over classes in that they more closely model our representation of parsers. Another advantage is that we can write lambda expressions as a more concise way of creating delegates. Lambda expressions make constructing parsers easier.

The advantage of modeling parsers as classes is that we can define custom operators. Especially the bitwise or operator could be redefined to function as the choice combinator. Another benefit is that we can more easily store some state along with the parser, which could be used in a potential extension. For example, we could allow the user to pass some custom state along the parsers in the future.

Ultimately, we decided on modeling parsers as delegates as we found them more straightforward to work with. In our estimation, the benefits of the lambda expression outweigh the advantages of custom operators. Further, the user-defined state can be implemented by adding it, for example, to the input argument of the parser.

Parser Input

Next, how will we model input for the parsers? One of our stated goals (4) is to support reading from a file. This is a functionality that the existing libraries do not have (Sprache and Superpower) or have but is not wokring properly (ParsecSharp). Thus we want to rectify this situation.

Usually, we want to read directly from a file when its contents are too large. As such, we want the capability for the parser input to work without loading the entire file into memory.

Another fact to consider is that the try combinator allows for unlimited lookahead. If we want to keep lookahead entirely in memory, then we have to have some information external to the input of when lookahead begins and ends. Otherwise, we would need to keep the entire input that we have read so far

buffered in memory. Thus we would need to either have some way to inform the input of when to start *buffering* or allow for *seeking* in the input stream.

The benefit of buffering input is, of course, that any lookahead would occur entirely in memory. This will not only mean that it is faster. It will also mean we do not require the input stream to support seeking.

The downside is that we need external information about when lookahead begins and ends. As such, we are adding temporal dependency into our inputs. We need to call these notification functions for the lookahead to work. Otherwise, any read that tries to return an already processed symbol is invalid. The implementation of the input becomes overall more complicated and has a higher potential for bugs.

On the other hand, if we rely on seeking in the input stream, then the interface becomes simpler to work with and comprehend. The main drawback is that we require our input streams to be able to seek, which, for example, network streams cannot. Network messages are, however, ordinarily much smaller than files on disk. Thus it is not a problem to load the message entirely into memory.

A further goal (5) we should discuss is that of modeling the input as immutable. An immutable interface is easier to reason about. Further, in the case we want to add an additional state to the input, such as a user-defined state, then the advantages in reasoning about an immutable interface as opposed to a mutable one become even greater.

The decision to make the input immutable also affected our choice about whether to go with the seeking or buffering implementation of input. The seeking variant has a significant advantage in always behaving predictably, just as an immutable object should. The buffering variant would have exceptions to this behavior. This was the main reason that made us go with the seeking variant.

The last goal (3) we will consider is that of supporting parsing symbols other than characters. The impact on our implementation will be simple. We will make the input interface generic.

Unfortunately, making the input immutable and being able to read from files makes it very hard to implement a regular expression parser. The C# provided regular expressions require the entire string we are trying to match against as input. Therefore, in the case of file streams, we would have to read the entire input into memory, which goes against our design. If we want to support regular expression parsers, we need to create our own custom regular expression engine which supports reading input one character at a time. This is beyond the scope of this thesis, and we will thus not support regular expression parsers.

Result

The final primitive we would like to examine is the result of the parsing. The result will need to contain the product of the parse. It will also include the unconsumed input.

One issue to look at is whether the product should be able to be accessed partially or only safely. What we mean is that currently, the result will contain either the product of the parse or an error. As such, if we allow direct access to both members, then access to one of them, the one that is not defined, will be an invalid operation. The way around this is to block direct access and only allow the user to work with the result through a pair of functions. The result would invoke the first function when the result contains the product of the parser. And it would call the other function when the result contains an error.

While this is an attractive choice, as this would eliminate any potential errors, it would also make working with the result unwieldy in an imperative language such as C#. Also, the function invocation would be significantly slower than direct access. Since the result type will be in many performance-critical sections, it would considerably impact performance. As stated previously, the performance of our library is not a primary consideration. However, it is still necessary for our library to be acceptably fast. As such, we will still support the access pattern through functions, but direct access will also be possible.

The result type will also play a role in the goal of making parsers covariant (6). For the parser to be covariant, the result must also be covariant. The result can be covariant only if we can exclusively retrieve the result of the parse and not set it. We will thus make the result immutable.

2.5.2 Error system

The next goal (7) for our library is to create a better error system than that of other libraries. All the libraries we examined use a simple string as an error message. In the case of Superpower, there are two messages, one explaining what we expected and one notifying us about what we encountered.

We desire to have a more comprehensive and extensible error system. The first step in this process is recognizing what errors we can have. We see two kinds of errors possible.

The first is a *standard error* that is generated automatically by the parser. For example, when we expect the keyword `while` but instead receive the keyword `for`, the parser will fail on the first letter. The error should be constituted of the expected keyword, `while`, and the unexpected token encountered, `for` or just the letter `f`, depending on the granularity with which we are parsing. In case we are at a point of choice in the parse, then the set of expected constructs can be larger

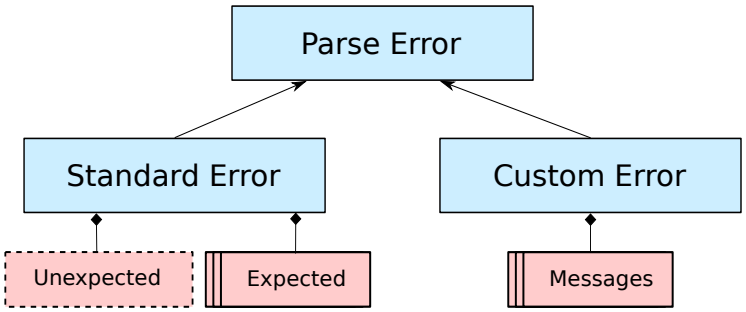

Figure 2.2 Structure of parse errors

than one. In such situations, we should also be able to aggregate errors so that we do not have repeated expected items, as well as to choose only the most relevant errors. As such, when combining two standard errors, we perform a union of the two expected sets. Also, when one error occurs later than the other, then we use the error that happens later as we presume that such an error is more specific. Finally, the items stored in the set of expected tokens and the unexpected token should be of an abstract type that is further specialized to suit the user's needs.

The second kind of error is a *custom error* that is given by the writer of the parser. It usually signifies uncommon situations that should have precedence. The errors could be simple messages or more complex structures containing contextual information about the error. As such, the custom error itself will be comprised of a single set of abstract messages that are further specialized. The combining of two custom errors follows the same structure as the combining of standard messages.

These two types of errors should suffice for all use cases that we predict a user would need. The vast majority of the needed customization should occur by specializing the error items located inside the errors.

2.6 Extensions

Let us now consider the two extensions we set as goals for our library. The first is an extension for parsing permutations (goal 9). The second adds the ability to parse indentation-sensitive grammars (goal 10).

2.6.1 Permutation

Let us first discuss the extension for parsing permutations. By parsing permutations, we mean the parsing of a sequence of elements in which each item appears precisely once, and the order is irrelevant. Additionally, some of the permutable elements may be optional. This can be used, for instance, in the parsing of XML tags or command-line options.

Summary of "Parsing permutation phrases"

Our solution is based on the paper "FUNCTIONAL PEARL Parsing permutation phrases" [26]. This paper showcases how to extend a parser combinator library with parsing permutations. Let us now quickly summarize the results of the paper. The basic idea is to gradually build a recursive data structure, a *permutation tree*, which holds all of the possible sequences of elements. When we want to create the final parser, we recursively create parsers from the permutation tree's different *branches*. Afterward, we combine all these parsers created from the branches in a choice combinator. In figure 2.3, we see one such permutation tree.

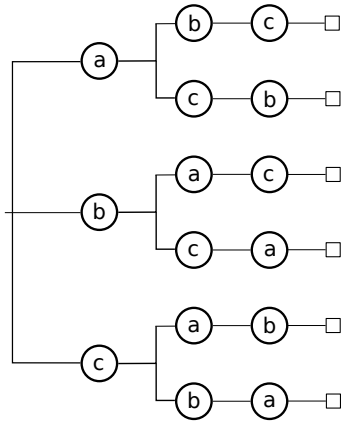

Figure 2.3 Permutation tree for parsers `a`, `b`, and `c` [26]

The permutation tree becomes more complicated when we introduce optional elements. If we merely add optional parsers to the permutation trees, then the final permutation parser becomes ambiguous. Therefore we need to differentiate between the addition of standard parsers and parsers that we want to be optional. Subsequently, we add early endings to the permutation trees in cases where all of the remaining parsers are optional. A permutation tree with parser `b` set as optional is in figure 2.4. When we add an optional parser to the permutation tree, we also provide a default value that should be used if the parser is not applied.

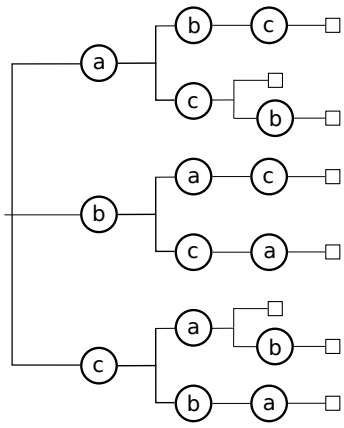

Figure 2.4 Permutation tree with optional parser `b` [26]

The permutation tree is also provided with a final transformation function. This function takes the results of the individual parsers and creates the result of the permutation parser.

Now we shall describe an example. Let parsers `a` and `c` parse integers and the parser `b` floating point numbers, or floats for short. First, when creating an empty permutation tree, we provide a function `add`, which adds an integer, a float, and another integer together. The result of this function is a float. Afterward, we add `a` as a standard parser, `b` as an optional parser with a default value of zero, and finally, `c` as a standard parser. The resulting permutation tree is the same as in figure 2.4. During this process of adding parsers, the type system checks that the result types of the added parsers are compatible with the types of arguments of the `add` function. Finally, we create the final permutation parser. It parses any permutation of `a`, `b`, and `c` and applies the function `add` to its results. The result of the parser is thus a floating point number.

