Parsing with First-Class Derivatives

Double-blind submission

Abstract

Brzozowski derivatives, well known in the context of regular expressions, have recently been rediscovered to give a simplified explanation to parsers of context-free languages. We add derivatives as a novel first-class feature to a standard parser combinator language. First class derivatives enable an inversion of the control flow, allowing to implement modular parsers for languages that previously required separate preprocessing steps or cross-cutting modifications of the parsers. We show that our framework offers new opportunities for reuse and supports a modular definition of interesting use cases of layout-sensitive parsing.

1. Introduction

The theory and practice of context-free grammars is well-developed, and they form the foundation for most approaches to parsing of computer languages. Unfortunately, some syntactic features of practical interest are inherently not context-free. Examples include indentation-sensitivity as in Haskell or Python, preprocessor directives as in C, or two-dimensional grid tables as in some Markdown dialects. To parse languages with such inherently non-context-free features, language implementors have to resort to ad-hoc additions to their parsing approach, often reducing the modularity of their parser implementation because these ad-hoc additions are cross-cutting with respect to the otherwise context-free syntactic structure of the language.

Parser combinators [9, 14, 19] are a parsing approach that is well-suited for such ad-hoc additions. With parser combinators, a parser is described as a first-class entity in some host language, and parsers for smaller subsets of a language are combined into parsers of larger subsets using built-in or user-defined combinators such as sequence, alternative, or iteration. The fact that parsers are first-class entities at runtime allows an elegant way of structuring parsers, simply by reusing the modularization features and abstraction mechanisms of the host-language. Despite this potential, the reuse of carefully crafted parsers is often limited: Necessary extension points in terms of nonterminals or exported functions might be missing. At the same time, cross-cutting syntactical features like two dimensional layout can often not be separated into reusable modules.

We propose to increase the modularity of parser combinators by adding a combinator to compute a parser's Brzozowski derivative. Derivatives, well known in the context of regular expressions, have recently been rediscovered to give a simple explanation to parsers of context-free languages [1, 6, 15, 16]. A parser p derived by a token c, is again a parser p' characterized by $\mathcal{L}(p') = \{ w \mid cw \in \mathcal{L}(p) \}$. For example, deriving the parser $p =$ `"while"` by the token `'w'` thus yields a parser recognizing the word `"hile"`. Deriving p by any other token yields the empty parser. So far, to the best of our knowledge, Brzozowski derivatives have only been used to describe the semantics of grammars or to implement parsers, but not to extend the language of grammars itself. The addition of a first-class derivative combinator effectively allows user-defined combinators to filter or reorder the token stream, which turns out to increase the modularity of parsers written with the library.

Overall, we make the following contributions:

- We identify the *substream property*, shared by many traditional parser combinator libraries, as the main hurdle precluding modular support for many inherently non-context-free syntactic features (Section 2).
- We present the interface of a parser combinator library with support for first-class derivatives and show how it can be used to modularly define an indentation combinator (Section 3).
- To evaluate feasibility, we have implemented a prototype of a parser combinator library with support for first-class derivatives. In Section 4, we explain how we first based the implementation of the usual combinators on derivatives, and then exposed derivatives also to the user as a combinator.
- To evaluate expressivity, we have performed three additional case studies (beyond the indentation example). Section 5 reports on our experience with skipping a prefix of a parser to increase reuse opportunities, parsing a document with two interwoven syntactic structures, and parsing two-dimensional grid tables modularly.

We review closely related work in Section 6. In Section 7, we discuss future work and phrase open questions. Section 8 concludes the paper.

The implementation of our library and all of the parser combinators presented in this paper is available online[1].

[1] `https://github.com/double-blind-review/fcd`

2. Parser Combinator Libraries and the Substream Property

Parsers process an input stream to compute an abstract syntax tree as result of the parse. Parser combinators are a popular way of writing parsers. They stand in contrast to parser generators, where an executable parser is generated from an external language. Instead, parser combinators are directly expressed in terms of the host language they are embedded in. To this end, a parser is defined compositionally as a host language object from primitives and combinators that are provided by a library.

Being an embedded domain-specific language (EDSL) [8], parser combinators inherit many of the advantages of EDSLs. If the host-language is typed, the typesystem can ensure that parsers are well-typed and thus support the parser developer. Both type- and term-level abstraction mechanisms can be reused to implement modular, reusable parsers. In particular, the use of host language functions allows to express parsers which are parametrized by values of the host language. Parsers can also be parametrized by other parsers. In this case we call the parametrized parser a higher-order parser, or a combinator. We sometimes refer to the parsers which are passed as argument to the combinator as "child-parser".

To recognize a given input stream, every single child-parser might only handle a small fraction of the input stream handled by the composite. Typically, a parser cannot distinguish whether the input stream it processes is the original input stream or only a segment of it. To highlight this, we call the input stream as seen from the point of view of one particular parser object a *virtual input stream*.

Analyzing which parts of the stream are processed by a parser, we observe the following *substream property*.

> A parser's virtual input stream corresponds to a continuous substream of the original input stream.

The sequence of tokens that is processed by a parser as its virtual input stream appears as an exact substream in the original input stream.

Parsers in traditional parser combinator libraries have the substream-property. The property makes sense from a language-generation perspective: The language of a nonterminal in a context-free grammar is always compositional in the languages of the grammar symbols appearing in its productions. One reason is, that in context-free grammars there is neither a way to remove symbols from the words produced by a nonterminal, nor is it possible to add symbols in the middle of a word. Additionally, the only way to create words from smaller words is by concatenation – in turn every word in the language can be split into continuous (potentially nested) regions that have been generated by the corresponding nonterminals.

From a recognition perspective, however, the property impose several restrictions. In particular, the following operations on streams are usually not supported by parser combinator libraries:

Removing from the stream. To implement a simple form of indentation sensitivity, for each block, one might want to strip the indentation and only pass the indented block to the child-parser. However, this block is not represented by a continuous segment in the original input stream. Each line in the block is separated by the whitespaces, representing indentation, that should be skipped over. We want to be able to explicitly select which parts of the original input stream should form the virtual input stream of the child-parser, potentially leaving out tokens.

Extracting interleaved segments of a stream. When parsing a document that contains textual and source-code fragments, one might want to interleave the parsing of the two content types to parse them in a single pass. This requires that parsing of one content type can be suspended to be resumed after parsing a fragment of the other content type. Again, the full source-code and the full text do not form continuous substream of the original input stream.

Adding to the stream. ASCII-Tables, that is tables represented in monospaced text, typically use dashes, pipes and other ASCII characters to define the two dimensional layout and separate the different cells. Again, it is difficult to implement a parser combinator that parses such a table in one pass, using one child-parser for each cell, since the contents of the cells do not form continuous segments in the original input stream. In particular, the virtual input stream of each cell-parser should contain newlines instead of the column-separating characters.

Some of this limitations are addressed by separate stream preprocessing solutions. For instance, the lexer of the indentation sensitive language Python inserts special `INDENT` and `DEDENT` tokens into the token stream, to communicate the layout structure to the parser. However, solutions like this hardly can be reused and in consequence the preprocessing often has to be designed in concert with the particular parser. The communication between lexer and parser is typically one-directional, leading to more complicated lexers. They are even less compositional: It would be desirable to combine the three features mentioned above to write a parser for a complex mixed document with indentation sensitive source code and ASCII tables in the text sections without having to carefully redesign the preprocessing stage.

In the next section we will introduce a parser combinator library that incorporates a new combinator which allows a fine grained control over the input stream delegation to child parsers. By fusing stream-preprocessing and parsing, our library allows to implement each of the above examples as a parser combinator in a separate module, which can then be combined to obtain a parser for the above mentioned complex mixed document. As a consequence of interleaving the preprocessing phase with parsing, more communication

between the phases is possible and thus the preprocessing can now happen dependent on the parsing. Being able to implement the preprocessing as a parser combinator also implies compositionality. In particular, the combinators can be applied recursively to allow for instance for nested tables that itself contain tables and indented code.

