Zippy LL(1) Parsing with Derivatives

Romain Edelmann  
IC, EPFL  
Lausanne, Vaud, Switzerland  
romain.edelmann@epfl.ch

Jad Hamza  
IC, EPFL  
Lausanne, Vaud, Switzerland  
jad.hamza@epfl.ch

Viktor Kunčak  
IC, EPFL  
Lausanne, Vaud, Switzerland  
viktor.kuncak@epfl.ch

Abstract

In this paper, we present an efficient, functional, and formally verified parsing algorithm for LL(1) context-free expressions based on the concept of derivatives of formal languages. Parsing with derivatives is an elegant parsing technique, which, in the general case, suffers from cubic worst-case time complexity and slow performance in practice. We specialise the parsing with derivatives algorithm to LL(1) context-free expressions, where alternatives can be chosen given a single token of lookahead. We formalise the notion of LL(1) expressions and show how to efficiently check the LL(1) property. Next, we present a novel linear-time parsing with derivatives algorithm for LL(1) expressions operating on a zipper-inspired data structure. We prove the algorithm correct in Coq and present an implementation as a part of Scallion, a parser combinators framework in Scala with enumeration and pretty printing capabilities.

CCS Concepts: • Software and its engineering → Parsers;  
• Theory of computation → Grammars and context-free languages; Logic and verification; Design and analysis of algorithms.

Keywords: Parsing, LL(1), Derivatives, Zipper, Formal proof

1 Introduction

In this paper, we propose a formally verified parsing approach for LL(1) languages based on derivatives. We present an implementation of the approach as a parsing combinator framework, which supports static checks that the grammar is LL(1), and provides not only parsing and semantic actions, but also enumeration and pretty-printing functionality. Our implementation remains functional yet efficient, which allows us to obtain a proof that closely follows implementation.

Whereas parsing is a well understood problem, recent years have seen a renewed interest in approaches that handle not just language recognition but also syntax tree construction, and that are proven correct formally. Such parsing techniques can then be leveraged to more productively construct efficient front ends for verified compilers such as CompCert [36] and CakeML [30]. Safe and correct parsers are also crucial for building serialization and deserialization layers of communication infrastructure, which has been a major target of high-impact security exploits [6].

Parsing traditionally uses context-free grammars as the starting specification formalism and proceeds using table and stack-based algorithms. Popular techniques include LR parsing [13, 27, 32], LL parsing techniques [37, 51], recursive descent [10], Earley’s algorithm [14], and the Cocke-Younger-Kasami (CYK) algorithm [11, 26, 59]. Due to the significant gap between implementation and specification in such approaches, the resulting proofs are often based on validation as opposed to proofs for the general case [25].

In 1964, Brzozowski introduced the concept of derivatives of regular expressions [9]. This concept has proven successful in many formal proofs of parsing regular expressions and their generalisations [4, 45, 56, 57].

Derivatives of context-free expressions [35] generalize derivatives of regular expressions and have recently been used as an alternative principled approach to understanding context-free parsing [12, 40], avoiding explicit conversion into pushdown automata. Context-free expressions offer an algebraic view of context-free grammars. In addition to describing a language, context-free expressions also describe the value associated with each recognised input sequence, which makes integration into real-world parsers more natural. The concept of context-free expression derivatives was shown to naturally yield a parsing technique aptly named parsing with derivatives [40], which was later proved to have worst-case cubic complexity [1].

For integration into verifiable functional infrastructure, a particularly promising interface are parsing combinators [10, 15, 22, 23, 58]. Parsing combinator frameworks have been proposed for many functional programming languages such as Haskell [34] and Scala [19, 31]. Most implementations of parser combinators use recursive descent for parsing,
which has exponential worst-case complexity due to backtracking and can encounter stack overflows with deeply nested structures. Parsing expression grammars (PEGs) [17] are also popular in parsing combinators and have been formally verified [28]. While PEGs on the surface resemble context-free grammars, they behave very differently and can exhibit unpredictable behaviour [49]. In addition, merging lexical and syntactic analysis (as often done in the context of PEGs) is not helpful for performance in our experience.

In contrast, LL(1) parsing [37] is restricted to context-free grammars that can be non-ambiguously parsed given a single token of lookahead and runs in time linear in the input size. An appealing aspect of such grammars is that they can be algorithmically and efficiently analysed to prevent grammar design errors. In addition, they are known to provide good performance and error localisation [2]. Previous parsing combinator libraries for LL(1) languages either do not perform LL(1) checks [55] or impose restrictions on emptiness when parsing sequences [29], beyond those necessary for the definition of LL(1) languages.

By using the methodology of context-free expression derivatives, we can arrive at an efficient implementation of LL(1) parsing combinators, without introducing needless restrictions. We further show that, by embracing Huet’s zipper [21, 38] data structure, parsing with derivatives on LL(1) languages can be implemented with linear time complexity.

Our parsing approach sits at an interesting point in the design space of parsers. The approach offers a parser combinator interface, which is naturally embeddable in functional programming languages. The deeply embedded monoidal [39] nature of our combinators makes it possible to programmatically analyse parsing expressions and enables features such as enumeration of recognised sequences and pretty printing of values. Thanks to this representation, our approach also supports efficiently checking whether a description is LL(1), ensuring predictable linear-time parsing. Thanks to our use of derivatives for parsing, the parser state is explicitly encoded as an expression, not an automaton, which eases reasoning and implementation. Moreover, this representation of the state makes features such as parsing resumption, error reporting and completion straightforward to support.

Contributions

- We present an algorithm for parsing with derivatives using LL(1) syntaxes. Compared to traditional parsing, the algorithm works directly at the level of syntaxes, not on a derived push-down automaton. We show a technique based on Huet’s zipper [21] to make LL(1) parsing with derivatives efficient. We show that such zippy LL(1) parsing runs in time linear in the input.
- We present a Coq formalisation (https://github.com/epfl-lara/scallion-proofs) of syntaxes and prove the correctness of the zippy LL(1) parsing with derivatives algorithm and its auxiliary functions. For performing LL(1) checks, we formalise rule-based descriptions from which we can obtain both an inductive predicate and an equivalent propagator network.
- We present Scallion (https://github.com/epfl-lara/scallion), an implementation of syntaxes as a Scala parser combinators framework with a unique set of features, implementing LL(1) parsing using derivatives and the zipper data structure. The framework is efficient, provides error reporting, recovery, enumeration of accepted sequences, as well as pretty printing. Benchmarking our framework on a JSON syntax suggests performance comparable to existing Scala Parser Combinators [31], while avoiding stack overflows and providing more features.

