Parsing with First-Class Derivatives

Jonathan Immanuel Brachthäuser Tillmann Rendel Klaus Ostermann
University of Tübingen, Germany
{brachthaeuser, rendel, ostermann}@informatik.uni-tuebingen.de

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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; D.3.4 [*Programming Languages*]: Processors—Parsing; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems—Parsing; F.4.3 [*Mathematical Logic and Formal Languages*]: Formal Languages—Operations on languages

General Terms Algorithms, Languages

Keywords Parsing; Modularity; Derivative; Left Quotient; Parser Combinators

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 can be found in Haskell or Python 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 because these ad-hoc additions are cross-cutting with respect to the otherwise context-free syntactic structure of the language.

Parser combinators [10, 10, 15, 21, 22] 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, 17, 18]. 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. Similar to prior work we base the definition of our parsers on derivatives. In addition, we introduce first-class derivatives as parser combinators by exposing the derivation operator to the user. First-class derivatives effectively allow 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 a combinator for a simple variant of indentation sensitivity (Section 3).
- To evaluate expressivity, we have performed three additional case studies (beyond the indentation example). Section 4 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.
- To evaluate how our approach scales to realistic languages, Section 5 and Section 6 extend the indentation example to support more features of a language modeled after Python and illustrate how to overcome some limitations.
- To evaluate feasibility and support our case studies, we have implemented a prototype of a parser combinator library with support for first-class derivatives. In Section 7, 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.

We review closely related work in Section 8. In Section 9, we discuss limitations, future work, and phrase open questions. Section 10 concludes the paper.

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

2. Parser Combinator Libraries and the Substream Property

We base our studies on parser combinators, since they already offer a wide range of techniques to modularly define parsers which we account to their nature of being an embedded domain-specific language (EDSL) [8]: Both type- and term-level abstraction mechanisms can be reused to implement modular, reusable parsers. Parser developers can use host-language functions to express parsers which are parametrized by values of the host language. In particular, 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".

Despite their already good support for modular definitions, "traditional" parser combinator libraries, that is, libraries that do not support first-class derivatives, share a set of shortcommings that we will discuss in the remainder of this section. Later sections will illustrate how first-class derivatives can help to overcome these limitations.

To recognize a given input stream with a composite parser, 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*. By 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 it is possible to add symbols in the middle of a word. 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 of the original input stream because each line starts by the indentation whitespace that should be stripped out. We want 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 are tables represented in monospaced text which typically use dashes, pipes, and other characters to define the two-dimensional layout and separate their 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. Such newlines are not part of the original input stream.

[1] `https://github.com/b-studios/fcd`

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 our parser combinator library which allows fine-grained control over the input stream delegation to child parsers. By fusing stream-preprocessing and parsing, with our library, we can implement each of the above examples as a parser combinator in a separate module and combine them 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 nested tables that itself contain tables and indented code.

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, 20] as well as context-free languages [6, 17, 23]. 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. Our presentation follows the syntax of traditional parser combinators as found in the Scala standard library.

In 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 7.

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

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

```
// 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]        : (P[R], P[R])            => P[R]
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)]
def not[R]        : P[R]                    => P[Unit]
// 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.

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

(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.

function parse is universally quantified by R and so given a parser of type $\mathrm{P}[R]$ it will process the list of tokens to potentially return syntax trees of type R. If the input cannot be recognized the returned list will be empty.

For ease of presentation, unless noted otherwise 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.

We also include the traditional parser combinators seq, alt, and map. The parser 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 alt(p, q) is used to represent an alternative in a production. The parser map$(p)(f)$ allows applying the transformation function f as a semantic action to the syntax tree returned by p[3].

In addition to these combinators, that alone can be used to represent context-free grammars, we also include the monadic combinator flatMap, the intersection of two parsers and as well as negation of a parser not. The combinator flatMap (p) (f) allows one 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 tokens. The intersection and(p, q) of the two parsers p and q recognizes a word only if both parsers recognize it. The negation of a parser not(p) recognizes the complement of the language of parser p. Both, intersection and negation are rarely found in combinator libraries. This probably relates to the fact that the language of the resulting parsers is in general not context-free. However, as we will see in Section 7 supporting intersection and negation in a derivative-based implementation is straightforward. Later examples show that both combinators are useful in our framework.

3.2 First-Class Derivatives

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

The combinator feed(p, c) represents the core contribution of this paper. It derives the parser p by the given token c. 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, 17, 18]. 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 of syntax trees.

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 the 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[4]. 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: $\mathrm{P}[\mathrm{Int}]$:

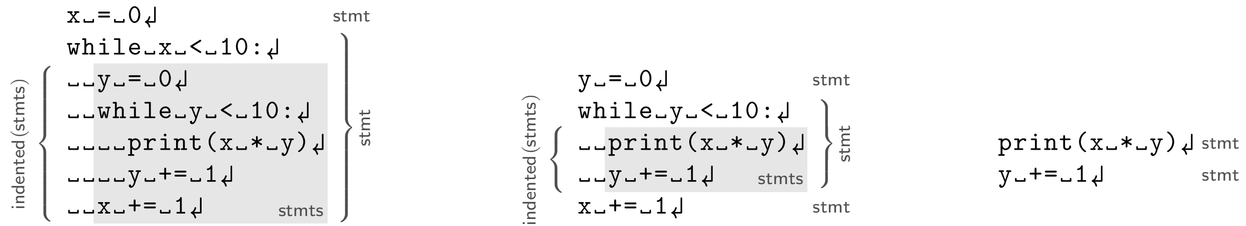
```
val number: P[Int] =
  nt ( map(seq(number, digit)) {
         case (n, d) ⇒ (n * 10) + d
       }
     | digit
     )
```

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 parser 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.

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 ($p \sim\!\rangle\, q$ and $p \,\langle\!\sim q$) and

[3] The arguments to the parser combinators map and flatMap are curried, since Scala offers better type inference on curried functions.

[4] 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.

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 these abbreviations, we can define the parser number again by:

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

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. Later, Section 5 and Section 6 will further improve the indentation combinator.

3.3 Indentation-Sensitive Parsing

Using indentation to indicate block structure goes back to Landin's "offside rule" [14], which, in variations, is 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 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 we for now assume to be 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.