Before we give the implementation of our combinator library in Section 4, the next section will first introduce the syntax of our library and illustrate its usage by implementing a parser combinator for indentation sensitivity.

3. First-Class Derivatives: Gaining Fine Grained Control over the Input Stream.

Derivatives are a well-studied technique to construct automata for the recognition of regular [5, 18] as well as context-free languages [6, 15, 20]. By introducing first-class derivatives as novel combinator, we internalize the semantic concept of a derivative and make it available to the parser implementor.

We use the programming language Scala for the presentation of our combinator library and the examples in this and following sections, but our approach is generally applicable to derivative based parsing and is not restricted to Scala. The presentation of derivative based parsers in this section uses the syntax of traditional parser combinators as found in the Scala standard library.

For the scope of this section, we hide implementation details and define a parser in our library by the abstract type[2]:

$$\textbf{type}\ \mathrm{P}[+R]$$

The concrete type together with the implementation of all combinators will be given in Section 4.

In addition a parser is characterized by the function

$$\textbf{def}\ \mathrm{parse}[R](p\colon \mathrm{P}[R], \mathit{input}\colon \mathrm{List}[\mathrm{Elem}])\colon \mathrm{Res}[R]$$

that can be used to process input (earlier alluded to as "original input stream") into a resulting syntax tree. The function parse is universally quantified by R and so given a parser of type $\mathrm{P}[R]$ it will process the list of characters to potentially return a syntax trees of type R. If the input cannot be recognized the returned list will be empty.

For ease of presentation, we fix the type of elements of the input stream (Elem) to character-literals and the type of the parser results (Res) to a list (to allow for ambiguous parses).

3.1 Traditional Parser Combinators

The syntax of our parser combinator library is summarized in Figure 1a. Calling the function $\mathsf{succeed}(r)$ gives a parser that only accepts the empty string and returns r as resulting syntax tree. The parser created by $\mathsf{acceptIf}(\mathit{pred})$ recognizes only a single character, filtered by the predicate pred. The parser fail never accepts any input. As we will see later, the failing parser proves especially useful when dynamically creating parsers using $\mathsf{flatMap}$.

We also include the traditional parser combinators seq, alt and map. The parser $\mathsf{seq}(p, q)$ recognizes an input if it can be split into two subsequent substreams where p recognizes the first and q recognizes the second substream. It returns the cartesian product of their results. The parser $\mathsf{alt}(p, q)$ is used to represent an alternative in a production[3]. The parser $\mathsf{map}(p)(f)$ allows applying the transformation function f as a semantic action to the syntax tree returned by p[4].

[2] The symbol $+$ is Scala syntax to mark type parameter R as covariant.

[3] Following the parsers in the Scala standard library, the results of parser q can be any super type of the results of parser p, indicated by the upper-bound $S >: R$.

[4] The arguments to the parser combinators map and $\mathsf{flatMap}$ are curried, since Scala offers better type inference on curried functions.

```
// primitive parsers
def succeed[R]    : R ⇒ P[R]
def acceptIf      : (Elem ⇒ Boolean) ⇒ P[Elem]
def fail[R]       : P[R]
// traditional parser combinators
def seq[R, S]     :(P[R], P[S])          ⇒ P[(R, S)]
def alt[R, S >: R]:(P[R], P[S])          ⇒ P[S]
def map[R, S]     : P[R] ⇒(R ⇒ S)        ⇒ P[S]
def flatMap[R, S] : P[R] ⇒(R ⇒ P[S]) ⇒ P[S]
def and[R, S]     : (P[R], P[S])         ⇒ P[(R, S)]
// non − traditional parser combinators
def feed[R]       : (P[R], Elem) ⇒ P[R]
def done[R]       : P[R]         ⇒ P[R]
def nt[R]         : (⇒ P[R])     ⇒ P[R]
```

(a) Syntax of the parser combinator library.

$$
\begin{aligned}
c_1 \quad &\rightsquigarrow \mathsf{accept}(c_2 \Rightarrow c_1 \equiv c_2)\\
p \ll c \ &\rightsquigarrow \mathsf{feed}(p, c)\\
p \sim q \ &\rightsquigarrow \mathsf{seq}(p, q)\\
p \ \&\ q \ &\rightsquigarrow \mathsf{and}(p, q)\\
p \gg f \ &\rightsquigarrow \mathsf{flatMap}(p)(f)\\
p \ \hat{}\ f \ &\rightsquigarrow \mathsf{map}(p)(f)\\
p \ \mid \ q \ &\rightsquigarrow \mathsf{alt}(p, q)
\end{aligned}
$$

(b) Syntactic abbreviations with operator precedence from high to low.

```
def any: P[Elem] =
  acceptIf(c ⇒ true)
def no: Elem ⇒ P[Elem] =
  c1 ⇒ acceptIf(c2 ⇒ c1 ≠ c2)
def many[R]: P[R] ⇒ P[List[R]] =
  p ⇒ some(p) | succeed(Nil)
def some[R]: P[R] ⇒ P[List[R]] =
  p ⇒ p ∼ many(p) ^ {case (r, rs) ⇒ r :: rs}
```

(c) Traditional derived parser combinators.

Figure 1. Syntax of our parser combinator library.

In addition to these combinators, that alone can be used to represent context-free grammars, we also include the monadic combinator flatMap and the intersection of two parsers and. The combinator flatMap (p) (f) allows to dynamically create parsers, based on the results of parser p. Using flatMap it is for instance possible to parse a number n and then based on that number create a parser for the remainder of the input stream that recognizes n-many bytes. The intersection and (p, q) of the to parser p and q recognizes a word only if both parser recognize it. It is rarely found in combinator libraries. This probably relates to the fact, that the intersection of two context-free languages in general is not a context-free language. However, as we will see in Section 4 supporting intersection in a derivative based implementation is straightforward and later examples show that the combinator is useful in our framework.

3.2 First-Class Derivatives

Having seen the traditional parser combinators, we now can take on the three combinators feed, done and nt that are non-standard and require some explanation.

First, and most importantly, the combinator feed (p, c) represents the core contribution of this paper. Provided with a parser p and a token c it derives the parser p by the given token. To understand what it means to derive a parser, it requires a brief introduction into derivatives that have their origins in formal languages.

Derivatives, well known in the context of regular expressions [5], have recently been rediscovered to give a simplified explanation to parsers of context-free languages [1, 6, 15, 16]. Roughly, a parser p derived by a token c, is again a parser p' and the language $\mathcal{L}(p')$ of the parser is given by

$$\mathcal{L}(p') = \{\, w \mid cw \in \mathcal{L}(p) \,\}$$

That is, if the parser p recognizes words that start with the token c, then its c-derivative will recognize all suffixes of these words. In addition, we require that the resulting syntax trees produced by parser p' after reading the remaining word w will be the same as the ones produced by p after reading cw. Derivatives immediately give rise to language recognition. A parser accepts a word w, if and only if, after repeated derivation with all tokens in w, the parser accepts the empty word. That is, calling results yields a non-empty list.

Example. Deriving the parser p that only recognizes the word "`for`" by the token "`f`" yields a parser recognizing the word "`or`". Deriving p by any other token yields the empty parser. After also deriving the resulting parser by "`o`" and "`r`" it will accept the empty word, returns the corresponding syntax tree as result, and thus recognizes the word "`for`".