2 Example

To give the flavour of our approach, Figure 1 presents a parser for JSON using Scallion, our parser combinators framework implemented in Scala. The sequencing combinator is denoted by infix \(~\), while disjunction is denoted by \(|\). The parser runs efficiently, even though it does not rely on code generation: with our simple hand-written lexer it takes about 40ms to parse 1MB of raw JSON data into a value of type `Value`, half of which is spent lexing. To provide a comparison point, an ANTLR-generated JSON parser [42–44] takes around 15ms per 1MB to produce a parse tree (using its own lexer).

As the Scallion framework is embedded in Scala, we can use the Scala REPL to query the parser. The following snippets show an example REPL interaction with the framework. We start by checking the LL(1) property for the top-level `JsonValue` syntax, and then list what kinds of tokens can start valid sequences using the method `first`.

```scala
scala> jsonValue.isLL1 // true
scala> jsonValue.first // Set(NullKind, SepKind('"'), ...)
```

When we feed a valid sequence of tokens to the syntax, we obtain, as expected, a JSON value.

```scala
scala> val tokens = JSONLexer("[1,2,3]")
scala> jsonValue(tokens) // Parsed(ArrayValue(...), ...)
```
We consider a single type \( T \). We can then query the residual syntax for valid ways to use the lower case letter \( v \) when we feed it an invalid sequence, the syntax duly re- turns a parse error, indicating the first unrecognised token.

We use \( ⟨⟩ \) for the empty sequence, \( x_1 \cdot x_2 \) for concatenation, and \( x : x_s \) for the prepending of \( x \) to \( x_s \).

3 Algebraic Framework for Parsing

In this section, we formalise the notion of syntaxes and describe their semantics as a relation between input token sequences and values.

We consider a set of values \( \mathcal{V} \) and a set of types \( \mathcal{T} \). For a value \( v \in \mathcal{V} \) and a type \( T \in \mathcal{T} \), we denote by \( v : T \) the fact that the value \( v \) has type \( T \). We denote by \( \{v_1, v_2\} \in \mathcal{V} \) the pair of the values \( v_1 \) and \( v_2 \) and by \( (T_1, T_2) \in \mathcal{T} \) the pair of types \( T_1 \) and \( T_2 \). We assume \( \{v_1, v_2\} : (T_1, T_2) \) if and only if \( v_1 : T_1 \) and \( v_2 : T_2 \). We denote by \( T_1 \rightarrow T_2 \) the set of total functions from values of type \( T_1 \) to values of type \( T_2 \).

We use \( \emptyset \) for the empty sequence, \( x_s \cdot x_s \) for concatenation, and \( x : x_s \) for the prepending of \( x \) to \( x_s \).

3.1 Tokens and Kinds

We consider a single type \( \text{Token} \) to be the type of tokens. The values \( v \in \mathcal{V} \) such that \( v : \text{Token} \) are called tokens. We will generally use the lower case letter \( t \) to denote such tokens. The task of parsing consists in turning a sequence of tokens into a value, or to fail when the sequence is invalid.

Each token \( t \) is assigned to a single kind \( \text{getKind}(t) \). Token kinds represent (potentially infinite) groups of tokens. We denote by \( K \) the set of all kinds, all assumed to be non-empty. There can be infinitely many different tokens, but only a finite, relatively small, number of kinds.

As an example, the strings "hello world", "foo" and "bar" could be considered tokens, and string would be their token kind. The numbers 3, 17, 42 could be considered tokens, while number would be their associated kind.

Token kinds are meant to abstract away details that are irrelevant for recognition. During parsing, the kinds alone are sufficient to decide whether or not a sequence of tokens is recognised. However, and importantly, the resulting value built by the parser might depend on the actual tokens.

3.2 Syntaxes

For every type \( T \in \mathcal{T} \), we define the set \( S_T \) of syntaxes that associates token sequences with values of type \( T \). Those sets are inductively defined by the rules in Figure 2.

```
object JSONParser extends Syntaxes[Token, Kind] {

  val boolValue: Syntax[Value] = accept(BoolKind) { case BoolToken(value) => BoolValue(value) }
  // Definition of other simple values in a similar fashion...

  implicit def separator(char: Char): Syntax[Token] = elem(SepKind(char))

  lazy val arrayValue: Syntax[Value] =
    ('[' ~ repsep(jsonValue, ',') ~ ']').map { case _ ~ values ~ _ => ArrayValue(values) }

  lazy val binding: Syntax[(StringValue, Value)] =
    stringValue ~ ':' ~ jsonValue).map { case key ~ _ ~ value => (key, value) }

  lazy val objectValue: Syntax[Value] =
    ('{' ~ repsep(binding, ',') ~ '}').map { case _ ~ bindings ~ _ => ObjectValue(bindings) }

  lazy val jsonValue: Syntax[Value] = recursive
    { arrayValue | objectValue | boolValue | numberValue | stringValue | nullValue }

  // Definition of other simple values in a similar fashion...
}
```

Figure 1. JSON Parser in Scala using Scallion, the parser combinator framework discussed in this paper.
The construct `elem_k`, `⊥`, and `ε_v` form the basic syntaxes. Intuitively, `elem_k` represents a single token of kind `k`, `⊥` represents failure, and `ε_v` represents the empty string. The value `v` tagged to `ε_v` represents the value associated to the empty string by the syntax. This value is important as syntaxes are meant not only to describe a language, but also the value associated with each recognised sequence of tokens.

The constructs `s_1 ⋁ s_2` and `s_1 ⋅ s_2` respectively represent disjunction and sequencing. The construct `f @ s` represents the application of the function `f` on values produced by `s`. Finally, the construct `var_x` represents a reference to a syntax defined in a global environment. The variables and the environment allow for mutually recursive syntaxes. We consider, for every type `T`, a finite set of identifiers `Σ_T`. The global environment is a finite mapping, that associates to each identifier `x` an environment `x ∈ Σ_T` a syntax `getDef(x) ∈ S_T`.

**Remark.** We use a global environment instead of the equivalent µ-combinator found in other approaches [29, 35] as it is closer to the representation we used in the implementation.

### 3.3 Semantics of Syntaxes

Syntaxs associate token sequences with values. The inductive predicate `s ⊢ ts ⊁ v` indicates that the syntax `s` associates the token sequence `ts` with the value `v`. The inductive predicate is defined by the rules in Figure 3.