```
val stmt: P[Stmt] =
  ( ("while" ~› expr ‹~ ':') ~ block ^^ {
      case(e, b) ⇒ new WhileStmt(e, b)
    }
  | ...
  )
val expr  : P[Stmt]        = ...
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

On the top level, the program consists of two statements, an assignment, and a while-statement. Interestingly, in order to recognize the body of the while-statement, indented(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. Second, it needs to invoke the body-parser stmts with the contents of the indented block (highlighted in gray). 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.

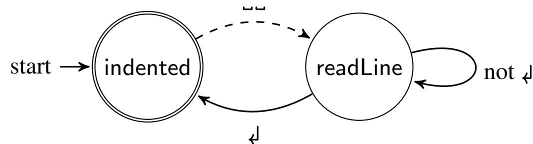
The virtual stream which is to be recognized by the indentation agnostic body-parser stmts is shown in Figure 2b. We can see that the virtual stream consists of three statements, where the second one is again a while-statement extracting two chunks of its virtual stream, "`print(x * y)↲`" and "`y +=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

Finally, these 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.

```
def indented[T](p: P[T]): P[T] =
  done(p) | (space ~ space) ~> readLine(p)

def readLine[T](p: P[T]): P[T] =
  ( no('\n') >> { c => readLine(p << c) }
  | '\n'     >> { c => indented(p << c) }
  )
```

(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.

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, 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 pseudocode:

$$\text{any} \gg \{\, c \Rightarrow ...p \ll c... \}$$

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 this threading of the child parser. We also refer to this child parser as the *delegatee*.

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. It models indentation just in terms of spaces and newlines. Section 6 discusses modularity issues that arise from interpretation of spaces and newlines that depend on the lexical or syntactical context.

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 refers to the process of forwarding the input stream to some other parser but not giving up control. The delegation can be suspended and the partially applied parser can be extracted to eventually resume delegation later. 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 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:

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

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 refers to repeatedly delegating fragments of the input stream to a parser and is implemented by the second derived combinator repeat$(f)(p)$. The surrounding "context" of the delegated fragment is expressed using the function f which eventually constructs a (delimited) delegation. The combinator 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.

```
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.

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 the next section.

4. Applications

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.

4.1 Increased Reuse 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 facilitate the reuse and later extension of parsers. 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 ⋘ "while"`. The operator $\lll$ is defined by lifting $\ll$ from one token 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:

```
val untilStmt = "until" ~> (stmt ⋘ "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. These are 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 select a parser for the fractional digits deriving expr by "`0.`".

4.2 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:

```
def unescape[R](p: P[R]): P[R] =
  ( '\\' ~> any ≫ { c ⇒ unescape(p ≪ unescChar(c)) }
  | no('\\')     ≫ { c ⇒ unescape(p) }
  )
```

The function 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;

```
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, ps2) =>
    (p ~ ps2) ^^ { case (r, rs) => r :: rs }
  }

def collect[T](ps: List[P[T]]): P[List[T]] =
  ps.foldRight(succeed(Nil)) { case (p, ps2) =>
    done(p) >> { r => ps2 ^^(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 6. Parser combinators for additional case studies illustrating delegation to more than one parser.

and one for textual contents. Similar to fenced code blocks in markdown, the code fragments are 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 6a 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. 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 contents 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 $\text{List}(6, 14, 6)$). 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 of the input, we parse the initial pipe and delegate six tokens to the first cell parser. Interestingly, on the 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 6c). To handle administrative details of delegating to a list of parsers it uses two new derived combinators defined in Figure 6b. The combinator distr(ps) takes a list of parsers and sequences them, it thus distributes parsers over lists. The combinator collect (ps) behaves like done(p) but lifted to a list of parsers. It is only successful, 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 continues with the next line.

```
val line      = many(no('\n')) ~ '\n'
val emptyLine = many(space)    ~ '\n'

def indentBy[T](i: P[Any]): P[T] ⇒ P[T] = repeat[T] { p ⇒
  ( emptyLine ^^ { _ ⇒ p ≪ '\n' }
  | i ~> (line &> delegate(p))
  )
}
def indented[T](p: P[T]): P[T] =
  some(space) ≫ { i ⇒ indentBy(acceptSeq(i))(p) ⋘ s }
```

(a) Definition of an improved version of the combinator indented (p).