Our new combinator feed now internalizes this semantic concept and offers the derivative of a parser as first-class feature to the parser implementor. For the above example, we write feed$(p,$ `'f'`$)$ or $p \ll$ `'f'` to refer to the "`f`"-derivative of the parser p.

As we will see later, in the presence of feed it can be useful to terminate a parser and prevent that parser from accepting any further input. To this end, the combinator done(p) will return the very same syntax tree, that the parser p would return. However, done(p) does not accept any input and hence can be seen as terminating the parse of p. Thus, for every parser p and every token c, feed (done (p), c) is equal to fail.

The combinator nt (p) is a technical necessity. Laziness is required to allow implementing parsers for grammars with left-, right- and mixed-recursion. To this end, we introduce the combinator nt(p) that is lazy in its argument p[5]. That is, p is only evaluated if used inside the implementation of nt but not during construction of the parser. This way, recursive grammars can be represented as cycles in memory. We apply the following convention for the use of this combinator: All parsers that represent nonterminals should wrap their implementation in a call to the nt combinator.

Example. Using this convention, we can implement a parser that recognizes numbers as sequences of the digits, using the parser $digit : \mathrm{P}[\mathrm{Int}]$:

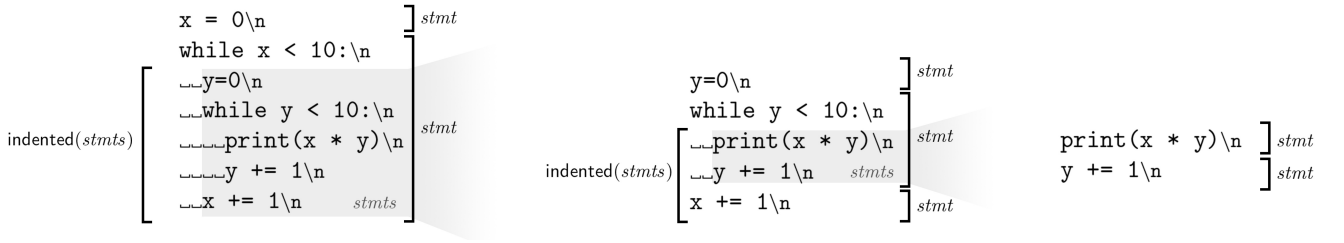
```
val number: P[Int] =
  nt ( map(seq(number, digit)) {
         case (n, d) ⇒ (n * 10) + d
       }
     | digit
     )
```

Here, we define the parser $number$ as a constant using **val**. Since the occurrence of $number$ in the second alternative is guarded by the laziness of nt the definition is well-defined. The second argument to map is provided as an anonymous function that pattern matches on its arguments to bind the results of the parser $number$ to n and the results of $digit$ to d, respectively.

For notational convenience, we use the syntactic abbreviations as summarized by Figure 1b. In addition, we implicitly lift string-literals to a parser of character sequences. We also omit explicit calls to nt, as this is only a technicality, necessary to assert termination of the parser construction. Since discarding of results occurs frequently, we define directed parser combinators for sequence ($p \sim\!\!> q$ and $p <\!\!\sim q$) and intersection (p &> q and p <& q). They recognize the same language as their undirected counterpart, but only return the results of the parser the arrow points to.Using this abbreviations, we can define the parser $number$ again by:

```
val number: P[Int] =
  ( number ~ digit ^^ {case (n, d) ⇒ (n * 10) + d}
  | digit
  )
```

[5] In Scala, arguments of functions can be marked as *by-name* by prefixing their type with ⇒. Laziness then can be encoded by caching the result of forcing the argument.



(a) Original input stream, as recognized by parser *stmts*.

(b) Virtual input stream after delegation by indented.

(c) Virtual input stream after two layers of delegation.

Figure 2. Python code as an example how the input stream can be recognized by a parser *stmts*. Only the boxed content is passed to the nested instances of the parser *stmts*, indenting whitespaces are stripped before by the combinator indented.

In the remainder of this section we will use our new combinator feed to implement a modular parser combinator for a simple form of indentation sensitivity. In this process, we will see how feed is key to overcome the limitations as imposed by the substream property.

3.3 Indentation Sensitive Parsing

Using indentation to indicate block structure goes back to Landin who introduced the "offside rule" [13], which is in variations still used by languages like Haskell and Python. How can indentation sensitivity be implemented with parser combinators?

Ideally, we would like to enable users to define indentation as a combinator $\mathsf{indented}(p)$ that transparently handles indentation, while the body-parser p in contrast is fully agnostic of the indentation. Defining and maintaining the combinator in a separate module could foster reuse and robustness of the implementation.

Figure 3 outlines how such a combinator could be used by giving a simplified skeleton-grammar for the programming language Python. For brevity, only the case for while-statements is given and the productions for parsing expressions (*expr*) are omitted. The parser *stmts* uses some, as defined in Figure 1c to recognize multiple statements, which are terminated by newlines. It is unaware of indentation. In contrast, *block* first reads an initial newline and then makes use of the indented combinator to accept multiple statements which are indented by two spaces.

To understand how an implementation of indentation using first-class derivatives can be structured, let us consider the example of an indentation sensitive program in Figure 2a.

On the toplevel, the program consists of two statements, an assignment and a while-statement. Interestingly, in order to recognize the body of the while-statement, $\mathsf{indented}(\mathit{stmts})$ needs to perform two separate tasks. First, it needs to assert that all lines that belong to the block are indented by two spaces[6]. Second, it needs to invoke the body-parser *stmts* with the contents of the indented block (highlighted in grey). The contents should however not include the two whitespaces at the start of each line. We observe, that what can visually be recognized as one block structure actually consists of five different regions in the original input stream, represented by each line in the highlighted block. In particular those regions are not a continuous substream. In the original input stream, they are separated by the two whitespaces which should be skipped.

```
val stmt: P[Stmt] =
  ( ("while" ~> expr <~ ':') ~ block ^^ {
      case(e, b) ⇒ new WhileStmt(e, b)
    }
  | ...
  )
val expr  : P[Expr]       = ...
val stmts: P[List[Stmt]] = some(stmt <~ '\n')
val block : P[Stmt]       =
  '\n' ~> indented(stmts) ^^ { ss ⇒ new BlockStmt(ss) }
```

Figure 3. A skeleton of a simplified python parser

The virtual stream which is to be recognized by the indentation agnostic body-parser *stmts* is shown in Figure 2b. We can see that the virtual stream consists of three statements, where the second one is again a while-statement extracting two chunks of its virtual stream, "`print(x * y)\n`" and "`y += 1\n`", which are neither form a continuous substream of the current virtual stream nor of the original stream.

Finally, the two chunks represent the virtual stream (Figure 2c) for the last invocation of the *stmts* parser that recognizes the two statements in a straightforward manner.

[6] It is straightforward to extend the definitions to handle indentation by an arbitrary amount (but at least one) spaces, determined by the first indented line.

3.4 Implementation using First-Class Derivatives

Utilizing our new combinator feed, Figure 4a shows the implementation of the parser combinator indented. Indentation is implemented by two mutually recursive parser combinators indented and readLine. Each of the two functions corresponds to one state in an automaton as illustrated in Figure 4b. The combinator indented (p) assures that each line subsequently processed by p is indented by two spaces, without delegating the spaces to p. If the body parser p is "done", that is, it accepts the empty string and can return a resulting abstract syntax tree, then also the indented parser can accept the empty string. Otherwise, after reading two spaces readLine (p) is called, which delegates all tokens to p until it encounters a newline. In that case it hands control back to indented. To this end, after comparing the next token in the input stream with a newline, flatMap is used to capture the token and delegate it to the underlying parser p using the feed combinator. This is a reoccurring pattern when writing parsers with first-class derivatives, captured by following pseudo-code:

$$\mathsf{any} \gg \{\, c \Rightarrow \ldots p \ll c \ldots \}$$

Here, a single character is consumed, just to be bound to c awaiting optional delegation to p.