**Figure 3.** Semantics of syntaxes.

**Theorem 3.1.** For any type `T ∈ T`, syntax `s ∈ S_T`, token sequence `ts` and value `v ∈ V`, if `s ⊢ ts ⊁ v` then `v : T`.

**Remark.** We do not present proofs of theorems in this paper and refer instead the reader to our proofs in Coq, discussed in Section 7. Given the order of theorems we present, most proofs follow relatively straightforwardly by induction, with main insight being the choice of induction variable and schema.

### 3.4 Example

As a simple example of our theoretical framework, we describe a syntax for the mapping `L = {a^n b^n \mapsto n \mid n ∈ N}`, which assigns to sequences of `n` `a`’s followed by `n` `b`’s the integer value `n`. The tokens we will consider are `a` and `b`, while their respective kinds are `A` and `B`. To describe a syntax for this language, we consider the singleton environment that maps the identifier `x` to (see Figure 4):

\[
\begin{align*}
&f @ ((\text{elem}_A \cdot \text{var}_x) \cdot \text{elem}_B) \lor \epsilon_0 \\
&\text{where } f(((t, n), t_z)) = n + 1
\end{align*}
\]

Intuitively, the syntax recognises sequences of the form:

- a token of kind `A`, followed another instance of the variable `x`, followed by a token of kind `B`, with `f` applied on the produced value, or
- the empty sequence of tokens, with value `0`.

In this environment, the mapping `L` is simply described by the syntax `var_x`. We can derive the following statements about the semantics of the syntax `var_x`:

\[
\begin{align*}
\text{var}_x &\vdash \langle \rangle \leadsto 0 \\
\text{var}_x &\vdash \langle a, a, b, b \rangle \leadsto 2
\end{align*}
\]

While the following statements are not derivable:

\[
\begin{align*}
\text{var}_x &\vdash \langle a, a, b, b \rangle \leadsto 17 \\
\text{var}_x &\vdash \langle a, a, b, b \rangle \leadsto 2 \\
\text{var}_x &\vdash \langle a, a, b, b \rangle \leadsto 2
\end{align*}
\]

The task of parsing is to find out, given a syntax `s` and a token sequence `ts`, if there exists a `v` such that `s ⊢ ts ⊁ v`, and if so, to return such a `v`. In the next section, we will introduce properties that characterise a class of syntaxes on which parsing is solvable in worst-case linear time complexity (LL(1) syntaxes). Afterwards, we will devise a parsing algorithm for such LL(1) syntaxes based on the concept of derivatives. Finally, we will present an optimisation based on the zipper data structure to obtain worst-case linear time complexity for LL(1) parsing with derivatives.

### 4 Properties of Syntaxes

This section defines several computable properties of syntaxes which we use for LL(1) checking and parsing.

#### 4.1 Productivity

A syntax is said to be productive (see Figure 5a) if it associates at least one sequence of tokens with a value.

**Theorem 4.1.** For any syntax `s`:

\[
\text{PRODUCTIVE}(s) \iff \exists ts, v. s \vdash ts \leadsto v
\]
Not all syntaxes are productive. For instance \( \perp, \perp \cdot \text{elem}_k \), \( \text{elem}_k \cdot \perp \) are all non-productive. Non-productive syntaxes can also occur due to non-well-founded recursion, as in the syntax \( \text{var}_x \) with an environment mapping \( x \) to \( \text{elem}_k \cdot \text{var}_x \).

### 4.2 Nullability

A syntax \( s \in S_T \) is said to be nullable with value \( v \) (see Figure 5b), if it associates to the empty sequence of tokens the value \( v \) of type \( T \). We will say \( s \) is nullable when we do not need to refer to the value that \( s \) is nullable with. We will use the function \( \text{nullable}(\_), \) to return an arbitrary nullable value from a syntax, if such a value exists, or none otherwise.

**Theorem 4.2.** For any syntax \( s \) and value \( v \):

\[
\text{NULLABLE}(s, v) \iff s \vdash () \leadsto v
\]

### 4.3 First Set

The first set of a syntax \( s \) (see Figure 5c) is the set containing the kinds of all tokens at the start of at least one sequence associated with some value by \( s \).

**Theorem 4.3.** The first set of a syntax \( s \) equals the set

\[
\{ k \mid \exists t, ts, v. \text{getKind}(t) = k \land s \vdash t :: t : \leadsto v \}
\]

### 4.4 Should-Not-Follow Set

The concept of a should-not-follow set is directly connected to the concept of LL(1) conflicts that we will later introduce. Intuitively, the should-not-follow set of a syntax is the set of kinds that would introduce an ambiguity if the first set of any syntax directly following that syntax was to contain that kind. The concept of should-not-follow set is used as an alternative to the concept of \text{FOLLOW} set generally used in the context of LL(1) parsing. While the \text{FOLLOW} set is a global property of a grammar, the should-not-follow set enjoys a local nature. Contrary to its \text{FOLLOW} set, the should-not-follow set of a syntax is a property of the syntax alone, not of its surrounding context. We define the should-not-follow set inductively in Figure 5d. Our definition differs from the one used by Krishnaswami and Yallop [29] and introduced in earlier works [8, 24]. While we introduce elements to the set in the case of disjunctions, they do so in the case of sequences. Our definition seems more appropriate: the previous work introduced additional restrictions on syntaxes, disallowing nullable expressions on left part of sequence, which is not needed in our approach (nor in conventional LL(1) definition for context-free grammars [3, Theorem 5.3, Page 343]).

**Theorem 4.4.** For any syntax \( s \) and kind \( k \), if \( k \) is part of the should-not-follow set of \( s \), then there exist a token \( t \) of kind \( k \) and (possibly empty) sequences of token \( ts_1 \) and \( ts_2 \) such that:

\[
s \vdash ts_1 \leadsto v_1 \land s \vdash ts_1 \oplus (t :: ts_2) \leadsto v_2
\]

### 4.5 LL(1) Conflicts

Finally, we introduce in Figure 5e the notion of LL(1) conflicts. When a syntax has LL(1) conflicts, a choice between two alternatives can arise during parsing which can not be resolved given a single token of lookahead. Informally, LL(1) conflicts arise in three cases: 1) Both branches of a disjunction are nullable, which means that two potentially different values are associated with the empty string by the disjunction. 2) Branches of a disjunction have non-disjoint first sets, so both branches can accept a sequence starting with the same token. Given a single token of lookahead, a parser thus cannot decide which branch to choose. 3) The should-not-follow set of the left-hand side of a sequence and the first set of the right-hand side of that sequence both contain the same token kind, \( k \). This means that there is a point in the left-hand side (after reading \( ts_1 \) from Theorem 4.4) where reading a token of kind \( k \) will make it impossible to decide whether we should stay in the left-hand side (and then read \( ts_2 \)), or start parsing in the right-hand side.