```
val always = many(any)
def biasedAlt[T](p: P[T], q: P[T]): P[T] =
  p | not(p ~ always) &> q
def indentBy[T](i: P[Any]): P[T] ⇒ P[T] = repeat[T] { p ⇒
  biasedAlt( emptyLine ^^ { _ ⇒ p ≪ '\n' },
             i ~> (line &> delegate(p)) )
}
def indented[T](p: P[T]): P[T] = some(space) ≫ { i ⇒
  not(space ~ always) &> ( indentBy(acceptSeq(i))(p) ⋘ s )
}
```

(b) Further improving the combinator indented (p) by reducing potential for ambiguities. Differences to Figure 7a are highlighted in *gray*.

Figure 7. Improving combinator indented (p).

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 combinators, 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.

5. Improved Indentation Sensitivity

In the previous sections, we have seen how first-class derivatives can be used to define parser combinators for several different layout-sensitive features. The implementation of the indentation combinator in Section 3 shows how first-class derivatives can encode non-context-free constructs in a modular way, but it is not as sophisticated as real languages with indentation-sensitive syntax. Does our approach to parse indentation sensitivity based on first-class derivatives and delimited delegation scale to the requirements of realistic languages such as Python? To further evaluate the expressiveness of first-class derivatives, in this and the following section we gradually improve the implementation of indentation sensitivity to support a larger subset of the Python language.

The development takes place in two steps: In this section, we show how two simple extensions can be implemented by modification of the indentation combinator. These modifications give rise to a discussion on how to implement lookahead and left biased alternative in our framework. In the next section, we then propose two different strategies to implement line joining that exhibit different modularity properties.

Our experiments suggest that it is possible to define indentation sensitivity and line joining for Python parsers as (modular) combinators in our framework. However, since the communication between the different combinators is achieved by modification of the input stream, doing so requires careful consideration of the possible "side-effects" on other involved combinators. The definition of line joining does not only require the preprocessing to depend on the context (which is well supported by our framework) but also on the lexical structure (which is less clear how to support in a modular way). Future work should investigate the integration of lexical analysis and first-class derivatives to add support for this form of dependency.

5.1 Improving the Indentation Combinator

The combinator indented as defined in the introductory example in Figure 5b implements a much simpler treatment of indentation than what can be found in most indentation-based programming languages. For instance, it only supports a fixed indentation of two spaces at the beginning of every indented line. It also requires empty lines to be indented correctly. However, according to the Python specification [19], empty lines should be ignored and hence do not need to be indented correctly. Also, the indentation of a block should be determined by the indentation of the first line of this block. Implementing these requirements exposes a limitation of our library: Lookahead and biased alternative cannot be expressed as user defined combinators. To work around this issue we use a combination of prefix-checking, based on negation and intersection, as well as rewriting of the grammar when necessary.

In a first step, Figure 7a defines an improved version of the indentation combinator that infers the indentation from the first line of a block and supports empty lines. Compared to the previous version, the implementation is split into two different combinators.

The combinator indentBy (i) (p) recognizes a block p where the indentation of every line is represented by the parser i. By repeatedly either recognizing an empty line or a line that is indented by i, the combinator indentBy accounts for the first requirement of ignoring the indentation of empty lines.

To recognize a block with arbitrary indentation, the combinator indented first processes multiple spaces. The resulting list of space tokens represents the indentation of the first line and is used in two ways. First, a parser that recognizes exactly the same indentation in form of a sequence of the space tokens ($\mathsf{acceptSeq}(i)$) is passed as the first argument to the combinator indentBy. Second, the spaces are fed back to the parser that recognizes the indented block. This is an instance of a pattern to simulate a form of non-consuming lookahead.

5.2 Lookahead and Biased Alternative