Translation to C#

Let us now explore possible adaptations of the described algorithm into C#. Unfortunately, the straightforward translation of the algorithm to C# does not work. In Haskell, the function type `a -> d` can stand for the function type `a -> b -> c`. The function `b -> c` is substituted for the type variable `d`. Also, we can perform partial application on the functions. For example, we can pass only an argument of type `a` to a function of type `a -> b -> c`. The result is a function of type `b -> c`. The original algorithm uses these abilities of the Haskell type system in constructing the permutation tree. Sadly, we cannot easily imitate these features in the C# type system.

If we would want to preserve only adding one transformation function to the permutation tree, then we have two solutions. The first is creating many versions of the permutation tree data structure. Each version would have a different number of generic type parameters. The number of type parameters would be given by the number of arguments the final transformation function has. The second version would use reflection to determine the number of required arguments and generate the needed classes during runtime.

Both of these solutions are not ideal. The first solution requires us to write many different versions of the same data structure. It would also be necessary to have a source code generating script for generating additional permutation trees. This would be used in the case that the function our user wants to use has more parameters than are supported by default. The second solution does not have the issue of including an additional script. However, we forgo static type checking. Let us remind ourselves that one of our design principles is the ease of use. Both of these solutions, in our opinion, go severely against this principle.

Our solution, therefore, relaxes the requirement for a single transformation function. Instead, we will have a series of functions. Each function will be added to the permutation tree with a corresponding parser. Every function will have as the arguments the result of the paired parser and the result of the rest of the permutation.

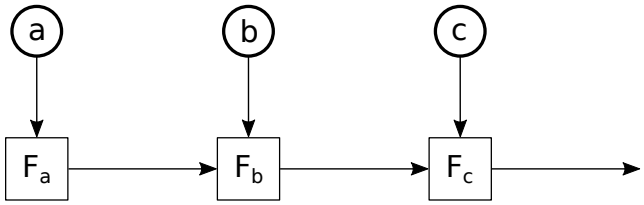

Figure 2.5 The sequence of functions in our permutation tree

The final algorithm is, in the end, similar to the one described in the original paper. The difference is that in each branch, we also remember a function that combines the result of the first parser in the branch and the result of the rest of the branch. When a permutation is parsed, we apply the separate combining functions in a sequence. The order of the applications is the same as the order in which we added the functions and parsers to the permutation trees. The final

function outputs the result of the entire parser. The sequence of applications is exemplified in figure 2.5.

Both optional and standard parsers are treated the same in our algorithm by default. It is only when all of the remaining parsers in the branch are optional that our behavior changes. Along with creating the sequences of function applications with the results of the parsing, we also create sequences of function applications with the provided default values. These function applications with a default can occur during the construction of the permutation tree itself, as the default values never change. If all of the remaining parsers are optional, then we also have the option to return the generated default value. We return this default value in the case that all of the subbranches fail to parse.

This solution is both type-safe and does not require generating any source code. The only hindrance is that along with adding parsers to a permutation tree, we also need to provide the corresponding functions. However, this is a minor inconvenience when compared to the previous solutions.

2.6.2 Indentation

Now we will examine the extension for parsing indentation-sensitive grammars. Several popular programming languages such as Haskell [5] and Python[27] use indentation as an integral part of their syntax. So far, parsing of indentation has no support in our library. Our plan is thus to provide combinators that handle the most common patterns used in indentation-sensitive parsing.

Our solution is based on the paper "Indentation-Sensitive Parsing for Parsec" [28]. The solution introduced in the paper is rooted in *relations* between *indetation levels*. More specifically, the relations are between a *parent* parser and their composite, or *child*, parsers The relations that are used are $=$, $>$, $\geq$. These behave as one would suspect; they require the indentation of a child parser to be equal, greater than, and greater or even than its parent parser, respectively. Additionally, there are operators $\circledast$ and $|p|$ where p is an arbitrary parser. The operator $\circledast$ disassociates the indentation level of a parent and child parser. In other words, it locally turns off indentation-sensitive parsing. The operator $|p|$ parses is used when we need to parse a sequence of statements, and they all need to have the same indentation. The paper shows that with these operators, we can implement the common usages of indentation in syntax. Concretely, it shows the implementation of indentation-sensitive blocks in Haskell and Python.

The solution in the paper annotates each parser with the desired relative relation to its parent parser. Afterward, the validity of the relation is checked during the reading of the input. Therefore, if we use this solution, we would need to implement a whole new set of parser inputs, as they would need to be able to check the relative indentation of the read symbol compared to some reference.

Instead, we choose to implement a solution based on inserting *indentation guards* as a new combinator. This combinator will be given a reference indentation level and a relation. Subsequently, it will answer whether the relation between the reference and current indentation levels holds. This guard serves us in place of the relation annotations of the paper. We will also introduce a parser that will output the current indentation level. These combinators can be defined without changing any part of the library outside the indentation module. The original paper mentions adding explicit guards as a possible solution. However, it dismisses them as susceptible to programmer error. To combat this weakness, we will supply convenient combinators that will take care of the most common patterns. Here, we were inspired by the Megaparsec[29] indentation module.

The most common pattern for indentation-sensitive parsing is when we have a *head item* and then a list of identically indented items. Usually, this head item and the subsequent list form a single statement. An example of this is the Python for loop. In fact, the operator $|p|$ is introduced in the paper to solve exactly this pattern. To implement this pattern, we only need to analyze what different indention levels during parsing signify.

```
1 for i in range(0, 10): # The head item
2     j = i * i # List of items -
3     print(j)  # each has the same indentation
```

Let us imagine that we already know the indentation level of the head item and the first item. We are about to parse the next item and check its indentation level. If the level is the same as that of the first item, then there is no problem, and we proceed. If the level is less or equal to the indentation of the head item, then we have reached the end of the list. The subsequent item will be the next statement. We thus stop parsing and return the parsed head item and list. If the level is anything else, then we have an indentation error.

```
1 for i in range(0, 10):
2     j = i * i
3     foo() # Correct indentation
4       bar() # Wrong indentation -
5             # more indented than it should be
6   foobar() # Wrong indentation -
7            # less indented than it should be
8
9 another_statement() # Another statement
```

The indentation level of the head and the first item is simple to acquire. The only thing we have to check is that the indentation level of the first item is greater than that of the head item. If it is not, then we either reject the parse if we require at least one item in the list or we return the head item and an empty list.

Another often-used pattern is when we want a statement not to be indented. In other words, we want it to begin on the first column of a line. This is a trivial use of the indentation guard. We only require the indentation level to be equal to that of the first position in a line.

The last pattern we will discuss is that of *line-folding*. This is useful when a statement can be optionally spread across multiple lines. An example of this is multi-line headers in the HTTP 1.1 protocol [30]. During line-folding, the subsequent lines should begin further than the first line. As such, we insert an indentation guard checking that this relation holds to the beginning of every line.

Using these three combinators, we can implement common indentation-sensitive grammars without difficulty. We can also define new combinators if ever the need arises. We will show our ability to parse indentation-sensitive grammars in solving goal (11).

Chapter 3

Implementation

Like most parser combinator libraries, our library is heavily inspired by Parsec. We saw it thus fitting to name our library *ParsecCore*. The second part of the name comes from the fact that our library is written for the framework .NET This framework was formerly known as .NET Core. We will be using ParsecCore to refer to our library going forward.

This chapter will explain relevant details useful for a programmer looking to expand our library. We shall explain the different parts of ParsecCore going inside out. We start with the primitives mentioned in the previous chapter (see section 2.5.1). Including the very core of the library, the `Parser` delegate itself. And end with a description of the extension modules.

We will only describe the core types and any interesting or unusual aspects of the library. The description of other types present in the library, as well as implementation details, can be found in the documentation comments of the source code.

In this chapter, we will only discuss the ParsecCore library itself. Any other projects in the solution are described in appendix A.

3.1 Parser input

Every parser input must implement the interface `IParserInput<T>`. The interface is designed so that the inputs behave as immutable, which is also expected of any class implementing the interface. This interface implements the `IEquatable` interface. Equality tests are used to determine whether any input has been consumed.

```
1 public interface IParserInput<T>
2     : IEquatable<IParserInput<T>>
3 {
4     public IParserInput<T> Advance();
5     public T Current();
6     public bool EndOfInput { get; }
7     public Position Position { get; }
8 }
```

There are three different kinds of parser inputs implemented in the library. `StringParserInput` is used when the input for the parser is a string. `TokenListParserInput` is employed when the input for the parser is a list of tokens. It is a generic type that is specialized by the type of the input tokens. Finally, `StreamParserInput` is the parser input used when the characters are read directly from a stream, such as an open file.

As we can see in the IParserInput interface, the inputs should keep track of what position in the input they are. The classes `StringParserInput` and `StreamParserInput` have a default implementation of how to keep track of the position. We increase the line number whenever we encounter the line-feed character. Otherwise, we increase the column number. All three input types support defining custom functions that control the updating of the position in the input. We encourage any user-defined inputs to follow the same pattern.

3.1.1 StreamParserInput and Buffer

The implementations of both `TokenListParserInput` and `StringParserInput` are straightforward. On the other hand, The `StreamParserInput` is more complicated and deserves a deeper examination.

We employ seeking in the input stream as stated in section 2.5.1. Seeking is used when lookahead is performed. In most situations, however, lookahead is only a few dozen characters. Further, when lookahead occurs, it is probable that more lookahead will start at the same position. It would be wasteful to read the same characters multiple times from the stream. To reduce the number of times we read from a stream, we implement buffering above the input stream.

Buffering is implemented, appropriately enough, in the `Buffer` object. It is this object that serves characters to the `StreamParserInput`. When a call for a given position comes, it looks up whether it has this character buffered. If it does, then the read occurs entirely in memory. If it does not, then it buffers a new continuous block of data, starting with the requested position. This occurs in preparation that more lookahead will occur starting from this position.