**Definition 4.5.** A syntax is LL(1) iff it has no LL(1) conflicts.

**Theorem 4.6.** [LL(1) syntaxes are non-ambiguous] For all LL(1) syntaxes \( s \), token sequences \( ts \) and values \( v_1 \) and \( v_2 \):

\[
s \vdash ts \leadsto v_1 \land s \vdash ts \leadsto v_2 \implies v_1 = v_2
\]

As a direct consequence of Theorem 4.6, we have that there can be at most a single nullable value associated to any LL(1) syntax, and thus that \( \text{nullable}(_) \) is completely deterministic for LL(1) syntaxes.

**Theorem 4.7.** Should-not-follow set of an LL(1) syntax \( s \) is

\[
\{ k \mid \exists t, ts_1, ts_2, v_1, v_2. \text{getKind}(t) = k \land \\
 s \vdash ts_1 \leadsto v_1 \land \\
 s \vdash ts_1 \oplus (t :: ts_2) \leadsto v_2 \}
\]

### 4.6 Left-Recursivity

We say an identifier \( x \) is left-recursive if \( \text{var}_x \) can be visited within \( \text{getDef}(x) \) without consuming input tokens. Formally, an identifier \( x \) is left-recursive if \( x \in \text{VISITABLE}(\text{getDef}(x)) \), where rules for \text{visibility} are given by Figure 6.

**Theorem 4.8.** For any left-recursive identifier \( x \), when \( \text{getDef}(x) \) is productive, the syntax \( \text{getDef}(x) \) is not LL(1).

### 4.7 Computing with Propagator Networks

The definitions we introduced in this section are based on inductive rules. Due to the cyclic nature of syntaxes arising from the variables and global environment, those definitions do not immediately give rise to recursive procedures. The usual solution to this problem is to iteratively apply rules until a fix-point is reached. We instead propose using propagator networks [47, 54] as a principled and efficient way to compute the properties defined in the present section. The idea is to build a network of cells, one for each node in the
Figure 5. Inductive definitions of properties on syntaxes.
syntax. For each identifier $x$, the $\text{var}_x$ nodes share the same cell. Each cell has a mutable state which holds information about the properties of the corresponding syntax node. We maintain a list of cells that need to be updated. Information is propagated through the network by updating the content of such cells according to the inductive rules presented in Figure 5. Using this approach, we found that properties can be propagated through the network by updating the content of such cells according to the inductive rules presented in Figure 5. Using this approach, we found that properties can be computed for a syntax and all its inner nodes in worst-case time linear in the size of the syntax, which is not direct from the conventional fix-point definitions. The constant number of kinds also factors in the cost of computations of first and should-not-follow sets. We have proven the correctness of the approach in Coq, as further discussed in Section 7.

5 Simple LL(1) Parsing with Derivatives

In this section, we introduce the concept of derivatives of LL(1) syntaxes and show how that concept leads to a simple parsing algorithm. Later, in Section 6, we discuss inefficiencies of that algorithm and propose a crucial optimisation which makes the algorithm run in linear time.

Remark. Theorems of this section are omitted from the Coq formalisation discussed in Section 7, as they are not relevant to the correctness of the parsing algorithm presented in Section 6.

5.1 Derivatives of LL(1) Syntaxes

The derivative of a syntax $s$ with respect to a token $t$ is a new syntax $\delta_t(s)$ which associates for every sequence $ts$ the value $v$ if and only if $s$ associates $t :: ts$ with $v$. The derivative of a syntax with respect to a token represents the state of the syntax after seeing the token $t$. Instead of defining the derivative for the general case, we only define it for LL(1) syntaxes $s$ and tokens $t$ such that $\text{getKind}(t) \in \text{FIRST}(s)$:

$$\delta_t(\text{elem}_k) := \varepsilon_t$$

$$\delta_t(s_1) := \begin{cases} \delta_t(s_1) & \text{if } \text{getKind}(t) \in \text{FIRST}(s_1) \\ \delta_t(s_2) & \text{otherwise} \end{cases}$$

$$\delta_t(s_1 \cdot s_2) := \delta_t(s_1) \cdot \delta_t(s_2)$$

$$\delta_t(f \circ s) := f \circ \delta_t(s)$$

$$\delta_t(\text{var}_x) := \delta_t(\text{getDef}(x))$$

The invariant that the token kind must be part of the first set reduces the number of cases to consider. In addition, the restriction to LL(1) syntaxes allows for drastic simplifications. Compared to the original definition of derivatives of context-free expressions by Might et al. [40], our definition only performs recursive calls on at most one child syntax. The choice of which child to recursively derive is informed by first sets. Thanks to Theorem 4.8, variables that are derived are not left-recursive, so the recursion is well-founded.

Theorem 5.1. The syntax $\delta_t(s)$ is well-defined for any LL(1) syntax $s$ and token $t$ of kind $k \in \text{FIRST}(s)$.

Theorem 5.2 (Progress). For any LL(1) syntax $s$, token $t$ of kind $k \in \text{FIRST}(s)$, token sequence $ts$ and value $v$ we have that $s$ associates the token sequence $t :: ts$ with the value $v$ iff $\delta_t(s)$ associates the token sequence $ts$ with the same value $v$:

$$\forall s, t. \neg \text{HAS-CONFLICT}(s) \land \text{getKind}(t) \in \text{FIRST}(s) \implies \forall ts, v. s \vdash t :: ts \leadsto v \iff \delta_t(s) \vdash ts \leadsto v$$

Theorem 5.3 (Preservation). For any LL(1) syntax $s$ and token $t$ of kind $k \in \text{FIRST}(s)$, the syntax $\delta_t(s)$ is LL(1).

$$\forall s, t. \neg \text{HAS-CONFLICT}(s) \land \text{getKind}(t) \in \text{FIRST}(s) \implies \neg \text{HAS-CONFLICT}(\delta_t(s))$$

5.2 Simple LL(1) Parsing with Derivatives

The derivation operation naturally leads to a parsing algorithm for LL(1) syntaxes:

$$s\text{Parse}(s, \langle \rangle) := \text{nullable}(s)$$

$$s\text{Parse}(s, t :: ts) := \text{if } \text{getKind}(t) \in \text{FIRST}(s)$$

$$\text{then } s\text{Parse}(\delta_t(s), ts) \text{ else } \text{none}$$

Theorem 5.4 (Correctness). For any LL(1) syntax $s$, token sequence $ts$ and value $v$:

$$s\text{Parse}(s, ts) = \text{some}(v) \iff s \vdash ts \leadsto v$$

6 Zippy LL(1) Parsing with Derivatives

In this section, we demonstrate that the simple parsing with derivatives for LL(1) syntaxes of Section 5.2 can have bad performance. To alleviate this problem, we introduce the
concept of focused syntaxes, which combine a syntax and a context. We show that, using such “zipper” data structure \[21\], LL(1) parsing with derivatives takes linear time.