Unfortunately, the implementation of the indented combinator in Figure 7a bears potential for ambiguities: If a parser p_s accepts whitespace at the beginning of a line and the input contains a block indented by n spaces, there are n different ways to recognize the block with $\mathsf{indented}(p_s)$. In particular, if a line of the block, other than the first line, is indented by $1 \leqslant m < n$ spaces, by backtracking of the sequence parser, the block can be recognized as indented by m spaces. For parsing a Python-like language this is clearly wrong. Instead, we expect the parser some(space) in the indented combinator to consume as many spaces as possible before checking indentation with indentBy. To make the combinator indented robust against parsers like p_s, we simulate a greedy behavior for some (space) by making sure it is not immediately followed by another space. While traditional parser combinator libraries support lookahead for this purpose, in our framework we can use intersection and negation to assert that the result of the combinator indentBy does not recognize a space as prefix after consuming the initial indentation. Figure 7b redefines indented and incorporates checking for the absence of the prefix. In general, for two parsers p and q we can identify the pattern

$$\mathsf{not}(p \sim \mathsf{always})\ \&\rangle\ q$$

to express "q without prefixes p". Here, the parser always is defined by many(any) and is used to implement checking for a prefix instead of an exact match for p.

We can also identify a second source of ambiguity in the implementation in Figure 7a. Correctly indented empty lines are recognized by both alternatives in indentBy. To prevent this ambiguity we present an implementation of left-biased alternative $\mathsf{biasedAlt}(p, q)$ in Figure 7b. By checking for the absence of a prefix p in the case of q we thus prefer the first alternative in case of an ambiguity.

Intersecting with a prefix is a workaround for the lack of built-in support for lookahead in our framework. But is this workaround sufficient to fully eliminate the need for built-in (non-consuming) lookahead as known from traditional parser combinator libraries? In these libraries, if negative lookahead ($p\ !\ q$, that is, "p not followed by q") occurs in a context with the next parser in a sequence being k (such as $(p\ !\ q) \sim k$) all the tokens that q uses are not consumed and instead will be passed to k. However, the translation to prefix checking as $p \sim \mathsf{not}(q \sim \mathsf{always})\ \&\rangle\ k$ is only correct in cases where k either represents the full continuation parser (the parser that processes all of the remaining input stream) or if no word recognized by k is a prefix of a word recognized by q. To circumvent this problem, we can inline more of the continuation and rewrite to $\mathsf{not}(q \sim \mathsf{always})\ \&\rangle\ (k \sim k')$. Eventually, this rewriting might lead to a global transformation of the grammar. Thus, our library is not expressive enough to implement lookahead and biased choice as user-defined combinators. However, in some cases, such as for the combinators in this and the next section, prefix checking and local rewriting of the grammar is sufficient as a workaround.

6. Line Joining

Along the concept of the indentation sensitivity, the Python specification distinguishes between *logical lines* and *physical lines*. Indentation checking is only applied to logical lines. A logical line can consist of multiple physical lines according to the following *line joining* rules:

1. linebreaks in strings that span multiple lines (also called "long strings" or "multiline strings") do not separate logical lines.
2. linebreaks preceded by a backslash character are escaped and do not separate logical lines (explicit line joining).
3. linebreaks in expressions that are enclosed in parenthesis, brackets or curly braces do not separate logical lines (implicit line joining).

In the remainder of this section we describe two attempts to implement line joining modularly as separate parser combinators, but exclude the handling of source code comments for ease of presentation. While implementing parser combinators that also deal with comments requires some care, it does not introduce a new set of problems.

All examples in this paper so far were presented in a scannerless style, but parsing with first-class derivatives is generally independent of the existence of a separate lexer phase. Thus, first we show how to implement line joining in the presence of a separate lexer phase. Then we illustrate what it takes to achieve the same in a scannerless setting. We defer a discussion of the limitations of both approaches to Section 9.

6.1 Line Joining with a Separate Lexer

The Python specification is written with a separate lexer phase in mind and thus it is not surprising to see that the line joining rules are not difficult to express in such a setting. We can use first-class derivatives with a separate lexer phase by defining the type Elem to be an appropriate type of lexemes. Parser combinators that do not match particular elements of the input stream can just be reused without changes in combination with a lexer. All others parsers, including indented, need to be modified to match the corresponding tokens instead