3.2 Result

The interface `IResult<out T, TInput>` defines the result type. It is barebones and only allows property access to the composite data. The rest is implemented using extension methods. The implementations in the extension methods are no more complicated nor significantly slower than if the methods were part of the interface. The advantage of extension methods is that in the case a new result type is to be implemented, then only the properties need to be defined. The rest of the methods come automatically.

As an aside, when referring to the *parse result*, or simply *result*, we mean this interface. We will refer to the structured output of the parsing, the actual object the user will work with afterward, as *output*.

The `IResult` interface is implemented in two separate classes. One represents a successful parse. The other represents a failed parse.

3.3 Parser

Let us now describe the `Parser` type. It is a straightforward delegate type. However, as it is at the core of our library, we think it is pertinent to go over the type in detail.

```
1 delegate IResult<T, TInput> Parser<out T, TInput>
    (IParserInput<TInput> input);
```

As we can see, `Parser` is a function that takes the `IParserInput` as input and outputs the `IResult` object. The result contains the output of the parse or an error. It also contains the input that has yet to be consumed. We use this unconsumed input to sequence parsers after each other. We can also use a previous input The use of an earlier input is lookahead.

We can also notice that the parser is covariant with respect to its parsing output. We have thus fulfilled the goal (6) for our parser to be covariant. We discussed covariance in section 2.2.2. This is one of the reasons the result is an interface, as only interfaces and delegates can be covariant. Otherwise, the result could be implemented as a `struct` which would be faster.

3.4 Errors

The errors follow the general structure introduced in section 2.5.2. An interesting aspect regarding errors is how the combining of errors is implemented. We cannot easily determine which error is more specific when two errors occur at the same

position. We decided in section 2.5.2 that we would prefer custom errors over standard errors.

But how do we implement this priority system? We could use integers as a way to determine priority. Then the object with the higher number would be preferred. However, if two errors of the same type meet, we must combine their messages. This would require casting the errors into their concrete type. In conclusion, this solution seems brittle, and we would want a better one.

Our solution uses the visitor pattern and the associated double dispatch to deal with this problem. We make the `ParseError` a visitor on itself. The first call to `Accept` determines the type of the first error through dynamic dispatch. Next, the call to `Visit` ascertains the type of the second error. The implementation of the combining of the error is subsequently simple. The visitor pattern has a weakness in that it is laborious to add new types. But as we mentioned in section 2.5.2, we do not expect that new types of errors will need to be implemented.

```
1 abstract partial class ParseError :
     IParseErrorVisitor<ParseError>
2 {
3     abstract T Accept<T>(IParseErrorVisitor<T> visitor);
4
5     // ParseError is its own visitor
6     // Used for merging two ParseErrors together
7     abstract ParseError Visit(StandardError error);
8     abstract ParseError Visit(CustomError error);
9 }
```

We also need to compare `ErrorItem` objects. We do this to combine two messages about the unexpected symbol encountered. However, in contrast to the combining of `ParseError` we simply use an overridden property `Length`. We expect that more ErrorItem types will be added, and the overriding of methods and properties is more extendable in such a scenario. The length of the unexpected symbols is also a good indication of specificity.

3.4.1 Error erasure

One issue we have encountered while implementing the error system is what we will call *error erasure*. Let us imagine we have a parser that requires lookahead. Such a parser will seem as if it does not consume any input when it fails, just as we want. However, when we further apply the `Optional` combinator onto this parser, then an unfortunate situation occurs. The `Optional` combinator succeeds even if the underlying parser fails as long as it does not consume any input. Therefore, when we encounter an error, we first backtrack and make it seem like no input was consumed. Afterward, the `Optional` combinator turns

the failed parse into a successful one. And during this transformation, we lose all information on any previous errors. This is what we call error erasure.

This does not matter if the parse is successful. But if the parsing fails, then there may have been errors that occurred that were more specific than the ones we ultimately reported. Error erasure can cause the announced errors to be wildly different from the actual error that caused the parsing failure. This is an issue that all of the surveyed C# libraries (see 2.2) have.

The way to solve this problem is to remember the most specific error in the case of a successful result. Just as we remember it in the case of a failed parse. Therefore, we change the `IResult.Error` such that a successful parse does not report an invalid operation but returns the most specific error. Also, whenever we construct a parsing result from the results of two or more parsers, we have to combine their errors as well.

3.5 Parsers and combinators

We shall now describe the included parsers and combinators located in the library. We will not give an exhaustive list as the number of parsers, and combinators is far too high and most are simple. We will only describe the most important ones.

3.5.1 Satisfy

The most crucial parser in our library is the *satisfy* parser. This is our version of the parser of the same name mentioned in section 1.4.1. It is the only parser that reads from the parser input. The way this parser works with the input is significant. The input is advanced to the next position only when the predicate succeeds. If this were not the case, then we would not have even a single token of lookahead.

The satisfy parser is implemented four different times. There are specialized versions for characters to create better messages. Otherwise, when encountering an unexpected line feed, the error message would print the actual line feed and not '\n'. Then there are versions made for the particular case where the predicate is a simple comparison with a single token. These versions are included to improve performance.

3.5.2 Imperative implementations

One aspect that is present in many of the parser implementations is that most are written imperatively. Instead of the recursive function calls that are used in Parsec, we mostly use loops. We do this for two reasons. The first reason

is that these implementations have better performance. The second and main reason is that C# does not reliably produce tail call optimizations. Therefore, in a combinator such as `Many`, a stack overflow error would occur with large enough input sizes.

A stack overflow error can still occur if we are parsing complex, highly recursive data. However, the complexity of the data does not generally increase with the length of the data. For this reason, in practical applications, we do not encounter this error.

3.6 Permutation

One possible issue with the implementation of the permutation parser is that the number of composite parsers grows exponentially. This is because our strategy for creating permutation parsers — as explained in section 2.6.1 — is to create the entire permutation tree.

Unfortunately, the memory usage is higher than in the functional version of this extension. When we create the `PermutationParser`, the recursive structure stores all possible combinations. However, most of this is done by creating new lists of branches that contain only references to already existing objects. Whenever we add to a `PermutationParser`, the number of newly created objects is linearly proportional to the number of parsers already contained in the `PermutationParser`. The number of objects is, therefore, quadratic and not exponential with regard to the number of parsers in the permutation tree.

Also, the actual parsing is lazily evaluated. This means that only the paths that we explore are materialized. The reason for this is that any parser that is used inside the LINQ query syntax is evaluated only at the moment it is needed. And the query for creating the permutation parser looks like this.

```
1 public Parser<Final, TInput> GetParser()
2 {
3     return from first in _parser
4     from rest in _permutation.GetParser()
5     select _combine(rest, first);
6 }
```

Accordingly, a specific branch of the permutation tree is fully evaluated only when we explore it during parsing. We, therefore, avoid the exponential memory usage.

3.7 Creating new parsers

Lastly, we want to give some advice on writing custom parsers and combinators. There are two ways to create new parsers and combinators. The first is to combine existing parsers using the combinators provided by the library. This approach is described in the next chapter 4. The second method is to implement a new delegate that is compatible with the type signature of `Parser`. In the following section, we will discuss this second approach.

A lambda expression is the most comfortable way to write a new parser. If we decide to write the parsers and combinators in this manner, then there are a few pieces of advice we would like to mention.

The first is to write the parsers primarily in an imperative style. The resulting parser will be more performant. Of course, writing the parser in an imperative style would sometimes be exceedingly complicated. In such situations, it is preferred to write them declaratively using the query syntax.

The next piece of advice is to precompute values that are not dependent on the runtime result of the parsing. These values can be computed only once, and this result is captured in the lambda expression. This synergizes with the guideline of initializing a parser only once mentioned in the documentation (see 4.3.1).

Another advice is not to use lookahead unless it is an essential part of the parser. Lookahead can be an expensive operation; the more we avoid it, the better. Moreover, the user can usually add lookahead as part of the arguments it passes to combinators. Thus, if we add lookahead, we are forcing the user instead leaving it as a choice.

Second to last, all of the performance advice mentioned in chapter 4 apply even more if the parsers are reused often. In short, try to use the query syntax as little as possible. Order parsers in the `Choice` and `Or` combinators according to their likeliness, from most probable to least. And lastly, try to instantiate each parser only once, especially avoid creating new parsers in queries.

Finally, remember to combine errors of all parses, especially the successful ones. Otherwise, there is a chance for error erasure. Use the appropriate overloads of the `Result.Success` and `Result.Failure` methods. Their usage is described in the documentation comments.

Chapter 4

User documentation

This chapter explains how to use the ParsecCore library. We will mention general guidelines and recommendations to get the most out of our library. We will explain how the library works through examples. Unless explicitly stated, all mentioned classes are located in the `ParsecCore` namespace.

4.1 Core library

The core library provides the ability to create parsers using parser combinators. It also gives us a base upon which other extensions can build. Unless stated otherwise, the parsers and parser combinators provided in this section are located in the `Parsers` class.

4.1.1 Creating parser input

Before we start the actual parsing, we need to create the parser input. We can create parser inputs from a string, a list of tokens, and a stream that allows seeking. All of these are created through the `ParserInput.Create` methods. These methods have all of the relevant overloads for creating the parser input.

The `updatePosition` parameter defines how the parser input keeps track of its position. It is called whenever an input token is read and determines, based on the token, what the next position should be. If we were reading characters, then an implementation could look like the following example. The `tabSize` is a captured value.

```
(readChar, position) =>
{
    return readChar switch
    {
        '\n' => position.WithNewLine(),
        '\t' => position.WithTab(tabSize),
        _ => position.WithIncreasedColumn()
    };
};
```

The parser inputs for strings and character streams have a default implementation of `updatePosition`. It is equivalent to the example above. For lists of other tokens, the `updatePosition` function has to be supplied manually. As this function is called whenever we advance in the parser input, we recommend not including any complex expressions or statements.