In the implementation in Figure 4a, similar to an environment-passing style, the body-parser p is explicitly threaded through the calls to indented and readLine, sometimes being fed with tokens. As we shall see shortly, in some cases we can abstract over passing the depending child-parser around.

During the process of parsing with the indentation combinator, tokens are fed to p immediately, once they are available. In particular, in this example p receives every token in the virtual input stream relative to the enclosing combinator indented exactly once.

As becomes visible, the combinator indented can be developed in a separate module, independently of the parser of the concrete language.

3.5 Derived combinators

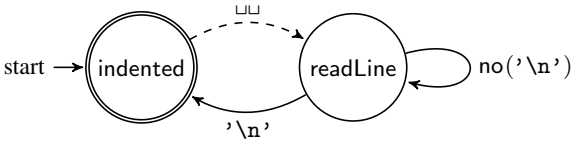
The mapping between the automaton in Figure 4b and the mutually recursive functions in Figure 4a is straightforward. However, explicitly threading p through the involved combinators is repetitive and error-prone. To simplify the definition of parser combinators like indented, we introduce the derived combinators delegate and repeat in Figure 5a.

Delegation. The combinator delegate (p) reflects on the process of delegation to p. It yields a parser that represents the delegation process by always delegating all input it receives to p. There are two interesting ways to interact with such a delegating parser. Firstly, we can extract the *delegatee* at the current state and construct a new parser using it. This is achieved by using flatMap to access the results of delegate. Secondly, we can specify the end of delegation process to a

$$\begin{array}{l}\textbf{def}\ \mathsf{indented}[T](p\colon \mathrm{P}[T])\colon \mathrm{P}[T] = \\ \quad \mathsf{done}(p) \mid (\mathit{space} \sim \mathit{space}) \rightsquigarrow \mathsf{readLine}(p)\end{array}$$

$$\begin{array}{l}\textbf{def}\ \mathsf{readLine}[T](p\colon \mathrm{P}[T])\colon \mathrm{P}[T] = \\ \quad (\ \mathsf{no}(\texttt{'\textbackslash n'}) \gg \{\, c \Rightarrow \mathsf{readLine}(p \ll c)\,\} \\ \quad \mid \texttt{'\textbackslash n'} \quad\ \gg \{\, c \Rightarrow \mathsf{indented}(p \ll c)\,\} \\ \quad)\end{array}$$

(a) Definition of the combinator indented (p) in terms of $\ll$.



(b) Automaton modeling the control flow of the parsers in Figure 4a; transitions with solid lines delegate the input to the underlying body parser.

Figure 4. Implementing the combinator indented (p) in terms of first-class derivatives.

particular region of the input stream. This is achieved using intersection. We use the pattern p &> delegate(q) to express *delimited delegation*. Here, we refer to the parser p as *delimiter*. The full virtual input stream of such a delimited delegation is fed to the delegatee q. However, intersection with p restricts which words should be accepted, effectively delimiting the region of the delegated input stream.
In concert with flatMap we can implement the following parser that accepts any two tokens, delegates them to some parser p to then construct a parser that accepts a token 'a' before continuing with the delegatee:

$$((\mathsf{any} \sim \mathsf{any})\ \&\!\!>\ \mathsf{delegate}(p)) \gg \{\, p_2 \Rightarrow \texttt{'a'} \rightsquigarrow p_2 \}$$

Using delimited delegation, the delegation does not need to be performed on a token-by-token basis as in Figure 4a. Instead, the region that should be delegated can be expressed in terms of the delimiter.

Iteration. The second derived combinator repeat (f) (p) takes a function to construct a (delimited) delegation. It then takes a parser p as initial delegatee and threads p through repeated applications of f. At every level of recursion, repeat is successful if and only if p is successful.

Using these two derived combinators, we can give an equivalent but more concise definition of the indentation parser (Figure 5b). After skipping two spaces, delimited delegation is used to feed tokens to p until a newline is encountered; the resulting parser (extracted by flatMap in the implementation of repeat) is then used to iterate and process the next line.

This example illustrates, how the novel combinator $p \ll a$ allows an inversion of the control flow and explicit handling of the input stream. This paves the way for further interesting use cases, which we present in Section 5.

```
def delegate[R](p: P[R]): P[P[R]] =
  succeed(p) | any ≫ { c ⇒ delegate(p ≪ c) }

def repeat[R](f: P[R] ⇒ P[P[R]]): P[R] ⇒ P[R] = p ⇒
  done(p) | f(p) ≫ repeat(f)
```

(a) Derived combinators to abstract over input-stream delegation and threading of parsers.

```
val line = many(no('\n')) ~ '\n'
def indented[T]: P[T] ⇒ P[T] = repeat[T] { p ⇒
  (space ~ space) ~> (line &> delegate(p))
}
```

(b) Definition of the combinator indented(p) in terms of delegate.

Figure 5. Using the derived combinator delegate to encapsulate delegation.

4. Implementation

In this section, after establishing the basic prerequisites, we present the implementation of our parser combinator library, show how to implement optimizations by using dynamic dispatch and compare our approach to Might et al. [15].

First, we define derivatives on languages formally. To this end, given some alphabet A, we say $w \in A^*$ is a word and $\mathcal{L} \subseteq A^*$ is a language over the alphabet A. We sometimes refer to the elements of the alphabet as "character" or "token". ϵ is used to denote the empty word. The (left) *derivative* of a language $\mathcal{L}$ by a token a is defined by

$$D_a(\mathcal{L}) = \{\, w \mid aw \in \mathcal{L} \,\}$$

Symmetrically, also right derivatives exists, but we will focus on the former. We can lift the notion of derivatives from tokens to words, by

$$D_{aw}(\mathcal{L}) = D_a(D_w(\mathcal{L}))$$

Example. The a-derivative of the language $\mathcal{L} = a^+ = \{\, a, aa, aaa, ... \}$ is $D_a(\mathcal{L}) = a^* = \{\epsilon, a, aa, ...\}$. The b-derivative of the same language in turn is $D_b(\mathcal{L}) = \emptyset$.

4.1 Derivative of a parser

In the previous section a parser was defined as an abstract type $P\ [+R]$. To define the parsers in our framework we now instantiate this abstract type with the equally named trait[7], defined in Figure 6a. Hence, a parser p is an object implementing the trait $\mathrm{P}[R]$. Its behavior is uniquely defined by observations that can be made using the two methods of the signature: The method results returns a list of syntax trees if the virtual input stream of that parser would end at the given point, and the method derive takes the parser into the next state, consuming the provided element. Analogously to derivatives on language, we call the result of p.derive (c) the *derivative* of parser p by the character c. We will also use Scala's support for infix notation and write(p derive c).

Using derive and results we can now give an implementation of parse as follows:

```
def parse[R](p: P[R], input: List[Elem]) =
  input.foldLeft(p) { (p2, el) ⇒ p2 derive el }.results
```

The definition of parse helps us to more precisely define our informal description of result-preservation from the previous section as:

$$\mathrm{parse}(p, aw) = \mathrm{parse}(p \text{ derive } a, w)$$

Finally, we define the language of a parser $\mathcal{L}\ (p)$ by induction over the length of the words in the language. A parser p accepts the empty word, if and only if the list of results is non-empty:

$$\epsilon \in \mathcal{L}(p) \text{ iff } p.\mathrm{results} \neq \mathrm{Nil}$$

A word aw is in the language of the parser p, if the suffix w is in the language of the a-derivative of the parser p:

$$aw \in \mathcal{L}(p) \text{ iff } w \in \mathcal{L}(p \text{ derive } a)$$

Using this definition, we can now relate derivatives of languages and derivatives of parsers by the following commutation:

$$D_a(\mathcal{L}(p)) = \mathcal{L}(p \text{ derive } a)$$

That is, the a-derivative of a parser's language is the language of the parser's a-derivative.