```
def repEl[T](f: Elem ⇒ P[T] ⇒ P[T]) =
  repeat[T] { p ⇒ any ^^ { el ⇒ f(el)(p) } }
def mapEl[T](f: Elem ⇒ Elem): P[T] ⇒ P[T] =
  repEl(el ⇒ p ⇒ p ≪ f(el))
def filterIn[T](allowed: List[Elem]): P[T] ⇒ P[T] =
  repEl(p ⇒ el ⇒ if (allowed contains el) p ≪ el else p)
```

(a) Derived parser combinators for preprocessing the input stream of a given parser.

```
def nlToWs[T] = mapEl[T] { c ⇒ if (c ≡ \n) ␣ else c }
val dyck = '(' ~ many(dyck) ~ ')'
val parens = ('(' ~ always ~ ')') &>
              filterIn(List('(', ')'))(dyck)
def ilj[T] = repeat[T] { parens &> nlToWs(delegate(p))
                       | no('(', ')') &> delegate(p)
                       }
def joiningIndented[T](p: P[T]): P[T] = ilj(indented(p))
```

(b) Implementation of implicit line joining as a parser combinator that filters newlines inside a set of balanced parenthesis.

Figure 8. Line joining in presence of a separate lexer phase.

of characters. This could be avoided by parametrization of indented over line ends and spaces.

For the purpose of parsing an indentation-sensitive language, we assume a simple lexer, that is, one implementable by a finite state automaton, with the following informal specification: besides the basic lexemes (such as literals, identifiers, (multiline) strings, punctuation etc.) the lexer also emits tokens for every whitespace, comment, and newline. Hence, the output of the lexer for an example input stream "`␣␣x+\\\n(\n'''\n)''')\n`" will be:

`␣` `␣` `'x'` `'+'` `\\\n` `'('` `\n` `STR` `')'` `\n`

This example input stream requires the application of all three line joining rules. However, the rule for line joining within multiline strings is already implemented in the lexer by grouping all input enclosed by `'''` into one string token. At the same time, the lexer might output a `\\\n` token for every escaped newline and explicit line joining can simply be implemented as a combinator filtering these tokens. Consequently, only the definition for implicit line joining is given in Figure 8b using utility combinators as defined in Figure 8a. Implicit line joining is implemented as combinator ilj(p) by removing all layout-irrelevant newlines within a pair of parenthesis. It performs a tokenwise scan over the input stream, delegating all tokens unless inside a region delimited by balanced parenthesis. There, additional filtering of the input stream with nlToWs(p) is applied.

The definition of the delimiter region parens is implemented by extending the parser dyck for the Dyck language of well-balanced parenthesis[5] to ignore all tokens other than parenthesis. The intersection with `'('` ~ always ~ `')'` bears repetition, but is necessary to restrict the scope of ignoring tokens to the inside of a pair of parenthesis.

When parsing the above-mentioned token stream with joiningIndented, the input to indented will be processed to

`␣` `␣` `'x'` `'+'` `\\\n` `'('` `STR` `')'` `\n`

[5] We only consider one sort of parenthesis here, which can easily be extended to any number of different parenthesis.

and appears as a single logical line. As before, the initial indentation ("`␣` `␣`") will be stripped by indented and in effect p only has to recognize the remainder of the processed input stream.

6.2 Scannerless Line Joining

The implementation of line joining in Figure 8b is straightforward. However, if lexing is external to our framework the composition of different layout features like indentation, tables, and mixed documents would require a composition of the involved lexers. To avoid this problem, we continue the development and explore how to implement line joining in a scannerless setting. We start by discussing an implementation of line joining for multiline strings which already exhibits most of the difficulties that arise when trying to implement also explicit line joining and implicit line joining.

The combinator indented (p) is implemented using the delimiter line and thus preprocesses the input to p on a line-by-line basis. In a scannerless setting, this is also true for string literals that span multiple lines. Every line of the string literal is checked for correct indentation and then the indentation is stripped before delegation. However, similar to the distinction between logical and physical lines, we should distinguish indentation-relevant newline tokens from irrelevant ones. Newline tokens within multiline strings should not constitute a line end from the point of view of the delimiter line.

To ignore these newlines during indentation checking and at the same time preserve them as input to the underlying parser, we define a preprocessing parser combinator mlj(p) that replaces every newline token enclosed in a multiline string by a special marker token $\hookleftarrow$. We assume that this marker does not occur in the original input stream. Marked newlines can be unmasked again, after checking indentation. This way indentation-irrelevant newline tokens can be ignored without completely deleting them from the input stream. For some parser p we thus can define indentation that respects multiline strings by:

```
mlj(indented(unmask(p)))
```

```
def within[T](    region: P[Any],
                  skip: P[Any],
                  f: P[P[T]] ⇒ P[P[T]]): P[T] ⇒ P[T] =
  p ⇒ done(p) | biasedAlt(
        ( skip   &›  delegate(p)
        | region &› f(delegate(p))
        ) ≫ within(region, skip, f),
      (any &› delegate(p)) ≫ within(region, skip, f))
```

(a) Parser combinator that allows region based preprocessing of the input stream.

```
def mask[T]    = mapEl[T] { c ⇒ if (c ≡ '\n')  ↩  else c }
def unmask[T]  = mapEl[T] { c ⇒ if (c ≡ ↩)   '\n' else c }
def constWs[T] = mapEl[T] { c ⇒ ' ' }

val string  = single | multi
val parens  = ('(' ~ always ~ ')') &›
  within(string | no('(') & no(')'), fail, filterIn(Nil))(dyck)

def mlj[T] = within[T](multi,   single, mask)
def elj[T]  = within[T]("\\\n", string, constWs)
def ilj[T]  = within[T](parens,  string, nlToWs)
```

(b) Implementing of line joining by approximation of the lexical structure. Definitions for single and multi are omitted. Parser dyck and combinator nlToWs are defined by straightforward translation from Figure 8b.

Figure 9. Implementation of line joining in a scannerless setting.

Given the example input above, the indentation combinator has to process "␣␣x+\\\n(\n''' ↩)''')\n" while the input to p will be "x+\\\n(\n'''\n)''')\n".

To implement mlj it is necessary to determine whether a newline is enclosed in a multiline string or not. This decision is dependent on the lexical structure. In a scannerless setting, however, recognizing lexemes like multiline strings is typically part of the parser for expressions and thus only performed during parsing. In particular, the various parsers for lexemes are part of the internal structure of the delegatee p. At the same time, our framework does not permit any communication of a delegatee to the delegating parser, except for the result value after a successful parse. Hence, it is not possible for a delegating parser to inspect the state of the delegatee to obtain information about the current position in the lexical or syntactical structure. This leads us to believe that in our framework and in a scannerless setting, it is not possible to implement the combinator mlj(p) fully independent of the parser p. The implementation of mlj has to repeat just enough of the lexical processing to make its delegation decisions.

To work around this lack of communication, Figure 9a defines the combinator within $(region, skip, f)$ which we use to approximate the lexical structure without having to implement a full lexer. The result of within is again a parser combinator that given a parser p similar to ilj in Figure 8b performs a linear scan of the input for the specified *region* while skipping the scan within regions specified by *skip*. All input processed by the combinator will be delegated to p, except within a *region*. There, the parser transformer f is applied before delegation. The body of the combinator within is the result of inlining the continuation parser (as described in Section 5.1) and may be easier to comprehend before the transformation:

```
repeat[T] { p ⇒ biasedAlt(
  skip &› delegate(p) | region &› f(delegate(p)),
  any &› delegate(p))
}
```

The use of biased alternative simulates the greedy behavior of a lexer and assures that within will only continue its scan by skipping a single token with any &› delegate(p) if it is not possible to skip or to transform a region.

Using within and mask we can define mlj (p) as in Figure 9b. Applying mlj to a parser p will mask all the newlines within a multiline string. It will skip over single-line strings to account for the interaction of single-line strings and multiline strings as in "'''"\n'''.

Similar to line joining within multiline strings, explicit line joining is implemented as combinator elj (p). The newline characters of explicitly joined lines are replaced by spaces since they might separate two tokens. Explicitly joined lines within strings are ignored.

The definition of implicit line joining in Figure 9b is very similar to the one in Figure 8b. The parser for the Dyck language dyck is again extended to ignore all tokens other than parentheses, but now also needs to ignore parentheses that are contained in strings.

Finally, the line joining combinators as well as the combinator indented can be combined to:

```
def joiningIndented[T](p: P[T]): P[T] =
  elj(ilj(mlj(indented(unmask(p)))))
```

For the above example, the input to indented will be preprocessed to a single logical line "␣␣x+␣(␣''' ↩)''')\n".

The concept of line joining as preprocessing of the input stream is well supported by first-class derivatives. However, the selection of indentation irrelevant newlines is dependent on the lexical structure and thus demands more communication with the delegatee.

7. 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. We defer a comparison of our implementation with the one of Might et al. [17] to Section 8.

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 exist, 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 $D_b(\mathcal{L}) = \emptyset$.

7.1 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[6]:

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

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 $p.\mathrm{derive}(c)$ the *derivative* of parser p by the character c. We will also use Scala's support for infix notation and write $(p \text{ derive } c)$.

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.

7.2 Derivative-Based Implementation of Parser Combinators

In Figure 1a (Section 3) 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 10a, that, when used recursively, can recognize the class of context-free languages[7]. For all combinators (except fail which does not take arguments), we use anonymous functions to implement the interface of Figure 1a.

The implementation of the parser combinators that can express context-free languages is standard for derivative based parsing [17]. 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. The implementation of the combinator map$(p)(f)$ uses the method map defined on Scala collections to transform the results. The results of the combinator seq are defined as the cartesian product of the results of p and q, using Scala's syntax for for-comprehensions. Consequently, 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 the combinator seq 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.

Figure 10b gives the implementation of the intersection of two parsers and(p, q), the negation of a parser not(p) and the monadic combinator flatMap. Similar to alt, the intersection

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

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

```
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] = (p, q) ⇒ new P[R] {
  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
}
```

(a) 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 not[R] = p ⇒ new P[Unit] {
  def results = if (p.results ≡ Nil) List(()) else Nil
  def derive  = el ⇒ not(p 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)
}
```

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

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

(c) Implementation of our new combinator feed.

Figure 10. Implementation of our combinator library, defined in terms of derivatives.

just derives both parsers p and q in parallel but like seq returns the cartesian product of the results. The negation returns a result (the Scala unit value) in case p has no result and the empty list in case p would succeed. The result of the combinator 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 10c defines our new parser combinator feed (p, el) simply as a user available alias for the internal method derive.

7.3 Nonterminals

Without support for recursive definitions, the combinators in Figure 10a 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 us 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 11. For one, it guards the construction of the parser-graph by being lazy in its argument p and thus allows the creation of cyclic structures in memory[8]. For instance, omitting the combinator `nt` in

```
val as: P[Any] = nt(as ~ 'a' | 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:

```
def derive = el ⇒
  memo.getOrElseUpdate(el, nt(p derive el))
```

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:

$$\mathsf{as}_2 = \mathsf{nt}(\mathsf{as}_2 \sim \texttt{'a'} \mid \mathsf{succeed}(((), \texttt{'a'})))$$

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

```
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 11. Implementation of the combinator nt, definition of attribute and empty omitted.

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

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 [17]. Due to the potentially cyclic structure, the computation of p.results might again involve the computation of the results on the nonterminal-parser itself.

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

The implementation of memoization in Figure 11 slightly differs from the first attempt presented above. The modifications are necessary to avoid divergence with exotic parser like the following:

```
val exotic: P[Any] = nt(exotic ≪ 'a')
```

To compute exotic.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.empty to obtain a conservative approximation of whether the parser p only recognizes the empty language[9]. However, computing p.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[10].

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

7.4 Compaction by Dynamic Dispatch

Applying equivalences like

$$p \mid \mathsf{fail} = \mathsf{fail} \mid p = p \quad (1)$$

$$p \sim \mathsf{fail} = \mathsf{fail} \sim p = \mathsf{fail} \quad (2)$$

in a directed way can lead to a significantly cut down of the parser-graph and in effect can improve the performance of parsing [1, 17]. 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 12 illustrates how the equivalence from Equation 2 can be implemented as compaction rule using an encoding of double dispatch. The implementation of combinators just forwards 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 the 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

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

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

done(fail) = 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.

8. 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 the basis for their implementation. While the existing 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. Might et al. [17] 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 [18] 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 worst-case complexity. They propose optimizations and further compaction rules that they claim to make derivative based parsing performant enough to be used in practice.