4.1.2 Simple parsers

The simplest parsers contained in ParsecCore are parsers for a single symbol. The two most common are the parsers `Satisfy` and `Char`. Both of their Parsec equivalents, which have similar semantics, are introduced in section 1.4.1. The simplest example we can show is the parsing of a single character.

```
IParserInput<char> input = ParserInput.Create("abc");
Parser<char, char> parser = Parsers.Char('a');

IResult<char, char> result = parser(input);
```

This example also displays a general outline of working with ParsecCore. We create the parser input. Then we prepare the parser with which we will work. Finally, we attempt to parse the input with the prepared parser. In this case, the parser succeeds, and its result contains the *output* of the parse, the character `'a'`.

The delegate type `Parser` and the interface `IResult` have two generic type arguments. The first argument is the output type of the parse. The second argument is the type of the symbols that serve as our input.

Another important primitive parser is the `EOF` parser. This parser succeeds if we are at the end of a file. Otherwise, it fails. We use this parser to check that the entire parser input has been processed.

4.1.3 The result of parsing

Every parse attempt creates an `IResult` object. This object can have two forms, either it can signify a successful parse, or it can represent a failed parse. We can ask the `IResult` object which outcome it embodies. The property `IsResult`

answers whether the parse was successful, and the property `IsError` tells us if the parse was a failure. Depending on which type of result it is, we can then access either `Result` or `Error`.

```
Parser<char, char> parser = Parsers.Char('a');
IParserInput<char> input = ParserInput.Create("abc");

IResult<char, char> result = parser(input);
if (result.IsResult) // Same as !result.IsError
{
    Console.WriteLine("Great, the parse was successful!");
    Console.WriteLine(result.Result);
}
else
{
    Console.Writeline("An error occured :(");
    Console.WriteLine(result.Error);
}
```

In both cases, the `IResult` object contains the remaining output that has not been processed yet. In this instance, it is equivalent to the string `"bc"`. We can access this output through the `UnconsumedInput` property.

4.1.4 Sequencing parsers

The ability to parse a single character is useful. However, in most cases, we need quite a bit more. So far, we have parsed only a single character at a time. Now, we want to parse a string of characters.

The first way we might think of to parse a string is by chaining calls to parsers together. Imagine we want to parse the string `"abc"`. With the stated method, we might come up with something like this.

```
var input = ParserInput.Create("abc");

var resultA = Parsers.Char('a')(input);
if (resultA.IsError)
{
    return resultA.Error.ToString();
}
var resultB = Parsers.Char('b')(resultA.UnconsumedInput);
if (resultB.IsError)
{
    return resultB.Error.ToString();
}
// Continued on the next page
```

```
14 var resultC = Parsers.Char('c')(resultB.UnconsumedInput);
15 if (resultC.IsError)
16 {
17     return resultC.Error.ToString();
18 }
19 return "abc";
```

We chain if statements together until all parsers have succeeded or one of them fails. This solution is verbose, inflexible, and prone to errors. Thankfully, we have a solution.

Parser LINQ

The ParsecCore implements the `Select`, `SelectMany`, and `Where` extension methods and thus allows the use of LINQ query syntax for parsers. For a quick explanation of LINQ, see section 1.5. The above example can be rewritten into a LINQ query in the following way.

```
1  var input = ParserInput.Create("abc");
2  // Definition of a new parser
3  Parser<string, char> parserABC =
4      from a in Parsers.Char('a') // Parse character 'a'
5      from b in Parsers.Char('b') // Parse character 'b'
6      from c in Parsers.Char('c') // Parse character 'c'
7      select "abc"; // if parses successful, return "abc"
8
9  var result = parserABC(input);
10 if (result.IsResult)
11 {
12     return result.Result;
13 }
14 return result.Error.ToString();
```

It is important to note that, in this example, we are creating a new parser. A parser query can take multiple parsers and combine them into a new one. In this example, we are combining the parsers for characters `'a'`, `'b'`, and `'c'`. If they are all successful, the newly created parser outputs the string `"abc"`. Otherwise, the new parser returns the error of the first parser that failed. The query syntax for parsers has four possible clauses. Here, we use the two most important.

The clause beginning with `from` represents an attempt to parse the input. From now, we will refer to this clause as the *from* clause. It has two parts. The first is the variable name after `from`. This variable stores the output of the parse if it is successful. It can be used further in the query. The second part is the parser located after `in`. This section defines with which parser we will attempt to parse.

The second clause we use is the *select* clause. This clause tells the parser what to return if all the previous parses are successful. Every parser query has to have

precisely one select clause, and this clause has to be the last one in the query.

As we mentioned before, we can use the output of a previous parse in the following clauses. Say we want to parse a sequence of three of the same letter — the strings `"aaa"` or `"ZZZ"`, for instance. The parser for a single letter is `Parsers.Letter`. We can thus define the parser for three letters in this manner.

```
Parser<string, char> threeLetterParser =
    from letter1 in Parsers.Letter
    from letter2 in Parsers.Char(letter1)
    from letter3 in Parsers.Char(letter1)
    select new string(letter1, 3);
```

The remaining two clauses we can use with parsers are the *let* clause and the *where* clause. The let clause allows us to define additional variables. If we reuse a piece of code in the query, we should to introduce a let clause to reduce the verbosity and unify the used values. The where clause is used to check assumptions. We provide a predicate to test; if the predicate fails, the entire created parser fails.

```
Parser<char, char> parser =
    from c in Parsers.Letter
    where c != 'z' && c != 'Z'
    let laterChar = c + 1
    from _ in Parsers.Char(laterChar)
    select laterChar;
```

However, we recommend the `Assert` parser combinator instead of the where clause. The where clause upon failure supplies only a generic message. The `Assert` combinator is much more customizable.

Then and FollowedBy

The query syntax is a helpful tool. However, the query implementation includes the creation of many lambda functions. This is something that can significantly decrease performance. As a piece of general advice for increased performance, try to avoid the query syntax unless necessary. For this reason, ParsecCore also includes methods that more efficiently implement some common patterns of combining parsers.

First is the `Then` method. This method applies the first parser, discards its result, applies the second parser, and returns its result. If any of the two parsers fail, the parser created by the `Then` method also fails. The two parsers in the next example are equivalent.

```
Parser<char, char> thenParser =
    Parsers.Char('a').Then(Parsers.Char('b'));
Parser<char, char> queryParser =
    from _ in Parsers.Char('a')
    from b in Parsers.Char('b')
    select b;
```

The second is the `FollowedBy` method. This method is very similar to the `Then` method. The difference is that in this method, we discard the result of the *second* parser and return the result of the *first*. The two parsers in the following example are again equivalent.

```
Parser<char, char> thenParser =
    Parsers.Char('a').FollowedBy(Parsers.Char('b'));
Parser<char, char> queryParser =
    from a in Parsers.Char('a')
    from _ in Parsers.Char('b')
    select a;
```

All and String

We looked at one way of sequencing parsers. Sometimes, however, we have a variable number of parsers to sequence. Alternatively, sequencing these parsers in a query would be a chore. In such an instance, we should use the `All` parser combinator.

The `All` parser combinator takes an enumerable of parsers and sequences them all, one after another. If any of the component parsers fail, then the entire parser fails. A list of their results is returned upon successful parsing of all constituent parsers.

A particular case of the `All` combinator is the `String` combinator. `String`, as we can deduce from its name, parses a given string, a sequence of characters. This combinator is one of the most used. The parser of the string `"abc"`, the first example we used query syntax with, can thus be written in the following way.

```
Parser<string, char> parserABC = Parsers.String("abc");
```

4.1.5 Choice

Often, multiple different values are possible to parse. For example, when parsing a boolean literal, there are two possibilities. Either it is a string `true` or the string `false`.

We use the parser combinator `Or` in such situations. This combinator composes two parsers. It tries to parse the first parser. If it succeeds, then it returns its result. Otherwise, if it has not consumed any input, the second parser is tried, and its

result is returned. The issues around consuming input will be explained in the next section (see 4.1.6). If both errors fail, then their errors are combined. This process of combining errors will also be explained in a later section (see 4.1.7).

If we return to our example of parsing the boolean literals, the parser could look like this.

```
Parser<bool, char> parserTrue =
    from t in Parsers.String("true")
    select true;
Parser<bool, char> parserTrue =
    from f in Parsers.String("false")
    select false;
Parser<bool, char> parserBool = parserTrue.Or(parserFalse);
```

When we want to decide between many such choices, we have two options. We could chain many `Or` combinators together. This, however, is needlessly verbose. The other option is to use the `Choice` combinator. This combinator takes a sequence of parsers, and the created parser returns the result of the first successful. The following two parsers are semantically equivalent.

```
// Definition of parsers a, b, and c

var parserOr = a.Or(b).Or(c);
var parserChoice = Parsers.Choice(a, b, c);
```

Since we are trying the parsers in a fixed sequence, we recommend putting the most likely option first. In some instances, it can significantly speed up parsing.

4.1.6 Lookahead

Some grammars require us to look many symbols ahead when choosing which alternative to proceed with. Looking forward by multiple symbols at a time is called *lookahead*. However, this can be an expensive operation. Thus ParsecCore, by default, only looks forward by one character.

As stated previously (section 4.1.5), the `Or` combinator fails if the first parser does not succeed. This is consistent with ParsecCore only looking forward by one character as the character parsers, `Char` and `Satisfy`, only consume input if they succeed. However, we can change this behavior. The `Try` combinator makes it so that if a parser fails, it does not consume any input. In other words, the combinator *backtracks* in the parser input to where it was located at the start of the parse. Afterward, we can use the resulting parser in the `Or` combinator. Thus, we can choose which parser to go with based on more than one symbol.