### 6.1 Inefficiency of Simple Parsing with Derivatives

While correct, parsing with derivatives as shown in the previous section is inefficient in practice. As we will show, the derivative of a syntax can grow larger than the original syntax with respect to its nodes, in the spirit of zippers \[21\]. We define a focused syntax as a pair of a syntax \(s\) and a stack of layers \(c\). Given a focused syntax \((s, c)\), we call \(s\) the focal point and \(c\) the context. Layer of the context are of three different forms:

- apply\((f)\), which indicate that a function \(f\) is to be applied on the parsed value.
- prepend\((v)\), which indicate that a value \(v\) must be prepended to the parsed value.
- follow-by\((s)\), which indicate that the syntax \(s\) follows in sequence.

The first element of the sequence \(s\) is the original syntax \(var_s\), while subsequent elements are derivatives of the previous syntax with respect to \(a\). This sequence models the state of the parsing with derivatives algorithm after encountering longer and longer strings of \(a\)’s. Figure 7 shows the trees corresponding to \(s_1\), \(s_2\) and \(s_3\).

We immediately observe that \(s_i\) grows larger and larger as \(i\) grows. Each time a new \(a\) is encountered, additional nodes are added at the bottom of the previous syntax in place of the \(var_s\) sub-syntax. Indeed, the derivation process works its way down the syntax, unfolding variables as needed, until an \(elem_a\) node is found. At that point, the located \(elem_a\) node is replaced by an \(ε\) node. As we can observe, the \(elem_a\) node (hidden within \(var_s\)) is located deeper and deeper within each \(s_i\), making the derivation process longer and longer. In addition to that, as syntaxes are immutable structures, replacing the located \(elem_a\) node forces each node from the root to the leaf node \(elem_s\) to be copied. Therefore, we can easily observe that computing the derivative of \(s_i\) takes time linear in \(i\). What this means is that, in this particular case, the parsing algorithm that we have discussed in the previous section would require time quadratic in the input size, which is not desirable for LL(1) languages.

To tackle this phenomenon, we introduce focused syntaxes. The idea is very simple: Instead of having pointers always flowing away from the root down to the leaves, we will use a different structure in which pointers flow away from an inner syntax node called the focal point. To represent the syntax nodes on the path from the root of the tree to the focal point, we will introduce the concept of layers. Figure 8 shows the focused syntax corresponding to \(s_3\), with the bottommost \(ε\) node as the focal point. In the next section, we will formalise this concept and show how to adapt the LL(1) parsing with derivatizes algorithm to this zipper-inspired structure.

### 6.2 Focused Syntaxes

A focused syntax is simply a syntax with a focus on one of its nodes, in the spirit of zippers \[21\]. We define a focused syntax as a pair of a syntax \(s\) and a stack of layers \(c\). Given a focused syntax \((s, c)\), we call \(s\) the focal point and \(c\) the context. Layer of the context are of three different forms:

- apply\((f)\), which indicate that a function \(f\) is to be applied on the parsed value.
- prepend\((v)\), which indicate that a value \(v\) must be prepended to the parsed value.
- follow-by\((s)\), which indicate that the syntax \(s\) follows in sequence.

...the environment consists of a single entry:...
The first operation we define on focused syntaxes is \(
\text{plug}(v, c) := \text{match} \ c \text{ with } \\
| \langle \rangle \rightarrow (\epsilon_v, \langle \rangle) \\
| \text{apply}(f) :: c' \rightarrow \text{plug}(f(v), c') \\
| \text{prepend}(v') :: c' \rightarrow \text{plug}((v', v), c') \\
| \text{follow-by}(s') :: c' \rightarrow (s, \text{prepend}(v) :: c')
\)

**Theorem 6.2.** The focused syntax obtained by plugging a value \(v\) in a context \(c\) is equivalent to \((\epsilon_v, c)\). Formally: \(\forall t, s, w\),

\[
\text{unfocus}(\text{plug}(v, c)) \vdash ts \sim w \iff \text{unfocus}((\epsilon_v, c)) \vdash ts \sim w
\]

The next operation we define is \(\text{locate}\), which takes as input a token kind and a focused syntax, and returns an optional focused syntax. The goal of the function is to move the focus towards a syntax node that can start with a given token kind, skipping nullable prefixes as needed.

\[
\text{locate}(k, (s, c)) := \\
\text{if } k \in \text{first}(s) \text{ then } \text{some}((s, c)) \\
\text{else match nullable} \text{ with } \\
| \text{none} \rightarrow \text{none} \\
| \text{some}(v) \rightarrow \text{if } c = \langle \rangle \text{ then none} \\
\text{else locate}(k, \text{plug}(v, c))
\]

In case the current focal point starts with the desired kind the current focused syntax is simply returned. Otherwise, the focus is to be moved to a consecutive syntax found within the context, at which point the operation is recursively applied. Note that the operation does not always succeed, and so for two reasons. First, in order to be able to skip the currently focused node, that node must be nullable. Second, the context might be empty, and therefore no consecutive syntax exists.