Of the work on parsing with derivatives, our implementation is most closely related to prior work by Might et al. [17] which requires some highlighting of 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 nonterminals, 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. In this way, our implementation can more easily be applied in languages where encoding laziness can be cumbersome (as in Java). At the same time, by 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, not, flatMap and expose derivatives to the user in form of the 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.

Data-dependent grammars. Data-dependent grammars [4, 11, 12] 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-sensitivity 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 [13] 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 on-line 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.

9. Discussion and Future Work

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

9.1 Indentation sensitivity

Indentation sensitivity itself cannot be expressed using context-free grammars. Nevertheless, there has been effort dedicated to implementing parsers that recognize indentation and to extend grammar formalisms in order to concisely express indentation sensitivity. However, to the best of our knowledge existing solutions require an *ad hoc* modification of the lexer to track the state of indentation [9, 19], 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].

However, implemented as parser generators, users can only use the abstraction mechanisms provided by the grammar formalism and thus cannot abstract over indentation.

The technique for parsing layout presented in this paper is based on first-class derivatives and delimited delegation. Defining layout as parser combinators enables composition and naturally scales to the recursive case. However, first-class derivatives exhibit limits as soon as the delegation depends on the delegatee. This is a paradigm currently not well supported. In the case of indentation with line joining, recognizing the delimiter (logical lines) requires knowledge about the lexical structure of the delegatee. Section 6 presented two different workarounds that reveal different drawbacks.

A separate lexer phase removes the dependency of the delegating parser (indented) to the delegatee by providing the necessary information in the structure of the token stream. However, a separate lexer will be specialized to one particular parser and hence composing parsers (a strength of scannerless parser combinator libraries) also requires composing the lexers.

A possible solution to this problem would be to interpret lexers as delegating parsers in our framework. In the end, a lexer is nothing more than a stream preprocessor that adds additional information and structure to the input stream. Abstracting over the type of elements in the input stream

$$
\begin{array}{l}
\textbf{trait}\ \mathrm{P}[-\mathrm{Elem}, +R]\ \{ \\
\quad \textbf{def}\ \mathrm{results}\colon \mathrm{Res}[R] \\
\quad \textbf{def}\ \mathrm{derive}\colon \mathrm{Elem} \Rightarrow \mathrm{P}[\mathrm{Elem}, R] \\
\}
\end{array}
$$

would allow to perform local lexing. For instance in a mixed document, a Python lexer would only be applied inside of a code section while preserving the advantages as described in Section 4. With such a generalization, the type of a Python lexer and parser would be:

$$
\begin{array}{ll}
\textbf{val}\ \mathsf{python}\colon & \mathrm{P}[\mathrm{Char}, AST] = \mathsf{pyLex}(\mathsf{pyProg}) \\
\textbf{val}\ \mathsf{pyProg}\colon & \mathrm{P}[Lex, AST] = ... \\
\textbf{def}\ \mathsf{pyLex}[T]\colon & \mathrm{P}[Lex, T] \Rightarrow \mathrm{P}[\mathrm{Char}, T] = ...
\end{array}
$$

In addition, for instance, it would be possible to "unlex" python comments and delegate the contents to a separate parser that reuses the table combinator as it is defined in Section 4. It is left to future work to explore the expressiveness of combining local lexing with first-class derivatives.

Approximating the lexical structure, the implementation in Figure 9 works around the limitation by repeating parts of the lexer in the definition of the delegating combinators. This however comes with a few drawbacks.

Firstly, the solution is not fully modular, since combinators now depend on the definition of lexemes used in the delegatee. Thus, adding a new lexeme (such as a comment) to the implementation of the delegatee might require adaption of the combinators. In fact, to also handle comments all three combinators mlj, elj and ilj need adaption to account for the lexical structure. In addition, similar to how ilj handles newlines it requires modification to translate comments into whitespace. Secondly, the solution of lexical approximation is fragile and might interact with the delegation and preprocessing. After recognizing a region as a lexeme, the preprocessing of that region might change whether it still can be recognized as a lexeme within the delegatee.Thirdly, the implementation bears repeated computation since (approximate) lexing is performed multiple times.

In summary, first-class derivatives work best for delegating parser combinators where the layout parsers that delimit the scope of delegation can be expressed independently of the delegatee (such as line for indentation, cellrow for tables, coderegion for fenced code blocks). Future research could investigate to make these dependencies explicit by allowing additional communication from a delegatee to the delegating parser.

9.2 Indentation Sensitivity in Haskell

The programming language Haskell is another well-known example of a language with an indentation-sensitive grammar. The Haskell Report [16] describes the so-called "layout rule" informally as a process of inserting additional opening brace, semicolon, and closing brace characters into the token stream. Similarly to the rules for Python discussed above, layout processing is suspended inside explicit pairs of braces, so a Haskell parser in our framework would require similar mechanisms to detect and handle balanced pairs of braces outside comments, and so on.