Let us show how the choice and lookahead combinators work together. Imagine we are parsing a language that has both `for` and `foreach` statements. If we

did not use the lookahead, `Try`, combinator, then we would not be able to parse some of the possible parser inputs correctly.

```
IParserInput<char> input = Input.Create("foreach");

var parserFor = Parsers.String("for");
var parserForeach = Parsers.String("foreach");
// We first try to parse "for", then "foreach"
var parserForStmt = parserFor.Or(parserForeach);
// The parsing fails because we first try "for",
// this fails but consumes input,
// therefore the entire parser fails
var result = parserForStmt(input);
```

We need to introduce lookahead, in the form of the `Try` combinator, to look forward and decide which parser to choose.

```
// Definitions of input, parserFor, parserForEach

// Notice the 'Try()'
var parserForStmt = parserFor.Try().Or(parserForeach);
// Parsing succeeds and outputs "foreach"
var result = parserForStmt(input);
```

There are more versions of lookahead beside `Try`. The others are `LookAhead` and `NotFollowedBy`. `LookAhead` backtracks in case the parser succeeds. `NotFollowedBy` backtracks and reverses the result to a failure in the case the parser succeeds.

By default, if a parser fails while consuming input, then whichever combinator it is contained within fails as well.

4.1.7 Errors

We shall now discuss the error system of ParsecCore. When a parser fails, then in most situations, it means that the current symbol is different than we expected. A default *standard error* is generated in such a scenario. This error contains the unexpected symbol we have encountered and a message explaining the symbol we expected instead.

Another type of error is the *custom error*. It represents situations other than the parser encountering an unexpected character.

When multiple errors are generated, such as when all parsers in the `Choice` combinator fail, they are combined. They are combined based on which error is more specific. An error is said to be more specific when it occurrs later in the parser input. A secondary consideration is the input type, with the custom errors treated as more specific than the standard errors.

We, as users, can change the error handling somewhat. First, we can label complex parsers with a more apt description. This influences the message explaining what the parser expected. The labeling is done by the `FailWith` combinator.

```
1  var parserTrue =
2      from t in Parsers.String("true")
3      select true;
4  var parserTrue =
5      from f in Parsers.String("false")
6      select false;
7  // In the case of error the parser will output
8  // "boolean literal" as the expected value
9  // instead of a single character
10 var parserBool = parserTrue.Or(parserFalse)
11     .FailWith("boolean literal");
```

Second, we can also create a parser that outputs a particular error. This is done by the `ParseError` and `Fail` parsers.

4.1.8 Maybe

`Maybe` is a struct that is the output of some of the parsers. It is a wrapper above a value that represents its possible nonexistence. The struct can be in two states, either empty or containing a value.

We can ask about its state using the properties `IsEmpty` and `HasValue`. If it contains a value, we can access it using the `Value` field or various extension methods such as `Else`, `Map`, and `Match`.

The `Maybe` value is created using two functions. These are `Maybe.Nothing` and `Maybe.FromValue`.

4.1.9 Other combinators

We would like to quickly mention other combinators. We will not go into any great detail. Nevertheless, we think it is useful for the user to know about different possibilities. The details can be looked up later on.

First, we will mention the optional combinators `Optional` and `Option`. These combinators successfully parse text, or they provide a default value. Next, the chain combinators, such as `ChainL`, are convenient when parsing a left recursive grammar. Afterward, we would like to mention the repetition combinators such as `Many`. This combinator applies a single parser as many times as possible and returns a list of all the results. We want to mention the many kinds of separator combinators which are similar to repetition combinators. However, between each repetition, a separator parser is applied as well. These combinators are helpful when we are parsing lists.

Indirect

Lastly, we would like to explain the `Indirect` parser combinator in detail. The indirect parser combinator is used when there is a circular dependency between parsers. For example, arithmetic grammars often have the following structure.

$$E \rightarrow\ E + E \mid E - E \mid F$$
$$F \rightarrow\ F * F \mid F/F \mid (E) \mid int$$

It is the (E) production that we are concerned with. In C#, most ways we can initialize our parsers are sequential. What are we supposed to do when two parsers are circularly dependent on each other? We use the `Indirect` combinator.

The idea is fairly simple. We wrap the first parser in a function, usually a lambda expression, and when it is time to use the parser, we invoke this function. We capture the value of the second parser in this function. Thus it is only when this function is called that the value of this parser is bound, and by that time, the second parser should be initialized. And so we have solved the circular dependency. The above grammar translated into ParsecCore would look similar to this code snippet (Please ignore that we would need to deal with the left recursion somehow).

```
1  var parserExpr = Parsers.Choice(
2      addParser,
3      subParser,
4      // Notice this parser and how we
5      // wrap parserFactor inside a lambda expression
6      Parsers.Indirect(() => parserFactor)
7  );
8
9  var parserFactor = Parsers.Choice(
10     multiplyParser,
11     divideParser,
12     Parenthesized(parserExpr),
13     integerParser
14 );
```

4.2 Extensions

We will now turn to extensions that the library provides. These tackle specific areas of parsing and make these areas more user-friendly. For example, creating a parser of a particular permutation is possible. However, we will supply a module that simplifies implementing such a parser.

4.2.1 Expressions

One type of parser that is created relatively often is a parser for arithmetic expressions. The *Expressions* module, located in the `ParsecCore.Expressions` namespace, helps with writing this parser.

The Expressions module supports binary infix operators — such as multiplication or division — as well as prefix and postfix unary operators. Operators are defined by one of three classes `PrefixUnary`, `PostfixUnary`, and `InfixBinary`. We, as users, supply them with parsers that parse the given operator and output a function transforming the operands. Additionally, the binary operators can decide their associativity.

The expression parser is created by supplying a two-dimensional array of operators. The order of the arrays defines the priorities of the operators. The earlier rows of operators have a higher priority than the later ones. Along with the operators, we must also give a parser for the basic term. This is the parser for integer literals, for instance.

The final arithmetic parser also detects whether the specified grammar is ambiguous. This is, however, done only during runtime when parsing expressions. During the parse, we detect if the operators are used ambiguously and return an error in such a case.

The arithmetic parser is created using the `Expression.Build` static method. This function takes the two-dimensional array directly, or an `OperatorTable` can be defined that acts as a wrapper around this array.

As an example, we will show the definition of an arithmetic parser that includes the standard operators of a unary plus and minus, addition, subtraction, multiplication, and division. This parser will work only with integers. The provided example also evaluates the expression as it parses it. Remaking the parser to provide only an abstract syntax tree of the expression is fairly trivial.

```
1  var operatorTable = OperatorTable<int, char>.Create(
2      new Operator<int, char>[][]
3  {
4      new Operator<int, char>[]
5      { // Highest priority - unary operators
6          Expression.PrefixOperator<int>("+", x => x),
7          Expression.PrefixOperator<int>("-", x => -x)
8      },
9      new Operator<int, char>[]
10     { // Higher priority binary operators
11         Expression.BinaryOperator<int>(
12             "*", (x, y) => x * y, Associativity.Left
13         ),
14         // Continued on the next page
```

```
15          Expression.BinaryOperator<int>(
16              "/", (x, y) => x / y, Associativity.Left
17          )
18      },
19      new Operator<int, char>[]
20      {  // Lower priority binary operators
21          Expression.BinaryOperator<int>(
22              "+",(x, y) => x + y, Associativity.Left
23          ),
24          Expression.BinaryOperator<int>(
25              "-",(x, y) => x - y, Associativity.Left
26          )
27      }
28  }
29  );
30  // Parser for natural numbers -- negative numbers
31  // are handled with the unary operator '-'
32  // We could also add parenthesized expressions
33  // using the Indirect combinator
34  var termParser = Parsers.Natural.FollowedBy(Parsers.Spaces);
35
36  var expressionParser =
37      Expression.Build(operatorTable, termParser);
```

Going further, we need only the `expressionParser` to parse any arithmetic expression. The static methods `PrefixOperator` and `BinaryOperator` help with a quicker definition of operators. These functions parse the provided string and any whitespace after them as the operator and return the given lambda function.

4.2.2 Permutations

A permutation phrase is a sequence of elements where each element occurs exactly once, and the order is irrelevant. Such phrases describe, for instance, XML tags or command-line arguments. Additionally, some of the elements can be optional.

We will call the parsers of such phrases *permutation parsers*. We will create permutation parsers using the `PermutationParser` class. We can think of this class as a builder class for the final permutation parser. New `PermutationParser` are created using the `Permutation.NewPermutation` and `Permutation.NewPermutationOptional` static methods. These methods define the first element of the permutation to add.

More elements are added using the `Add` and `AddOptional` methods. When adding new parsers, we not only add the parser itself but also add a function that tells the permutation parser how to combine the parsed element with the result of

the rest of the permutation. From now on, we will call this function the *combining function*. When a phrase is parsed, the combining functions are called one after the other, accumulating the parsing results. We also need to provide a default value when we add parsers for optional elements. The combining functions will use these values if the element is not present. Finally, we create the permutation parser using the `GetParser` method.

Let us present an example of how to create a permutation parser. We will show how to create a permutation parser for the elements 'a', 'b', 'c', and 'd'. The final parser will therefore accept strings "abcd", "abdc", "acbd", and so on.

```
1  var permutationParser =
2      Permutation.NewPermutation(Parsers.Char('a'))
3      .Add( // Add parser for element 'b'
4          Parsers.Char('b'),
5          // Create a tuple pair from 'a' and 'b'
6          (a, b) => (a, b)
7      ).Add( // Add parser for element 'c'
8          Parsers.Char('c'),
9          // Create a triple from
10         // the previously created pair and 'c'
11         (pair, c) => (pair.Item1, pair.Item2, c)
12     ).Add( // Add parser for element 'd'
13     Parsers.Char('d'),
14     // Create a quadruble
15     (triple, d) =>
16         (triple.Item1, triple.Item2, triple.Item3, d)
17     ).GetParser();
```

We can see the combining functions accumulating the values of the parse. In this case, we are creating larger and larger tuples.