**Theorem 6.3.** When the locate function returns nothing, the focused syntax can not possibly start with the desired kind.

\[
\text{locate}(k, (s, c)) = \text{none} \implies k \notin \text{first} \Rightarrow (s, c)
\]

**Theorem 6.4.** For any focused syntax \((s, c)\) and token kind \(k\), when locate successfully returns a new focused syntax, the new focal point starts with the given token kind \(k\).

\[
\text{locate}(k, (s, c)) = \text{some}((s', c')) \implies k \in \text{first}(s')
\]

**Theorem 6.5.** For any \(LL(1)\) focused syntax \((s, c)\), token \(t\) and associated kind \(k = \text{getKind}(t)\), when locate successfully returns a new focused syntax, then that focused syntax is equivalent for all sequences that start with the token \(t\).

\[
\text{locate}(k, (s, c)) = \text{some}((s', c')) \implies \forall t, s, v.
\]

\[
\text{unfocus}(s', c') \vdash t :: ts \sim v \iff \text{unfocus}(s, c) \vdash t :: ts \sim v
\]
Theorem 6.6. For any LL(1) focused syntax (s, c) and token t of kind k where k ∈ \text{\texttt{first}}(s), the following holds:

\[ \forall t, v. \, \text{unfocus}((\text{\texttt{elem}}_k, \text{pierce}(k, s, c))) \vdash t :: ts \Rightarrow v \iff \text{unfocus}(s, c) \vdash t :: ts \Rightarrow v \]

Theorem 6.7. Once focused on an \text{\texttt{elem}}_k node, derivation is trivial. For any context c and token t of kind k:

\[ \forall t, v. \, \text{unfocus}((\text{\texttt{elem}}_k, c)) \vdash t :: ts \Rightarrow v \iff \text{unfocus}((\epsilon_t, c)) \vdash t :: ts \Rightarrow v \]

Finally, the function \text{\texttt{derive}} brings the various operations we have seen so far together. The function takes as argument a token t and an LL(1) focused syntax (s, c). The function returns a new focused syntax (s′, c′) that corresponds to the derivative of (s, c) with respect to t, or none if the token is not accepted by the focused syntax.

\[ \text{\texttt{derive}}(t, (s, c)) \triangleq \text{\texttt{let}} k \triangleq \text{\texttt{getKind}}(t) \text{\texttt{in}} \]

\[ \text{\texttt{match}} \text{\texttt{locate}}(k, s, c) \text{\texttt{with}} \]

\[ | \text{\texttt{none}} \rightarrow \text{\texttt{pierce}}(k, s_1, \epsilon_c) \]

\[ | \text{\texttt{some}}(v) \rightarrow \]

\[ | k \in \text{\texttt{first}}(s_1) \rightarrow \text{\texttt{pierce}}(k, s_1, c) \]

\[ | \text{\texttt{false}} \rightarrow \text{\texttt{pierce}}(k, s_2, c) \]

\[ | f \circ s' \rightarrow \text{\texttt{pierce}}(k, s', \text{\texttt{apply}}(f)) \]

\[ | \text{\texttt{var}} \rightarrow \text{\texttt{pierce}}(k, \text{\texttt{getDef}}(x), c) \]

The next operation we consider is \text{\texttt{pierce}}. Given a LL(1) syntax s and a token kind k where k ∈ \text{\texttt{first}}(s), the function returns the context around the unique \text{\texttt{elem}}_k in a left-most position in s. An initial accumulator context is given to the function, and is only built upon by \text{\texttt{pierce}}.

\[ \text{\texttt{pierce}}(k, s, c) \triangleq \text{\texttt{match}} s \text{\texttt{with}} \]

\[ | \text{\texttt{elem}}_k \rightarrow c \]

\[ | s_1 \lor s_2 \rightarrow \]

\[ | \text{\texttt{if}} k \in \text{\texttt{first}}(s_1) \rightarrow \text{\texttt{pierce}}(k, s_1, c) \]

\[ | \text{\texttt{else}} \rightarrow \text{\texttt{pierce}}(k, s_2, c) \]

\[ | s_1 \cdot s_2 \rightarrow \]

\[ \text{\texttt{match}} \text{\texttt{nullable}}(s_1) \text{\texttt{with}} \]

\[ | \text{\texttt{none}} \rightarrow \text{\texttt{pierce}}(k, s_1, \text{\texttt{follow-by}}(s_2) :: c) \]

\[ | \text{\texttt{some}}(v) \rightarrow \]

\[ | k \in \text{\texttt{first}}(s_1) \rightarrow \text{\texttt{pierce}}(k, s_1, \text{\texttt{follow-by}}(s_2) :: c) \]

\[ | \text{\texttt{else}} \rightarrow \text{\texttt{pierce}}(k, s_2, \text{\texttt{prepend}}(v) :: c) \]

\[ | f \circ s' \rightarrow \text{\texttt{pierce}}(k, s', \text{\texttt{apply}}(f) :: c) \]

\[ | \text{\texttt{var}} \rightarrow \text{\texttt{pierce}}(k, \text{\texttt{getDef}}(x), c) \]

The operation first invokes \text{\texttt{locate}} to move the focus to a point which starts with the desired kind k, then, using \text{\texttt{pierce}}, moves the focus down to the left-most \text{\texttt{elem}}_k within that syntax. Once focused on that particular \text{\texttt{elem}}_k node, derivation is trivial, as it suffices to replace the focal point by an \epsilon_t node.

Theorem 6.8. The \text{\texttt{derive}} operation preserves the LL(1)-ness of the focused syntax. In other words, for any LL(1) focused syntax (s, c), if its derivation exists, then the resulting focused syntax is also LL(1).

Theorem 6.9. When the \text{\texttt{derive}} operation returns \text{\texttt{none}} for a token t (of kind k) and a focused syntax (s, c), then the corresponding unfocused syntax doesn’t start with k.

\[ \text{\texttt{derive}}(t, (s, c)) = \text{\texttt{none}} \implies k \notin \text{\texttt{first}}(\text{\texttt{unfocus}}((s, c))) \]

Theorem 6.10. For all LL(1) focused syntax (s, c) and token t, if the derivation returns a new focused syntax (s′, c′), then (s′, c′) is the derivative of (s, c) with respect to t.

\[ \text{\texttt{unfocus}}((s, c)) \vdash t :: ts \Rightarrow v \iff \text{\texttt{unfocus}}((s', c')) \vdash t :: ts \Rightarrow v \]

The final piece of the puzzle is the result operation, which returns the value associated with the empty string by the focused syntax.

\[ \text{\texttt{result}}((s, c)) \triangleq \text{\texttt{match}} \text{\texttt{nullable}}(s) \text{\texttt{with}} \]

\[ | \text{\texttt{none}} \rightarrow \text{\texttt{none}} \]

\[ | \text{\texttt{some}}(v) \rightarrow \]

\[ | c = () \rightarrow \text{\texttt{some}}(v) \]

\[ | \text{\texttt{else}} \rightarrow \text{\texttt{result}}(\text{\texttt{plug}}(v, c)) \]
Theorem 6.11. For all LL(1) focused syntax \((s, c)\):
\[
\text{result}((s, c)) = \text{nullable}(\text{unfocus}((s, c)))
\]