However, layout processing in Haskell is not only based on determining indentation and counting balanced pairs of braces but also interacts with the grammar as follows: "A close brace is also inserted whenever the syntactic category containing the layout list ends; that is, if an illegal lexeme is encountered at a point where a close brace would be legal, a close brace is inserted" [16, Sec. 2.7]. The notion of "illegal lexeme" is further clarified in the formal definition of the layout rule as a function L given in the Haskell Report, which inserts a "}" token "if the tokens generated so far by L together with the next token t represent an invalid prefix of the Haskell grammar, and the tokens generated so far by L followed by the token '}' represent a valid prefix of the Haskell grammar" [16, Sec. 10.3]. A direct implementation of this rule seems to require strong coupling of the lexer and the parser, since either the lexer (or the lexical approximation) need to decide whether a token stream is a valid prefix of the Haskell grammar and thereby duplicate the work of the parser, or they need a non-standard communication channel with the parser to collaborate in the layout processing. It is an open question whether our framework can implement these approaches in a modular way.

Both Erdweg et al. [7] and Adams [2] describe indentation-sensitive grammar formalisms where constraints about the indentation of sub-phrases can be annotated to grammar productions in order to express indentation-sensitive syntax

in a declarative way. These approaches are implemented by generation of a generalized LR-parser and checking the layout constraints after parsing or while parsing. Since the layout constraints are part of the grammar, the dependence of Haskell layout processing on Haskell parsing is no longer problematic in these approaches. It would be interesting to explore how a similar approach could be encoded in our framework and what its implications on performance and modularity would be.

9.3 Lookahead

Our case study in Section 5 suggests that even with the powerful (and non-context-free) features of negation and intersection it remains difficult to express lookahead modularly. We track this difficulties to our choice of basing our library on parsing with derivatives: In the framework of traditional parser combinator libraries with a definition of a parser similar to

$$\mathrm{P}[T] = \mathrm{List}[\mathrm{Elem}] \Rightarrow (\mathrm{Res}[T], \mathrm{List}[\mathrm{Elem}])$$

it appears natural to add a lookahead parser combinator that inspects the remainder of the input stream to decide whether to accept a word or not without actually consuming input. In contrast, to allow delegation and the resulting inversion of control our library is based on parsing with derivatives with a definition of a parser similar to:

$$\mathrm{P}[T] = (\mathrm{Res}[T], \mathrm{Elem} \Rightarrow \mathrm{P}[T])$$

In this paper, we set out to show that it is natural to add parser combinators that use two properties of parsers based on derivatives. Firstly, derivation is push-based and thus driven from the outside. This allows us to invert the control flow and offer parser combinators that preprocess the input to other parsers. Secondly, a parser derived by parts of the input stream is again a parser. Having a first-class representation of partially applied parsers allows us, in particular, to suspend and resume delegation to parsers.

Where with lookahead (and similar features motivated by the type of traditional parser combinators) it is possible to modularly describe what should happen with the (output) stream after a parser has processed parts of it, in our setting with delegation (and similar features motivated by the type of derivative-based parser combinators) it is possible to modularly describe what should happen with fragments of the (input) stream before a parser may process it.

Existing work on derivative-based implementations of parsing expression grammars [18] suggests that adding full support for lookahead is possible. Still, additional research is necessary to fully combine and leverage the modularity benefits implied by traditional and derivative-based parser combinators.

9.4 Other Forms of Derivatives

We introduced derivatives (or *left-quotients*) as first-class feature. We think it is worthwhile to also explore other forms of quotienting such as the following two:

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 `'{'` $\sim$ `expr` $\sim$ `'}'`.

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.

9.5 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:

> 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.

9.6 Performance

Finally, the performance of parsers implemented using our library is not in the scope of this paper which focuses on modularity. Albeit, building on derivative-based parsing the implementation of basic combinators automatically benefits from improvements in that area, such as the optimizations proposed by Adams et al. [1]. Since our library includes parser combinators like `flatMap`, intersection and negation it is not possible to guarantee cubic bounds on complexity in general. More research is necessary to investigate efficient ways to incorporate these non-context-free combinators into the framework of derivative based parsing. First experiments on using laws of boolean algebra as compaction rules suggest that this is a viable path for future improvements.

10. Conclusion

We have shown that the semantic concept of Brzozowski derivatives 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 feasibility of first-class derivatives by means of a parser combinator library in Scala and a small set of accompanying case studies.

Acknowledgments

We would like to thank the anonymous reviewers for their comments that helped improve the paper. This work was supported by DFG project OS 293/3-1.

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. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 511–522, New York, NY, USA, 2013. ACM.

[3] M. D. Adams and Ö. S. Ağacan. Indentation-sensitive parsing for Parsec. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 121–132, New York, NY, USA, Sept. 2014. ACM.

[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 *Proceedings of the International Conference on Functional Programming*, ICFP, pages 285–296, New York, NY, USA, 2010. ACM.

[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. Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, 7 1992.

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

[11] T. Jim and Y. Mandelbaum. A new method for dependent parsing. In *Proceedings of the Programming Languages and Systems: European Symposium on Programming*, ESOP, pages 378–397. Springer, 2011.

[12] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 417–430, New York, NY, USA, 2010. ACM.

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

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

[15] P. Ljunglöf. Pure functional parsing-an advanced tutorial. *Licenciate thesis, Göteborg University and Chalmers University of Technology, Gothenburg, Sweden*, 2002.

[16] S. Marlow (editor). Haskell 2010 language report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.

[17] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: A functional pearl. In *Proceedings of the International Conference on Functional Programming*, ICFP, pages 189–195, New York, NY, USA, 2011. ACM.

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

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

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

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

[22] P. Wadler. How to replace failure by a list of successes. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

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