4.2 Derivative-Based Implementation of Parser Combinators

In Figure 1a we have seen the syntax of parser combinators in our library. We will now define each of the combinators by a parser-object implementing the corresponding behavior in terms of the methods results and derive. At first, let us consider the set of combinators in Figure 6b, that, when used recursively, can recognize the class of context free languages[8]. For all combinators (except fail which does not take arguments), we use anonymous functions to implement the interface of Figure 1a.

As one might expect from intuitive description of the parser primitives, succeed(r) is implemented as a parser that immediately succeeds with the given result r and fails on any further input, acceptIf $(pred)$ succeeds after consuming one character, but only if the character matches the predicate $pred$, and the combinator fail never has any result and will fail on any further input.

[7] For our purposes it is enough to interpret a Scala trait as interface with abstract members.

[8] Later in this section we present the combinator nt which is necessary for recursive definitions.

The implementation of the combinator $\mathsf{map}\,(p)\,(f)$ uses the method map defined on Scala collections to transform the results.

Using Scala's syntax for for-comprehensions, the results of the combinator seq are defined as the cartesian product of the results of p and q. In consequence, only if both parsers return a syntax tree, the sequence of p and q can successfully return a result. The definition of the derivative of a sequence makes use of the nullability combinator done, which returns the same results as p, but terminates the parser by returning fail on every step. To now derive the sequence of p and q by el, we have to consider two cases. First, p might be done, that is, it accepts the empty word. In that case we continue with q derive el. Second, p still can consume input, so we continue with p derive el.

Finally, the alt combinator explores both alternatives in parallel, aggregating the results using list concatenation.

Example. Just using the above combinators we can define a parser that recognizes the language $\mathcal{L} = \{\, a, ab \,\}$

$$\textbf{val}\ ex = \texttt{'a'} \sim (\mathsf{succeed}(()) \mid \texttt{'b'})$$

and derive it by 'a' to get

$$\begin{aligned}
&(ex \text{ derive } \texttt{'a'})\\
&\quad = (\ \mathsf{done}(\texttt{'a'}) \sim ((\mathsf{succeed}(()) \mid \texttt{'b'}) \text{ derive } \texttt{'a'})\\
&\qquad \mid \mathsf{succeed}(\texttt{'a'}) \sim (\mathsf{succeed}(()) \mid \texttt{'b'})\\
&\qquad)\\
&\quad = (\ \mathsf{done}(\texttt{'a'}) \sim (\mathsf{fail} \mid \mathsf{fail})\\
&\qquad \mid \mathsf{succeed}(\texttt{'a'}) \sim (\mathsf{succeed}(()) \mid \texttt{'b'})\\
&\qquad)\\
&\quad = \mathsf{succeed}(\texttt{'a'}) \sim (\mathsf{succeed}(()) \mid \texttt{'b'})
\end{aligned}$$

In the last step, we used the following standard equivalences [5, 15]:

$$p \mid \mathsf{fail} \ = \mathsf{fail} \mid p \ = p \tag{1}$$

$$p \sim \mathsf{fail} = \mathsf{fail} \sim p = \mathsf{fail} \tag{2}$$

Additionally to the above (context-free) parser combinators, Figure 6c gives the implementation of the monadic combinator flatMap and the intersection of two parsers $\mathsf{and}\,(p, q)$. Similar to alt, the latter just derives both parsers p and q in parallel but like seq returns the cartesian product of the results.

The result of the combinator $\mathsf{flatMap}\,(p)\,(f)$ is defined to be the concatenated results of the parsers after applying the function f. Similar to seq, for the derivative of flatMap two cases have to be considered. Firstly, if p has results, f can be applied to the results to obtain a list of parsers that are then joined using alt. Secondly, p itself is derived by el and the result is wrapped in a call to flatMap.

Finally, Figure 6d defines our new parser combinator $\mathsf{feed}\,(p, el)$ simply as an alias for the method derive.

```
trait P[+R] {
  def results: Res[R]
  def derive: Elem ⇒ P[R]
}
```

(a) The interface of a parser in our library.

```
def succeed[R] = res ⇒ new P[R] {
  def results = List(res)
  def derive  = el ⇒ fail
}
def acceptIf = pred ⇒ new P[Elem] {
  def results = Nil
  def derive  = el ⇒ if (pred(el)) succeed(el) else fail
}
def fail[R] = new P[R] {
  def results = Nil
  def derive  = el ⇒ fail
}
def map[R, S] = p ⇒ f ⇒ new P[S] {
  def results = p.results.map(f)
  def derive  = el ⇒ map(p derive el)(f)
}
def seq[R, S] = (p, q) ⇒ new P[(R, S)] {
  def results = for (r ← p.results; s ← q.results) yield (r, s)
  def derive  = el ⇒ alt(seq(done(p), q derive el),
                         seq(p derive el, q))
}
def alt[R, S >: R] = (p, q) ⇒ new P[S] {
  def results = p.results ++ q.results
  def derive  = el ⇒ alt(p derive el, q derive el)
}
def done[R] = p ⇒ new P[R] {
  def results = p.results
  def derive  = el ⇒ fail
}
```

(b) Implementation of combinators that can express context-free languages.

```
def and[R, S] = (p, q) ⇒ new P[(R, S)] {
  def results = for (r ← p.results; s ← q.results) yield (r, s)
  def derive  = el ⇒ and(p derive el, q derive el)
}

def flatMap[R, S] = p ⇒ f ⇒ new P[S] {
  def results = p.results.flatMap {r ⇒ f(r).results}
  def derive  = el ⇒ p.results.map {r ⇒ f(r).derive(el)}
                .foldLeft(flatMap(p derive el)(f))(alt)
}
```

(c) Implementation of non-context-free parser combinators

```
def feed[R] = (p, el) ⇒ p derive el
```

(d) Implementation of our new combinator feed.

Figure 6. Implementation of our combinator library, defined in terms of derivatives.

4.3 Nonterminals

Without support for recursive definitions, the combinators in Figure 6b can only express regular languages [5]. To also allow recursively defined parsers, in our framework, recursive definitions are explicitly marked as such by using the nt combinator. However, by doing so the combinator fulfills multiple purposes. It allows to represent parsers as cyclic structures in memory, assures that the derivative of a recursive parser can in general be recursive again, and uses fixed point iteration to compute attributes over the parser graph.

The implementation of nt is given in Figure 7. For one, it guards the construction of the parser-graph by being lazy in its argument p and thus allows to create cyclic structures in memory[9]. For instance, omitting the combinator nt in

$$\textbf{val}\ as\colon \mathrm{P}[Any] = \mathsf{nt}(as \sim \texttt{'a'} \mid \mathsf{succeed}(()))$$

would immediately diverge, since evaluating the body of as itself involves constructing the parser of as. Deriving as by 'a' illustrates another, similar problem. To compute the derivative of as, we need the 'a'-derivative of as itself. In general, the derivative of a recursive parser might again be a recursive parser. This can be achieved by the following first attempt at implementing derive:

$$\begin{array}{l}\textbf{def}\ \mathrm{derive} = el \Rightarrow \\ \quad memo.\mathrm{getOrElseUpdate}(el, \mathsf{nt}(p\ \mathrm{derive}\ el))\end{array}$$

This simple form of memoization, local to the nonterminal, assures that computing the derivative with the same token a second time, will yield a reference to the very same parser. In addition, the laziness of nt assures that this is even the case if the derivative is requested during the computation of the derivative itself. Thus, the a-derivative of as gives as_2, which is almost exactly as just with a change in the result of succeed:

$$as_2 = \mathsf{nt}(as_2 \sim \texttt{'a'} \mid \mathsf{succeed}((), \texttt{'a'})))$$

The method results in nt is implemented by a fixed point iteration, using Nil as bottom of the lattice, set union as join and set-inclusion as ordering[10]. Due to the potentially cyclic structure, the computation of $p.\mathrm{results}$ might again involve the computation of the results on the nonterminal-parser itself.