6.4 Zippy Parsing with Derivatives Algorithm
Using the previous definitions, we can finally present the zippy LL(1) parsing with derivatives algorithm. Given a focused syntax \((s, c)\) and a token sequence \(ts\), the algorithm returns the value associated with the token sequence, if any.
\[
\text{parse}((s, c), ts) := \text{match } ts \text{ with }
| \langle \rangle \rightarrow \text{result}((s, c))
| t :: ts' \rightarrow \text{match } \text{derive}(t, (s, c)) \text{ with }
| \text{none} \rightarrow \text{none}
| \text{some}((s', c')) \rightarrow \text{parse}((s', c'), ts')
\]

Theorem 6.12 (Correctness). For any LL(1) syntax \(s\), token sequence \(ts\) and value \(v\):
\[
\text{parse}(\text{focus}(s), ts) = \text{some}(v) \iff s \triangleright ts \sim v
\]

6.5 Example Execution
Figure 10 shows the execution of the algorithm on the syntax for the language \(L = \{a^n b^n \mid n \in \mathbb{N}\}\) introduced in Section 3.4, on the sequence of tokens \((a, a, b, b)\). The run consists in four derive calls (one per token) and an invocation of result. Each derive call is decomposed into a call to locate and a call to pierce. The focused syntax obtained after each such call is displayed.

6.6 Runtime Complexity of Parsers
In this section, we argue that the zippy LL(1) parsing with derivatives algorithm runs in time linear in the number of tokens (ignoring the cost of user-defined functions appearing in the syntax, which typically apply AST constructors).

A key observation towards this result is that the parsing algorithm never creates new syntax nodes apart from trivial \(\varepsilon_t\) nodes. This can be directly observed by inspecting the code of the parsing algorithm. This has major implications:

1. The computation of the properties of syntaxes by propagator networks can be performed once and for all before processing any input token. The properties of a syntax can then be directly stored in the syntax node. The process thus takes constant time with respect to number of input tokens.
2. Calls to pierce made by the parsing algorithm can only be made on a fixed number of preexisting syntaxes, whose sizes are therefore constant with respect to the input size. Thus each call runs in constant time with respect to the number of input tokens.
3. For the same reason, the number of layers added to the context by pierce is constant with respect to the input size.

We now argue that the complexity of the algorithm is proportional to the number of layer nodes traversed at parse time. Indeed, all operations performed by the parsing algorithm (calls to pierce, property checks and so on) only take constant time with respect to the number of input tokens and are performed at most once per traversed layer.

We now show that the number of layers traversed at parse time is linear in the input size. We observe that the plug procedure is the only procedure that directly traverses the stack of layers. Layer nodes are only ever visited once by plug in their entire lifetime: Indeed, once visited, layers are removed from the context and completely discarded. Therefore, the number of traversed layers is bounded by the number of layers created at parse time.

We now show that the number of layers created at parse time is linear in the input size. Only two procedures ever create layers: pierce and plug.

- As previously argued, pierce is only ever invoked on a finite number of preexisting syntaxes. For this reason the maximal number of layers nodes that are created by a single call to pierce is constant with respect to the input size. Since there is at most a single call to pierce per input token, the total number of layers created by pierce is linear in the input size.
- Additionally, the plug procedure itself can create layers. It however does so in a very limited way: plug only creates a prepend layer in case the layer it popped from the context was a follow-by layer. Since follow-by layers can only be created by pierce, the number of layers created by plug is bounded by the number of layers created by pierce, and thus is linear in the input size.

We therefore have that the total number of layers ever created at parse time is bounded linearly in the input size, which concludes the proof that the zippy LL(1) parsing with derivatives algorithm runs in time linear in the number of input tokens.

6.7 Connections to Traditional Parsing
The zippy LL(1) parsing with derivatives algorithm that we have presented in this section shares many features with the traditional LL(1) parsing algorithm. Immediately, we can observe that both algorithms maintain a stack of rules that are created by a single call to pierce and are performed at most once per traversed layer.

Instead of storing the transitions in a table, our transitions are performed at most once per traversed layer. Since follow-by layers can only be created by pierce, the number of layers created by plug is bounded by the number of layers created by pierce, and thus is linear in the input size.
Additionally, the zippy parsing with derivatives algorithm that we have presented in this section can be adapted to support general context-free expressions, and so by representing the context as a graph instead of a linear stack. The generalised version of the zippy parsing with derivatives algorithm is reminiscent of the GLL parsing algorithm \[51\].

7 Coq Proofs

We formalised the parsing with derivatives algorithm with zippy syntaxes in Coq (around 9000 lines). The Coq proofs are freely available at https://github.com/epfl-lara/scallion-proofs. We defined the recursive functions that require non-trivial measures using the Equations library \[52\]. There are two main parts in the formalism: one (around 7000 lines, including around 3000 lines about propagator networks constructions and proofs) to define the functions corresponding to the basic properties of syntaxes and many properties about them, and one (around 2000 lines) for the parsing algorithm based on zippy syntaxes (and its correctness).

In the first part, we defined for each function the inductive rules as described in Figure 5 and a corresponding propagator network that gives a way to compute the function. We defined a uniform way to specify these rules on syntaxes using the notion of a description. We then made a generic construction that takes a syntax and a description, and builds a propagator network that computes the function corresponding to the description on the syntax. This propagator network has one cell per node in the syntax, and each cell is updated using the inductive rules based on the cells corresponding to the children of the syntax. We proved soundness and completeness of this construction. Our Coq definitions of propagator networks (and their termination guarantees) are general and can be reused independently of this paper and independently of syntaxes.

In the second part, we defined zippy syntaxes, the functions plug, locate, pierce, derive and proved all the necessary properties to show the correctness of parsing as stated in Theorem 6.12. In particular, we proved that these functions terminate, that they do not introduce conflicts, and that they produce syntaxes that recognise the expected languages (Theorem 6.10).

8 Parsing and Printing Combinators

In this section, we discuss the implementation of syntaxes as a parsing and printing combinators framework in Scala. The framework is freely available under an open source license\(^1\). The Scala implementation closely follows the Coq formalism. For performance reasons, we did not mechanically extract an implementation from the formalisation.

---

\(^1\)The framework is available at https://github.com/epfl-lara/scallion

8.1 Syntax Definition

Syntaxes are defined as a generalised algebraic datatype named Syntax[A]. Each construct of the formalism straightforwardly corresponds to one constructor of the datatype. The environment syntaxes are directly stored in the instance of the corresponding variable syntaxes. To enable (mutually) recursive syntaxes, the syntax getDef(x) is stored in a lazy field of the instance of varx.