4.2.3 Indentation

Some programming languages make use of indentation as part of their syntax. Examples of such languages are Python and Haskell. The namespace `ParsecCore.Indentation` contains combinators that make parsing common indentation-sensitive constructs much easier.

The parsers `Indentation.Level` and `Indentation.Guard` are the foundation of the module. Every other combinator in the module is constructed from these two parsers. The parser `Level` is simple; it returns the current level of indentation. The parser `Guard` accepts a relation — greater than, greater than or equal, or equal — and a reference indentation level. Afterward, it tells us whether the current level of indentation satisfies the given relation when compared to the reference indentation level.

With only these parsers, the parsing of a variety of indentation-sensitive grammars is possible. Nevertheless, these are relatively low-level parsers, and implementing a complex grammar would be cumbersome. Thus the Indentation module provides combinators for specific common patterns.

NonIndented

The first of these is the `Indentation.NonIndented` combinator. This is the simplest combinator. It only checks that the provided element parsed by the provided parser is not indented.

```
// The character 'a' has to be located
// at the beggining of a line
var parser = Indentation.NonIndented(
    Parsers.Spaces, Parsers.Char('a')
);
```

BlockMany

The next combinators we will discuss are the `Indentation.BlockMany` and the `Indentation.BlockMany1`. These combinators serve to parse a statement block that is defined by whitespace. Examples of such a statement block are Python statements. The combinators parse the "head" of the statement and, afterward, the "body", a list of statements that occur at the same indentation. For illustration, in a Python if statement, the head would be the keyword `if` a condition and colon followed by the end of the line.

```
// Define parsers Statement, Expression, Keyword...

Parser<Expr, char> IfHead =
    from _ in Keyword("if")
    from condition in Expression
    from _ Colon
    select condition

Parser<IfStmt, char> If =
  Indentation.IndentationBlockMany1(
    PythonWhitespace,
    IfHead, // First parse the head of if statement
    Statement, // Then a list of indented statements
    (head, stmts) => new IfStmt(head, stmts)
);
```

The first item gives the level of indentation in the block. If an item is indented differently than required, then an `IndentationError` is returned. `IndentationError` is one of the items possible in a `CustomError`. If the block

of statements ends, i.e., the indentation is less than or equal to the indentation of the head item, then the parser stops.

```
if a > 0: # The head item
    # The next three statements are part of the block
    foo()
    bar()
    foobar()
another_statement() # Parser stops before this statement
```

The difference between `BlockMany` and `BlockMany1` is whether there has to be at least one item in the indented block. In `BlockMany`, there does not have to be; the indented block can be empty. In `BlockMany1`, at least one item has to be present. Otherwise, it is an error.

Many

Further combinators are the `Indentation.Many` and `Indentation.Many1`. These combinators serve to parse a list of items that satisfy a certain relation compared to a reference level of indentation. For instance, if we want to parse a list of items all indented to column four, we would use the following combinator.

```
var listParser =
    Indentation.Many(
    (IndentationLevel)4, // Indent to column 4
    Relation.EQ // Items should have exactly indentation
       level of 4
    Parsers.Spaces,
    itemParser // How to parse items
);
```

The parser of the items is applied as many times as it succeeds. The parser correctly ends when the list of items ends. However, the item parser cannot consume any input while incorrectly attempting to parse the next statement. Otherwise, the entire parser fails.

Optional

The following combinator is used if we need to parse an item while checking indentation, but the item may not be present. In such a case, we use the `Indentation.Optional` combinator. The created parser will attempt to parse the item. It succeeds if the item is present and has the correct indentation. Alternatively, it succeeds if the item is absent and the item parser does not consume any input.

LineFold

The final combinator that is provided is the `Indentation.LineFold` combinator. *Line-folding* appears when a statement can be spread across multiple lines. Let us imagine we have a simple language with line-folding. This language is made of keys and lists of values. The values are separated by whitespace. In such a case, the following two statements are equivalent.

```
Key: value1 value2 value3
```

```
Key: value1 value2
  value3
```

The second statement uses line-folding. The statement continues until we parse an identifier that starts on indentation that is less than or equal to the indentation of the `Key`. We can use the `LineFold` combinator to parse this language.

```
1  // Definitions of parsers Id (Identifier),
2  // keyP (Key) and Spaces
3
4  var KeyValueParser = Indentation.LineFold(
5      Spaces, // Whitespace to consume inbetween lines
6      spaceConsumer => // Indentation checking parser
7      {
8          var valuesP =
9              from values in Parsers.SepBy1(Id, spaceConsumer)
10             from _ in Spaces
11             select values;
12         return from key in keyP
13                from values in valuesP
14                select (key, values);
15     }
16 )
```

The `LineFold` combinator receives a function in which we defined the line-folding combinator. This function has as a parameter a *space consumer* parser. We call it a space consumer because it usually parses an arbitrary amount of whitespace. We define what characters this parser consumes using the first argument of the `LineFold` combinator. This space consumer checks whether we are at the end of the linefold. In the case we are, it fails without consuming input. We use this argument to parse the list of values on line 8. We subsequently also parse whitespace until the next key.

4.3 Advice for writing parsers

During the description of our library, we stated several recommendations. We want to add a few more. Most of these concern the performance of the parsers.

4.3.1 Create parser only once

The first advice is to create parsers only once, whenever possible. The creation of parsers can be expensive, and excessive instantiation of new parsers can hinder performance. We recommend creating the parser as either a static field or as a variable that is reused whenever we need to parse any input. One unfortunate aspect of the query syntax is that any parsers created inside the query are instantiated every time the query is invoked.

```
1 var parser =
2     from firstStr in Parsers.String("abc")
3     from secondStr in Parsers.String("xyz")
4     select firstStr + secondStr;
```

Here, both of the used parsers are newly instantiated whenever the query is invoked. If we need to speed up our parsing, then it is worth considering moving the creation of the string parsers elsewhere and only referencing them from this parser.

4.3.2 Avoid backtracking

The second piece of advice is to avoid backtracking if possible. We use lookahead and backtracking if we cannot decide which path in parsing to choose based solely on the next character. However, often we can eliminate this need by *left-factoring* our parser. Left-factoring is the process of combining the common prefixes of the various paths. Afterward, we can parse the prefix and only then decide which path to take using only the next symbol. We thus transformed a parser with lookahead into a $LL(1)$ parser.

An example of a parser where left-factoring is useful is the parser for escaped sequences in strings. Often, we can either use hexadecimal digits to specify an exact symbol, or we can use a single character that acts as a shortcut. For instance, `\x000A` and `\n` refer to the same character in different ways. One possible way to parse these two different styles of escape sequences is to first attempt the single character option and, if unsuccessful, then the hexadecimal option. There is, however, the problem that they both share the '\' character. We would therefore need lookahead. Consequently, the better way to parse them is first to parse the '\' and then attempt to parse either one of the shortcut characters or the character 'x'.

4.3.3 Parse whitespace after each element

The last piece of advice we will state does not relate primarily to performance but to the ease of writing parsers themselves. If we are parsing a text with whitespace and the whitespace is not significant, then we recommend parsing and discarding this whitespace after each element. There are two benefits to this approach. First, if we do not have a unified strategy for consuming whitespace, then the writing of the parse becomes more complicated. This is because we have to keep in mind which parsers consume whitespace and which do not. The second reason is that we do not want to have whitespace as a prefix in parsers. In such a situation, we would have to constantly left-factor parsers; otherwise, we would need to use lookahead unnecessarily.

Chapter 5

Examples

Two of our goals were to showcase the viability of our solution. The first goal (8) was to create a parser which displays our core library. The second goal (11) stated that we shall create an example parser to demonstrate our ability to parse indentation-sensitive languages. We thus created a simplified parser for Python.

For the first goal, we created a JSON [31] parser. We decided on JSON because it is a well-known and widely used interchange format. Additionally, the JSON grammar is fairly simple.

This chapter describes how to use the two example applications. It also details some aspects of the implementations of the programs, especially when it comes to parsing.

5.1 JSONtoXML

JSONtoXML is a simple application that takes a JSON file as input and outputs an XML [32] file with the equivalent contents. The application is located in the directory `examples/src/JSONParser`. If no output file is specified, the XML is emitted into standard output. The input and output files are specified with the `-input` and `--output` command-line arguments, respectively.

To illustrate, let us have the file `example.json` with the following contents and the subsequent invocation of the application.

```
{
    "fruit": "Apple",
    "size": "Large",
    "color": "Red"
}
```

```
C:\ParsecCore\examples\src\JSONParser> dotnet run -- --input example.json
```

This will be the result of the application.

```
<?xml version="1.0" encoding="utf-8"?>
<root>
    <fruit>
        Apple
    </fruit>
    <size>
        Large
    </size>
    <color>
        Red
    </color>
</root>
```

There are five integral parts to this solution. The first two are the in-memory representations of JSON and XML values. Then there is the JSON parser itself. Finally, we have a converter from JSON to XML values and a class that prints XML values.

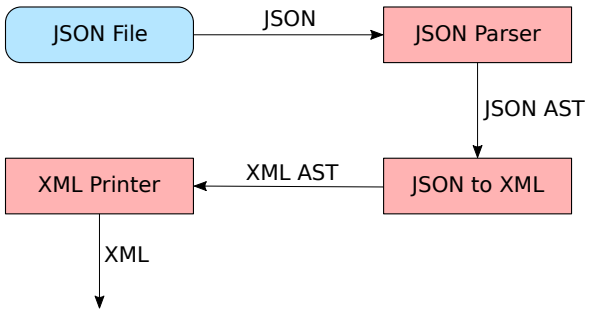

Figure 5.1 Pipeline of the JSONtoXML program

Both the converter and the XML printer are not particularly interesting. They use the visitor pattern to transform the JSON and XML abstract syntax tree into the desired form.

The JSON parser displays one of the ways to work with the ParsecCore library. All the parsers are static members of the `JSONParsers` class. When the `JsonDocument` parser is used for the first time, all of the static members are initialized, and afterward, the same parser can be reused.