For instance, to compute $as.\mathrm{results}$ we start with Nil as bottom of the lattice. The left-hand-side of the alternative ($as \sim$ 'a') gives the crossproduct of Nil and Nil, hence Nil. The right-hand-side gives $\mathrm{List}(())$. In a second iteration, starting with $\mathrm{List}(())$ as previous result, $as \sim$ 'a' gives the crossproduct of $\mathrm{List}(())$ and Nil and the right-hand-side did not change, again resulting in $\mathrm{List}(())$. The implementation of the fixed point computation itself is completely standard and no different from that of related work.

```
def nt[R] = _p ⇒ new P[R] {
  lazy val p  = _p
  val memo    = mutable.HashMap.empty[Elem, P[R]]
  val res     = attribute(p.results)
  def results = res.value
  def derive  = el ⇒ memo.getOrElseUpdate(el, {
    memo(el) = fail
    if (p.empty) {
      fail
    } else {
      nt(p derive el)
    }
  })
}
```

Figure 7. Implementation of the combinator nt, definition of attribute and empty omitted.

The implementation of memoization in Figure 7 slightly differs from the first attempt presented above. The modifications are necessary to avoid divergence with exotic parser like the following:

$$\textbf{lazy val}\ exotic\colon \mathrm{P}[Any] = \mathsf{nt}(exotic \ll \texttt{'a'})$$

To compute $exotic.\mathrm{results}$ first the a-derivative has to be computed. However, the computation of the derivative again involves the a-derivative and hence diverges.

The reason is, that in our implementation above, derive will always just return a new nonterminal, guarding the actual derivative (which diverges) with laziness, leading to a non-productive, infinite chain of nonterminals. To avoid this, instead of just returning the nonterminal that represents the derivative, we can check whether the underlying parser p is the empty language. If this is the case, then it is safe to assume that also the derivative of the nonterminal-parser will be empty, so we can as well return fail. To this end, we use $p.\mathrm{empty}$ to obtain a conservative approximation of whether the parser p only recognizes the empty language[11]. However, computing $p.\mathrm{empty}$ will force the evaluation of parser p, which in turn leads to computing the derivative and hence diverges. This can be avoided, by first storing the parser fail as preliminary result in the memo-table, which is then updated with the actual result after computing the derivative[12].

In our implementation, the parser $exotic$ thus behaves like the parser fail.

[9] Since Scala only support by-need parameters, laziness is encoded by caching the result of forcing $\_p$ as p.

[10]

[11] We omit the implementation of the attribute empty here which is also implemented by fixed point iteration.

[12] This is similar to how blackboxing is used in the programming language Haskell to implement the forcing of a thunk.

```
trait P[+R] {
  ...
  def seq1[S](q: P[S]): P[(R, S)] = q.seq2(p)
  def seq2[S](q: P[S]): P[(S, R)] = new P[(S, R)] {...}
}
...
def fail[R] = new P[R] {
  ...
  override def seq1[S](q: P[S]) = fail
  override def seq2[S](q: P[S]) = fail
}
...
def seq[R, S] =(p, q) ⇒ p.seq1(q)
...
```

Figure 8. Using double dispatch to implement compaction rules in order to reduce the size of the parser-graph.

4.4 Compaction by Dynamic Dispatch

Equivalences, like the ones earlier in this chapter, cannot only be used for reasoning. Applying them in a directed way, they can lead to a significantly cut down of the parser-graph and thus improve the performance of parsing [1, 15]. This process is also called *compaction*. To elegantly implement compaction rules in our object oriented setting, we slightly need to modify the parser implementation.

Figure 8 illustrates how the equivalence from Equation 2 can be implemented as compaction rule using an encoding of double dispatch. The implementation of combinators just forward to a dispatching call on the first receiver, which itself dispatches on the second receiver. The original implementation of seq in turn can now be found as default implementation of seq_2. The two methods seq_1 and seq_1 are template methods which should be overwritten for optimization. Such an optimization is achieved in the implementation of the fail combinator, overwriting seq_1 and seq_2 to immediately return fail.
For unary combinators, such as for the compaction rule

$$\mathsf{done}(\mathsf{fail}) = \mathsf{fail}$$

simple dynamic dispatch is sufficient.

This approach is similar to *smart-constructors* in functional programming. Optimizations are performed already during the construction of the parser objects.

4.5 Parsing with Derivatives: Related Work

The implementation of our library is based on derivatives and builds on prior work by Might et al. [15]. We briefly want to point out similarities and differences. A first high level difference is, that we chose an object oriented decomposition, grouping the equations for results and derive per combinator, while Might et al. on the other hand maintain an explicit term representation of the grammar and define their equivalent of the functions in terms of pattern matching.

Like Might et al. we use fixed point iteration for the computation of the parser results as well as memoization and laziness to support recursive grammars by allowing cycles in the parser-graph. However, at the same time we limit this treatment to parsers which represent non-terminals, only. Annotating potentially left-recursive parsers and only applying memoization selectively is a well-known technique, for instance used by the Packrat-parser implementation in the Scala standard library. While Might et al. require all parser combinators to be lazy in their arguments, restricting this requirement to only the combinator nt also has a practical benefit. This way our implementation can more easily be applied in languages where encoding laziness can be cumbersome (as in Java). At the same time, limiting the handling of laziness, fixed point iteration and memoization to one combinator also makes it easier to reason about the behavior of all other combinators in isolation.

Additionally to the parsers that allow expressing context-free grammars, we also implement the parser combinators and, flatMap and our novel parser combinator feed.

Finally, we show how compaction rules can be implemented in an object oriented setting, using simple dynamic dispatch for nullary and unary combinators and an encoding of double dispatch for binary combinators.

5. Applications

Now that we have seen how first-class derivatives can be implemented, we practically evaluate our approach by giving more examples that illustrate the gained expressive power.

As we shall see, the expressive power reveals itself with respect to two aspects: The definition of new modular combinators that unify stream processing and parsing; and the reuse of existing parser definitions that lacked appropriate extension points. Obviously, the two aspects are not orthogonal since existing parser definitions can be reused in the definition of new combinators.

5.1 Increased Reuuse through Parser Selection

Even when designed with great care, a parser implementation for a certain language will always only export a limited set of extension points which can be used to reuuse the parsers or extend them. For instance consider the parser for while-statements in Figure 3. In a traditional setting, implementing a parser for an until-statement would repeat most of the implementation of the while statement without any possibility of reuse.

In our framework, we can use the combinator feed to navigate into a grammar and select a “sublanguage”. Since derivatives are defined for languages, we can perform this action even without knowing the actual parser implementation. We can thus select the parser for the body of the while-statement by $stmt \lll$ `"while"`. Similar to lifting deriva-

tives of parsers to words, $\lll$ is defined by lifting $\ll$ to a sequence of tokens (or strings). Omitting the handling of the resulting syntax tree, the parser for until-statements can now be implemented by:

$$\textbf{val}\ \mathit{untilStmt} = \texttt{"until"} \rightsquigarrow (\mathit{stmt} \lll \texttt{"while"})$$

We can also again view this from a language generation perspective: Alternative productions just grow the language and the sequence of parsers (concatenation) adds to the words in the language. At the same time, these are also the operations traditional parser combinator libraries offer to reuse existing parsers.

This stands in contrast with intersection, which is "dual" to alternative and quotienting (deriving) which is "dual" to concatenation. Offering these operations in a parser combinator library allows new ways to reuse existing parsers which we call *restriction* and *selection*.

As another example, intersection could be used to restrict the $expr$ parser to productions that start with a number while feed can be used to derive $expr$ by "`0.`" to select a parser for the fractional digits.

5.2 Modular Definitions as Combinators

The introductory example of indentation-sensitivity showed that first-class derivatives also are useful to gain fine grained control over a child parser's virtual input stream. Building on this functionality, in the remainder of this section, we develop additional combinators with increasing complexity: From simple stream processing to a combinator for one-pass parsing of two dimensional ASCII-tables.

Stream preprocessing. One example for a simple form of stream-preprocessing is the escaping and unescaping of special tokens or sequences of tokens in the input stream. Just using feed, done and parametrized parsers we can implement unescaping of newline symbols as a combinator unescape:

$$\begin{array}{l}\textbf{def}\ \mathsf{unescape}[R](p\colon \mathrm{P}[R])\colon \mathrm{P}[R] = \\ \quad (\ \texttt{'\textbackslash\textbackslash'} \rightsquigarrow \mathsf{any} \gg \{\, c \Rightarrow \mathsf{unescape}(p \ll \mathrm{unescChar}(c))\,\} \\ \quad \mid \mathsf{no}(\texttt{'\textbackslash\textbackslash'}) \quad \gg \{\, c \Rightarrow \mathsf{unescape}(p)\,\} \\ \quad)\end{array}$$

The function $\mathrm{unescChar}$ maps characters like `'n'` to their unescaped counterpart `'\n'`. Since unescaping is defined as a parser combinator and not a separate preprocessing phase, it can selectively be applied to other parsers and thus its scope is limited to the virtual input stream of that parser. For instance one could use unescape on a parser for regular expressions to reuse the parser inside string-literals, where special characters need to be escaped.

Mixed content: Delegating to two parsers. Our implementations of combinators for indentation sensitivity and simple stream pre-processing are examples of parser combinators that delegate the input stream to a single child-parser. However, it is straightforward to generalize the notion of delegation to multiple parsers. As an example for delegation to two parsers let us consider a document with two interleaved content fragments, one for source-code of a particular language the other one for textual contents. Similar to fenced code blocks in markdown, the code fragments a explicitly delimited by "`~~~`". Virtually, we want to split the document into one code-fragment and into one text-fragment. However, we want to parse the document in one pass, using a separate parser for the code fragments and another one for the textual contents.

Figure 9a gives the definition of two mutually recursive parser combinators inCode and inText, that given the two content parsers, implement parsing of the mixed document by interleaved delegation[13]. To enable switching between delegating to $text$ and to $code$ the two parsers are passed to all subsequent calls of inCode and inText. In effect, the virtual input stream of the $code$ parser is given by all content inside the fenced code blocks and the virtual input stream consists of all contents outside the fenced code blocks.

ASCII tables: Delegating to a list of parsers. Our approach is not limited to delegate to two (or a statically known amount of) parsers. It is possible to delegate to a list of parsers, where the size of the list dynamically depends on previously processed input. This is for instance the case when parsing an ASCII table, such as:

```
+------+-------------+------+
|x += 1|while x < 1:  |y += 1|
|      |  print(x * y)|      |
+------+-------------+------+
```

Here, we do not know in advance how many parsers we will need for the cells of one row. However, after processing the first line we know the vertical layout in terms of column size (in our case $\mathrm{List}\ (6, 14, 6)$). Depending on the layout, to parse one row of the table, we now can initialize one parser per table cell. Since the contents of the cell do not satisfy the substream property, we need to interleave the delegation of all involved cell-parsers. To process the second line, we parse the initial pipe and delegate six tokens to the first cell parser. Interestingly, on encounter of the second pipe in that line, we feed a newline to the first parser before we suspend it and continue delegating to the remaining two parsers. To process the third line, we resume delegation to the corresponding cell parsers of the previous line. Finally we process the terminating row-separator.

Similar to this informal description, the combinator table $(cell)$ parses a two dimensional table, given an initial parser for cells (Figure 9c). To handle administrative details of delegating to a list of parsers it uses two new derived combinators defined in Figure 9b. The combinator distr(ps) takes a list of parsers and sequences them, it thus distributes parsers over lists. The combinator collect (ps) behaves like

[13] With support for negative lookahead we could more precisely express, that the second alternative in inCode and inText should not start with the delimiting marker. This is left for future work.

```
def inCode[R, S](text: P[R], code: P[S]): P[(R, S)] =
  ( "~~~" ~> inText(text, code)
  | any   >> { c => inCode(text, code << c) }
  )
def inText[R, S](text: P[R], code: P[S]): P[(R, S)] =
  ( done(text & code)
  | "~~~" ~> inCode(text, code)
  | any   >> { c => inText(text << c, code) }
  )
```

(a) Combinators for interleaved parsing of fenced code blocks.

```
def distr[T](ps: List[P[T]]): P[List[T]] =
  ps.foldRight(succeed(Nil)) { case (p, ps_2) =>
    (p ~ ps_2) ^^ { case (r, rs) => r :: rs }
  }
def collect[T](ps: List[P[T]]): P[List[T]] =
  ps.foldRight(succeed(Nil)) { case (p, ps_2) =>
    done(p) >> { r => ps_2 ^^ (rs => r :: rs) }
  }
```

(b) Derived parser combinators for handling lists of parsers.

```
type Layout = List[Int]
def table[T](cell: P[T]): P[List[List[T]]] =
  (head <~ '\n') >> { layout => body(layout, cell) }
def head: P[Layout] = some('+' ~> manyCount('-')) <~ '+'
def body[T](layout: Layout, cell: P[T]): P[List[List[T]]] =
  many(rowLine(layout, layout.map(n => cell)) <~ rowSep(layout))
def rowSep(layout: Layout): P[Any] =
  layout.map { n => ("-" * n) + "+" }.foldLeft("+")(_+_) ~ '\n'
def rowLine[T](layout: Layout, cells: List[P[T]]): P[List[T]] =
  ( ( '|' ~> distr(delCells(layout, cells)) <~ '\n') >> {
      cs => rowLine(layout, cs)
    }
  | collect(cells)
  )
def delCells[T](layout: Layout, cells: List[P[T]]): List[P[P[T]]] =
  layout.zip(cells).map {
    case(n, p) => delegateN(n, p).map(p => p << '\n') <~ '|'
  }
```

(c) Modular definition of a parser combinator for ASCII-tables.

Figure 9. Parser combinators for additional case studies illustrating delegation to more than one parser.

done (p) but lifted to a list of parsers. It is only successfull, if all involved parsers can return a result and aggregates all results in a list. For brevity, we omitted the implementation of delegateN and manyCount. delegateN(n, p) delegates the next n tokens to p and then fails on further input. manyCount(p) is similar to many but returns the number of elements in the resulting list.

In the implementation of the table-parser flatMap is used twice for data-dependency: Firstly, it is required to access the layout and dynamically construct the corresponding parsers. Secondly, it is used after reading every line of a row to access the suspended parsers and continue with the next line.

The implementation is modular: It is possible to define a table-parser once and for all as a separate module – no cross cutting changes to other parsers are necessary. At the same time, being defined as a parser combinator, it can just be used recursively to allow for nested tables.

In the spirit of parser combiantors, all the combinators we have implemented in this paper, such as preprocessing, indentation, mixed-documents and tables can naturally be combined to parse a complex structured document.

6. Related Work

In this section we review work that is closely related either in terms of implementation or expressive power.