We can see that the implementation follows more or less the advice given in the previous chapters. We want to highlight the use of `MapConstant` combinator such as in the `NullValue` parser. Also, the parsers for strings and numbers are left-factored to avoid the use of lookahead.

The JSON parser is not the only parser used in the application. The command-line arguments are also parsed using the ParsecCore library. This parser is composed using the second usual way of constructing parsers. We have a function, `CommandLineParser`, that constructs it, and the returned parser can afterward be reused.

The command-line arguments are parsed using the permutation module. This secures the ability for them to be written in any order.

5.2 PythonParser

PythonParser is a program that parses a simplified version of Python. It takes a python source code as an input, parses it into internal structures, and then outputs it. PythonParser is located in the directory `examples/src/PythonParser`.

The main reason for this parser to exist is to showcase the library's ability to parse indentation-sensitive languages. As such, some features of the Python language that are not necessary to demonstrate this functionality are missing. The `README.md` located in the application's directory includes a list of all missing features.

Since this program is straightforward outside the parser itself, we will only discuss the implementation of the Python parser. The parser is split into five parts. Each is located in the directory `examples/src/PythonParser/Parser`. The first part, in file `Control.cs`, includes parsers for operators, keywords, whitespace, and comments. The second contains string, number, and identifier literals parsers and is located in the file `Literals.cs`. Parsers for expressions make up the third part and are situated in the `Expressions.cs`. These first three parts contain nothing new to us, and so we will not describe them.

The last two parts contain statement parsers. We use the Indentation module for the parsing of compound statements. Let us take the parser for the Python if statement as an example. The individual parts of the if statements, i.e., blocks of code preceded by `if`, `elif`, and `else`, are parsed using `BlockMany1`. This combinator was made precisely for this situation, and its use is natural.

```
1 Parser<(Expr cond, Suite body), char> If =
2     Indentation.BlockMany1(
3         Control.EOLWhitespace,
4         ConditionHead("if"),
5         Statement,
6         (test, stmts) => (test, new Suite(new
              List<Stmt>(stmts)))
7 );
```

Afterward, these individual parts are combined into a single if statement using

the `Indentation.Optional` and `Indentation.Many` combinators. Using these combinators, we ensure that all parts have the same indentation.

```
1 Parser<If, char> IfStatement =
2     from referenceLvl in Indentation.Level<char>()
3     from @if in If
4     from elifs in Indentation.Many(referenceLvl,
          Relation.EQ, Control.EOLWhitespace, Elif)
5     from @else in Indentation.Optional(referenceLvl,
          Relation.EQ, Else)
6     select new If(@if.cond, @if.body, elifs, @else);
```

Finally, the combinator `NonIndented` is used to ensure all of the top-level statements start at the first column.

All combinators used are further described in the section 4.2.3 as well as in the documentation comments in the source code.

Chapter 6

Performance evaluation

In this chapter, we will evaluate the performance of our library. We will assess ParsecCore's performance in contrast with other methods of parsing. We chose to compare ParsecCore to the original Parsec library, *ANTLR*, and Sprache. ANTLR is a parser generator for, among other languages, C# [33]. Additionally, we will describe the process of optimizing our library.

We chose to compare these libraries using a parser for JSON. We already have a JSON parser written for ParsecCore, and there are existing solutions for the other libraries. The exception to this is Sprache. We could not find a JSON parser written using Sprache. As such, we wrote the parser ourselves. We tried to make the Sprache implementation as performant as possible.

We shall use the *BenchmarkDotNet* library for measuring the performance of the C# parsers. For Parsec, we will use the *Criterion* library [34]. [1] The JSON files we will use to benchmark the libraries are in the `test-files` directory. [2] The values shown in the following tables are the means measured by the benchmarks.

As already stated, we used our implementations of a JSON parser for the ParsecCore and Sprache benchmarks. For the Parsec benchmark, we used the library *JSONParser* [39], which is created using Parsec. For ANTLR, we employed an implementation from *Definitive ANTLR 4 Reference* [40].

One of the files, `empty.json`, serves as a control to see any differences between the overhead costs of the different implementations.

[1] The benchmarks were performed with Intel Core i5-8300H 2.30GHz CPU on Windows 10 (10.0.19045.2846/22H2/2022Update). The C# benchmarks were done using .NET 7.0.1 and BenchmarkDotNet 0.13.5. The Haskell benchmarks were executed using GHC 8.10.7 and Criterion 1.6.0.0.

[2] The files `large.json`, `zips.json`, and `countries.json` were taken from github [35] [36] [37]. The JSON found in the "complex" files was taken from json.org [38].

	ParsecCore	Sprache	
File	mean	mean	scaled
`empty`	59.68 μs	59.48 μs	1.00
`list`	110.2 μs	125.6 μs	1.14
`object`	111.3 μs	125.6 μs	1.11
`complex1`	212.4 μs	254.6 μs	1.20
`complex2`	246.3 μs	290.5 μs	1.18
`complex3`	154.4 μs	201.9 μs	1.31
`countries`	54.16 ms	65.96 ms	1.22
`zips`	1.583 s	2.387 s	1.51
`large`	6.152 s	7.914 s	1.28

Table 6.1 Comparison of ParsecCore and Sprache in JSON benchmarks

As we can see, our implementation is about $25\,\%$ faster than Sprache. We ascribe this to the fact that our library uses more imperative implementations of the library parsers. We also want to note that ParsecCore is faster despite us combining errors in the case of both failure and success. This error combining takes a significant portion of the parser's execution time.

	ParsecCore	Parsec			
File	mean	mean	scaled	adjusted	scaled
`empty`	59.68 μs	148 μs	2.48	59.68 μs	1
`list`	110.2 μs	191 μs	1.73	102.7 μs	0.93
`object`	111.3 μs	187 μs	1.68	98.7 μs	0.89
`complex1`	212.4 μs	289 μs	1.36	200.7 μs	0.94
`complex2`	246.3 μs	314 μs	1.25	225.7 μs	0.92
`complex3`	154.4 μs	208 μs	1.34	119.68 μs	0.78
`countries`	54.16 ms	52.8 ms	0.97	52.7 ms	0.97
`zips`	1.583 s	1.22 s	0.77	1.22 s	0.77
`large`	6.152 s	8.08 s	1.31	8.08 s	1.31

Table 6.2 Comparison of ParsecCore and Parsec in JSON benchmarks

The raw values we measured tell us that our library is faster than Parsec. However, as we can see, the benchmark for `empty.json` is much slower in the Parsec version. We estimate that this difference arises from the overhead of opening the files. Because this overhead is quite significant, we thought adjusting the measured means to eliminate the overhead would be useful. We did this by subtracting the difference between the results of the `empty` benchmark.

If we look at these adjusted numbers, then we can observe that ParsecCore is slightly slower than the original Parsec library. On average, our library is 6%

slower than Parsec. We think this slowdown can be attributed to the fact that C# does not optimize lambda methods as well as Haskell. Another advantage Parsec has over ParsecCore is that Haskell is, by default, lazy. Consequently, combining errors has to occur only when the parser input is actually incorrect and the parsing fails.

	ParsecCore	ANTLR	
File	mean	mean	scaled
`empty`	59.68 μs	50.65 μs	0.84
`list`	110.2 μs	62.3 μs	0.57
`object`	111.3 μs	64.5 μs	0.58
`complex1`	212.4 μs	87.0 μs	0.41
`complex2`	246.3 μs	92.2 μs	0.37
`complex3`	154.4 μs	73.6 μs	0.48
`countries`	54.16 ms	12.46 ms	0.23
`zips`	1.583 s	0.601 s	0.38
`large`	6.152 s	2.496 s	0.41

Table 6.3 Comparison of ParsecCore and ANTLR in JSON benchmarks

As suspected, ANTLR is substantially faster than ParsecCore. Concretely, it is about two and a half times faster. We remarked in section 2.4 that parser generators are more performant than parser combinators, and these benchmarks follow this assertion. One of the main reasons parser generators can be faster is because the generated code can be specialized for the specific syntax. We also cannot overlook that ANTLR has gone through years of development.

6.1 Process of optimizing

There were several changes we made in the process of optimizing our library. The first of these was rewriting large portions of the library in an imperative style. Many parser combinators were initially implemented using the query syntax and recursive calls. We refactored the code to use loops and other control flow that is usual for C#. To help with this task, we also introduced the combinators `Then`, `FollowedBy`, and `MapConstant`. These do not add any new ability to ParsecCore, but they are more performant. We also find them to be syntactically more pleasant than their query syntax equivalents.

Another change we made was changing the `Maybe` type into a struct. Structs are located on the stack as opposed to objects that are on the heap. This means that instantiating new structs is faster than creating objects.

Additionally, we added specializations to the `Satisfy` parser for when we are only comparing characters. As the `Satisfy` parser is the only place where we read from the input, it is thus crucial for it to be optimized. Therefore, there were various small changes to the `Satisfy` parser that improved its performance. We will not list these as they were minor and not particularly interesting.

The last optimization we will mention is changing the `IParserInput`. Formerly, when we tested whether two parser inputs were equal, we compared the input's type and position, including the line and column number. We changed this only to compare the offsets of the two inputs. In the usual case, this does not make a difference. However, we implicitly presume the input does not change during parsing. But we do not see this as a problem because the parser's behavior is not defined if the input changes during the parsing process.

Finally, we would like to mention that the solution to the error erasure issue (see section 3.4.1) unfortunatelly significantly slowed down our library. We measured approximately a $25\ \%$ decrease in performance compared to the solution before the change.

In total, we increased the performance of ParsecCore about fivefold. The majority of this improvement came from reimplementing combinators imperatively. Especially changing the `Select` and `SelectMany` extension methods.

Conclusion

Let us revisit the specific goals stated in section 2.3.

1. *Implement the basic Parsec parsers and combinators* — We implemented all of these parser and combinators in the class `Parsers`.