Derivative based parsing. Parsing with derivatives is a relatively new research area. Still, there already exist multiple parser combinator libraries using derivatives as basis for their implementation. Might et al. [15] introduce parsing with derivatives as a general parsing technique that is simple to understand, Danielsson [6] uses derivatives as parsing backend for a parser combinator library in Agda, that guarantees to be total. Moss [16] gives a derivative based implementation for parsing expression grammars (PEG) implementing support for biased choice and lookahead without consuming tokens. Adams et al. [1] show that derivative based parsing can be cubic in its worse case complexity. They propose optimizations and further compaction rules that they claim to make derivative based parsing performant enough to be used in practice.
However, the above mentioned approaches only use derivatives as implementation technique. None of them offers first-class derivatives as part of the term language to the user. They all have a similar expressive power as traditional combinator libraries.

Data-dependent grammars. Data-dependent grammars [4, 10, 11] support implementing parsers for many of the use cases mentioned in the present paper. In the framework of data-dependent grammars the user can express a certain context-sensitity by saving context information in global state and later use the state in predicates to constrain the application of productions. The parser framework implicitly threads this state through the parsing process and evaluates the constraints to guide recognition. While data-dependent grammars offer a declarative abstraction over passing global state, they are implemented as parser generator, not as a combinator library. Thus users are limited to the abstractions provided by the framework.

Iteratees. Kiselyov [12] introduces a programming style, which he refers to as *Iteratee IO*. Using the concepts of iteratees (essentially stream consumers that can be chained), enumerators (producers) and enumeratees (consumer and producer at the same time) as building blocks, Iteratee IO is a structured way of processing potentially large data incrementally. Iteratees can be also used to implement parsers. Similar to derivative based parsing and other forms of online parsing, the resulting parsers process the stream incrementally. In the terminology of the Iteratee IO, our parsers are *iteratees* and the first-class derivative is an *enumerator*.

Kiselyov introduces a combinator en_str that is very similar to our combinator feed. However, like in the related work on derivative based parsing, en_str is again only used for the formalization of the parsers and not explicitly designed as tool for a user to define parsers.

7. Discussion and Future Work

In this section we would like to address a couple of different topics that require discussion and point to potential future work.

7.1 Other Forms of Derivatives

Building on Brzozowski derivatives, it appeared natural to us to introduce derivatives (or *left-quotients*) as first-class feature. However, we think it is worthwhile to also explore other forms of quotienting:

While the right-derivative can be expressed as a derived combinator in our framework, it imposes performance penalties. However, in combination with the left-derivative the right-derivative could be useful for instance to select $expr$ from the production $\texttt{'\{'} \sim expr \sim \texttt{'\}'}$.

It is well-known that context-free languages are closed under quotienting with regular languages. Future work could explore the design space of adding first-class derivation by regular expression and the effects on the gained expressive power. Concatenation and alternative appear straightforward. However, we anticipate that an efficient support of deriving by Kleene-star will be more challenging.

7.2 Effect on the Language Class

While this paper establishes that first-class derivatives can be useful in practice to provide modular and compositional parser implementations, a theoretical question remains, that bears asking:

> Does extending a specification language for context-free grammars with derivatives affect the corresponding language class?

In particular, is such an extended grammar still context-free? While it is well-known that context free languages are closed under left- (and right quotienting), we do not know of a case where quotienting is considered as part of a grammar itself.

7.3 Indentation sensitivity

Indentation sensitivity itself cannot be expressed using context-free grammars. Nevertheless, there has been effort dedicated to implement parsers that recognize indentation and to extend grammar formalism in order to concisely express indentation sensitivity. However, existing solutions require a *ad hoc* modification of the lexer to track the state of indentation [17], specialized extensions to grammar formalisms [2], global transformations [3] or layout-constraint based post-processing of the parse forest [7]. The closest to a modular description of indentation sensitivity are data dependent parsing approaches [4]. Howevever, implemented as parser generators, users can only use the abstraction mechanisms provided by the grammar formalism and thus cannot abstract over indentation.

While being user-definable, modular and concise, the our implementation of $\mathsf{indented}(p)$ only supports a limited variant of indentation sensitivity and does not account for ignoring indentation in explicitly delimited blocks. Also, in a scannerless setting the lexical structure inside p will be ignored and thus indentation will be enforced in literals containing newlines (such as multiline strings). Ignoring indentation within delimited blocks could be implemented in our framework by adding additional states to the corresponding automaton to track whether the parser should be sensitive to indentation or not. However, respecting lexical structure would require additional communication between lexer and parser resulting again in a non-modular design.

7.4 Extensions and Improvements

Moss [16] gives a parsing algorithm for PEG based on derivatives. To this end it was necessary to also define the derivatives for a lookahead operator. This leads us to believe, that it is straightforward to extend our parser combinator library with support for (negative) lookahead. Complementing feed, negative lookahead would be useful to more concisely express the conditions under which a parser should delegate to the child-parser.

Finally, the performance of parsers in our library is not in the focus of this paper. Albeit, building on derivative based parsing our approach automatically benefits from improvements in that area, such as the optimizations proposed by Adams et al. [1].

8. Conclusion

We have shown that the semantic concept of a Brzozowski derivative of a parser can be internalized in the form of a novel parser combinator. We have seen that this parser combinator can improve the modularity and reusability of parsers in situations where the substream property is a problem. We have demonstrated the feasilibity of first-class derivatives by means of a parser combinator library in Scala and a small set of accompanying case studies.

References

[1] M. Adams, C. Hollenbeck, and M. Might. On the complexity and performance of parsing with derivatives. In *Proceedings of the Conference of Programming Language Design and Implementation*, 2016.

[2] M. D. Adams. Principled parsing for indentation-sensitive languages: revisiting landin's offside rule. *ACM SIGPLAN Notices*, 48(1):511–522, 2013.

[3] M. D. Adams and Ö. S. Ağacan. Indentation-sensitive parsing for parsec. In *ACM SIGPLAN Notices*, volume 49, pages 121–132, 2014.

[4] A. Afroozeh and A. Izmaylova. One parser to rule them all. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 151–170. ACM, 2015.

[5] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[6] N. A. Danielsson. Total parser combinators. In *ACM Sigplan Notices*, volume 45, pages 285–296. ACM, 2010.

[7] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-sensitive generalized parsing. In *Software Language Engineering*, pages 244–263. Springer, 2012.

[8] P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of the International Conference on Software Reuse*, pages 134–142. IEEE, 1998.

[9] G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of functional programming*, 8(04):437–444, 1998.

[10] T. Jim and Y. Mandelbaum. A new method for dependent parsing. In *Programming Languages and Systems*, pages 378–397. Springer, 2011.

[11] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. *ACM Sigplan Notices*, 45(1):417–430, 2010.

[12] O. Kiselyov. Iteratees. In *Functional and Logic Programming*, pages 166–181. Springer, 2012.

[13] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, Mar. 1966.

[14] P. Ljunglöf. Pure functional parsing-an advanced tutorial. 2002.

[15] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a functional pearl. In *ACM Sigplan Notices*, volume 46, pages 189–195. ACM, 2011.

[16] A. Moss. Derivatives of parsing expression grammars. *CoRR*, abs/1405.4841, 2014.

[17] Python Software Foundation. The Python language reference: Full grammar specification. `https://docs.python.org/3.5/reference/grammar.html`. Accessed: 2015-03-24.

[18] J. J. Rutten. *Automata and coinduction (an exercise in coalgebra)*. Springer, 1998.

[19] S. D. Swierstra. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*, pages 252–300. Springer, 2009.

[20] J. Winter, M. M. Bonsangue, and J. Rutten. *Context-free languages, coalgebraically*. Springer, 2011.