2. *Provide the LINQ syntax for combining parsers* — All four of the discussed clauses (see section 2.1) are supported and behave as discussed.

3. *Support parsing of symbols other than characters* — Users can parse a list of any symbols. Provided combinators support parsers of any input type. If it makes sense, parsers included in the library support input types other than characters.

4. *Implement the ability to read from files without the need to load the entire input into memory* — We provide `StreamParserInput`, which can read from files.

5. *Make the input interface immutable* — The input interface, as well as all three provided implementation classes, behave as entirely immutable.

6. *Have the parsers be covariant with respect to their result type* — The `IResult` is an immutable interface, and so is covariant with respect to the contained type. As such, the parser type is also covariant with respect to its output type.

7. *Robust and extendable error system* — The error system is composed of two levels. `ParseErrors` have two forms that should be able to express most errors that will ever be needed. `ErrorItems` are contained inside the individual errors and can be extended with new types. Most errors not already contained in ParsecCore should be easy to add. We also avoid using strings as much as possible, and thus the errors are easier to keep uniform.

8. *Create an example parser to showcase our core library* — We wrote the JSONtoXML application that includes a parser for JSON. Examples of JSON

files that are correctly parsed are included. Files with syntactic errors are contained as well and the parser reports these errors.

9. *Implement an extension to the library that provides ability to parse permutations* — We created the module `Permutations` to create parsers for permutations easily. We have shown how to use this module in parsing command-line arguments of the JSONtoXML application.

10. *Add an extension to the library that allows easy parsing of indentation-sensitive grammars* — The module `Indentation` contains primitives and combinators needed for parsing indentation-sensitive grammars. With these, most parsers that are required to be indentation-sensitive should be easy to create.

11. *Write an example parser for a simplified version of Python to demonstrate the ability to parse indentation-sensitive languages* — The program `PythonParser` contains a parser for a simplified version of Python that needs to process indentation-sensitive syntax, namely in the parsing of compound statements.

In the previous chapter (see 6) we have also demonstrated that ParsecCore is practically usable. Its performance is comparable to the original Parsec library and it can be used to parse even larger files.

Future work

During the making of ParsecCore, there were some ideas for improving the library that we could not realize. Let us list these ideas here as possibilities for future development.

- Error recovery and multiple errors — Currently, the created parsers stop at the first syntax error they encounter. It would be helpful if the parser would be able to perform error recovery and report as many errors as possible. This would greatly help the users of the written parsers in correcting their mistakes.

- More support for debugging — During debugging, it is often helpful to step through a program to see what it does at every point in the execution. However, stepping through our parser can be confusing due to the frequent use of lambda functions and the query syntax. Thus, improving the programmer's debugging experience would be useful. This could be done with additional combinators that print the current state of the parse or attributes, which help the debugger interpret information.

- Add ability for the user to set and receive their own state — The parser takes as input on the `IParserInput` interface. We could bundle this parameter along with a user-defined state. The parser writer could afterward store and retrieve an arbitrary object and use it during parsing.

Bibliography

[1] Graham Hutton and Erik Meijer. "Monadic parser combinators". In: 1996.

[2] Daan Leijen. "Parsec, a fast combinator parser". In: 2001.

[3] Stack Overflow. *Stack Overflow Developer Survey 2022*. URL: https://survey.stackoverflow.co/2022/#technology-most-popular-technologies (visited on 03/18/2023).

[4] Paolo Martini Daan Leijen and Antoine Latter. *parsec: Monadic parser combinators*. URL: https://hackage.haskell.org/package/parsec (visited on 03/25/2023).

[5] *Haskell Language*. URL: https://www.haskell.org/ (visited on 03/25/2023).

[6] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[7] Maggie Johnson and Julie Zelenski. *Formal Grammars*. URL: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/080%20Formal%20Grammars.pdf (visited on 03/21/2023).

[8] Maggie Johnson and Julie Zelenski. *Top-Down Parsing*. URL: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/090%20Top-Down%20Parsing.pdf (visited on 03/21/2023).

[9] Bryan Ford. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation". In: *SIGPLAN Not.* 39.1 (Jan. 2004), 111–122. ISSN: 0362-1340. DOI: 10.1145/982962.964011. URL: https://doi.org/10.1145/982962.964011.

[10] Roman R. Redziejowski. "Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking". In: *Fundam. Inf.* 79.3–4 (Aug. 2007), 513–524. ISSN: 0169-2968.

[11] Daan Leijen and Erik Meijer. "Parsec: direct style monadic parser combinators for the real world". In: 2001.

[12] C# developers. *Language Integrated Query (LINQ) (C#)*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ (visited on 03/25/2023).

[13] C# developers. *Introduction to LINQ Queries (C#)*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries (visited on 03/25/2023).

[14] C# developers. *Query Syntax and Method Syntax in LINQ (C#)*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/query-syntax-and-method-syntax-in-linq (visited on 03/25/2023).

[15] Erik Meijer, Brian Beckman, and Gavin Bierman. "LINQ: Reconciling Object, Relations and XML in the .NET Framework". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 706. ISBN: 1595934340. DOI: 10.1145/1142473.1142552. URL: https://doi.org/10.1145/1142473.1142552.

[16] Luke Hoban. *Monadic Parser Combinators using C# 3.0*. URL: https://learn.microsoft.com/en-us/archive/blogs/lukeh/monadic-parser-combinators-using-c-3-0 (visited on 03/28/2023).

[17] Tomáš Petříček. "Encoding Monadic Computations in C# Using Iterators". In: *Conference on Theory and Practice of Information Technologies*. 2009.

[18] *Nuget Gallery: Home*. URL: https://www.nuget.org/ (visited on 04/10/2023).

[19] *Sprache*. URL: https://github.com/sprache/Sprache (visited on 03/18/2023).

[20] Paolo Martini Daan Leijen and Antoine Latter. *Text.Parsec*. URL: https://hackage.haskell.org/package/parsec-3.1.16.1/docs/Text-Parsec.html (visited on 04/04/2023).

[21] Datalust. *Superpower*. URL: https://github.com/datalust/superpower (visited on 03/20/2023).

[22] Martín Abadi and Luca Cardelli. "A Theory of Objects". In: *Monographs in Computer Science*. 1996.

[23] C# developers. *Covariance and Contravariance (C#)*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/ (visited on 04/02/2023).

[24] acple. *ParsecSharp*. URL: https://github.com/acple/ParsecSharp (visited on 03/18/2023).

[25] C# developers. *Delegates (C# Programming Guide)*. URL: https://learn.microsoft.com/en-US/dotnet/csharp/programming-guide/delegates/ (visited on 04/04/2023).

[26] ARTHUR I. BAARS, ANDRES LÖH, and S. DOAITSE SWIERSTRA. "FUNCTIONAL PEARL Parsing permutation phrases". In: *Journal of Functional Programming* 14.6 (2004), 635–646. DOI: 10.1017/S0956796804005143.

[27] *Python*. URL: https://www.python.org/ (visited on 04/12/2023).

[28] Michael D. Adams and Ömer S. Ağacan. "Indentation-Sensitive Parsing for Parsec". In: *SIGPLAN Not.* 49.12 (Sept. 2014), 121–132. ISSN: 0362-1340. DOI: 10.1145/2775050.2633369. URL: https://doi.org/10.1145/2775050.2633369.

[29] Daan Leijen and Paolo Martini. *megaparsec: Monadic parser combinators*. URL: https://hackage.haskell.org/package/megaparsec (visited on 04/19/2023).

[30] J. Reschke Adobe-Fastly R. Fielding M. Nottingham and greenbytes. *HTTP/1.1*. RFC 9112. RFC Editor, June 2022. URL: https://www.ietf.org/rfc/rfc9112.txt.

[31] T. Bray and Textuality. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. RFC Editor, Dec. 2017. URL: https://www.ietf.org/rfc/rfc8259.txt.

[32] C. M. Sperberg-McQueen Eve Maler Tim Bray Jean Paoli and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. URL: https://www.w3.org/TR/xml/ (visited on 04/21/2023).

[33] *ANTLR*. URL: https://www.antlr.org/ (visited on 04/26/2023).

[34] Bryan O'Sullivan. *criterion: Robust, reliable performance measurement and analysis*. URL: https://hackage.haskell.org/package/criterion (visited on 04/26/2023).

[35] *Jsoniter*. URL: https://github.com/json-iterator/test-data/blob/master/large-file.json (visited on 04/26/2023).

[36] *Zips*. URL: https://gist.github.com/simonista/0c09188c55356b2baa89f90d04644228 (visited on 04/26/2023).

[37] *all-countries-and-cities-json*. URL: https://github.com/ToniCifre/all-countries-and-cities-json/blob/master/countries.json (visited on 04/26/2023).

[38] *JSON Example*. URL: https://www.json.org/example.html (visited on 04/26/2023).

[39] Alan Hawkins. *JSONParser: Parse JSON*. URL: https://hackage.haskell.org/package/JSONParser (visited on 04/26/2023).

[40] Terence Parr. *Definitive ANTLR 4 Reference*. en. 2nd ed. The pragmatic programmers. Raleigh, NC: Pragmatic Programmers, Feb. 2013.

Appendix A

Project structure

- `/benchmarks/JSONtoXML.Benchmarks` — Benchmarks of the JSON parser. These benchmarks are used in chapter 6. Started by calling `dotnet run -c Release`.

  `/test-files` — The files used for benchmarking.

- `/examples`

  `/src`

  `/JSONParser` — The JSONtoXML application (see section 5.1).

  `/PythonParser` — The PythonParser program (see section 5.2).

  `/tests`

  `/JSONtoXMLTests` — Unit tests for the JSONtoXML application. Run using `dotnet run`.

  `/PythonParserTests` — Unit tests for the PythonParser program. Run using `dotnet run`.

- `/src/ParsecCore` — The source code for the ParsecCore library.

- `/tests/ParsecCoreTests` — Unit tests for the ParsecCore library. Run using `dotnet run`.