8.2 Computing Properties of Syntaxes

Properties of syntaxes (productivity, nullability, first sets etc.) are stored as public fields of Syntax instances. Fields are used by the LL(1) checking procedure and by the parsing algorithm. In addition, the first set of a syntax can be used to suggest fixes in case of parse errors. Propagator networks \[47, 54\] are used to initialise the fields ahead of parsing, as explained in Section 4.7.

The LL(1) property of syntaxes can be checked via a simple method call. In case a syntax is not LL(1), the list of conflicts can be obtained and their root causes identified. Coupled with the enumeration capabilities of the framework, users of the framework can easily get examples of token sequences which lead to conflicts.

8.3 Parsing

Parsing is performed via the apply method of Syntax[A]. The method takes as input an Iterator of tokens and returns a value of type ParseResult[A], which can be:

1. Parsed(value, descr), which indicates that the given value (of type A) was successfully parsed.
2. UnexpectedToken(token, descr), indicating that token was not expected. Values from the input iterator are not consumed beyond that token.
3. UnexpectedEnd(descr), which indicates that the end of input was not expected.

In each case, a residual focused syntax descr is also returned. This syntax represents the state at the end of parsing or at an error point. This syntax can be queried and used as any other syntax. In particular, it can be used for error reporting and recovery, and to resume parsing. Thanks to the derivatives-based algorithm, this syntax is available "for free".

The framework faithfully implements the zippy LL(1) parsing with derivatives presented in Section 6. The methods plug, locate and pierce are tail-recursive, which ensures that the call stack of underlying virtual machine does not overflow during parsing. The framework also implements memoisation of calls to pierce. The additional layers of context returned by pierce are stored in reverse order for fast concatenation.

8.4 Enumeration and Pretty Printing

Our framework also supports 1) enumeration of recognised sequences of token kinds and 2) pretty printing, that is, the
Table 1. Performance comparison between simple LL(1) parsing with derivatives (Simple), zippy LL(1) parsing with derivatives (Zippy), and Scala Parser Combinators (SPC) for parsing JSON. Entries marked with † encountered a stack overflow. Entries correspond to the mean of 36 measurements on a hot JVM.

<table>
<thead>
<tr>
<th>File size (KB)</th>
<th>Tokens</th>
<th>Parse time (ms)</th>
<th>Speed (token/ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Simple</td>
<td>Zippy</td>
</tr>
<tr>
<td>100</td>
<td>9649</td>
<td>99.9</td>
<td>2.8</td>
</tr>
<tr>
<td>1000</td>
<td>97821</td>
<td>7069.2</td>
<td>14.3</td>
</tr>
<tr>
<td>10000</td>
<td>971501</td>
<td>†</td>
<td>150.2</td>
</tr>
</tbody>
</table>

The zippy LL(1) parsing with derivatives outperforms the simple LL(1) parsing with derivatives algorithm by orders of magnitude. The speed of the zippy variant is comparable to the performance of the recursive descent algorithm implemented by the Scala Parser Combinators library. Worth noting, the simple parsing algorithm doesn’t suffer from stack overflows, which can occur with recursive descent when parsing deeply nested structures. Since parsers are often exposed to user inputs, an attacker could exploit this vulnerability in approaches based on recursive descent to cause crashes, and so with a relatively small input JSON file (as small as 2616 bytes in our tests), Our implementation also offers more comprehensive error reporting and recovery, in part thanks to encoding of parser states as explicit analysable expressions.

Table 2. Performance of the zippy parsing with derivatives algorithm combined with a handwritten lexer (Ours) compared to an ANTLR-generated lexer and parser (ANTLR).

<table>
<thead>
<tr>
<th>File size (KB)</th>
<th>Tokens</th>
<th>Lex &amp; parse time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Ours</td>
</tr>
<tr>
<td>100</td>
<td>9649</td>
<td>6.4</td>
</tr>
<tr>
<td>1000</td>
<td>97821</td>
<td>40.9</td>
</tr>
<tr>
<td>10000</td>
<td>971501</td>
<td>449.9</td>
</tr>
</tbody>
</table>

Finally, we benchmarked the performance of our approach compared to an ANTLR-generated JSON parser [42–44]. Results are presented in Table 2. Our approach, which doesn’t resort to code generation, is a constant factor (~3) slower than the ANTLR-generated solution. However, our approach offers a highly flexible and extensible parser combinators interface embedded in a rich programming language, while ANTLR grammars are described in a domain-specific language of limited expressivity. In addition, while our approach...
While these works operate at the level of context-free grammars, authors Lasser et al. [33] present a Coq-verified LL(1) parser generator. As a semester-long project, students used the presented framework to build a parser and pretty printer for a first-order logic formulas quasiquoter, a parser and pretty printer for lambda-calculus, a parser for an expression language with infix, prefix and postfix operators, as well as several other examples. We have also used our framework in EPFL CS-320, a third-year BSc compiler construction course with over 40 students. As a semester-long project, students build a compiler for a subset of Scala. Students successfully used the presented framework to build their parsers, appreciating the debugging capabilities offered by the framework.

10 Related Work

Ford [16] presents packrat parsing, a parsing technique for parsing expression grammars (PEGs). Packrat parsers are non-ambiguous and guaranteed to run in linear time through heavy use of memoisation but tend to be slower than many other linear-time parsing techniques [5, 18]. Whereas PEGs disallow ambiguities through biased choices, LL(1) approaches such as ours support detecting ambiguities before parsing starts. We believe that it is better to detect and report ambiguities rather than to hide them. Our combinators also enjoy more natural algebraic properties, with our disjunctions being commutative and associative, which is not the case in PEGs, making the composition of PEGs trickier.

Ramananandro et al. [48] demonstrate the importance of parsers in security and present combinators for building verified high-performance parsers for lower-level encodings of data formats. In contrast, we focus on parsing generalisations of context-free grammars. Formally verified parsers are of special interest to verified compilers such as CompCert [36] and CakeML [30]. Koprowski and Binsztok [28] present a formally verified Coq parser interpreter for PEGs. In recent work authors Lasser et al. [33] present a Coq-verified LL(1) parser generator. The generated parser uses the traditional table-based LL(1) algorithm, and relies on fix-point computations for properties such as nullability, first sets and others. While these works operate at the level of PEGs or context-free grammars, our work works on value-aware context-free expressions. As an alternative approach, Jourdan et al. [25] developed a validator (implemented and verified in Coq) for LR(1) parsers. Their approach works by verifying a posteriori that an automaton-based parser faithfully implements a context-free grammar, while we present a general correctness proof of a parser operating directly on context-free expressions. Swierstra and Duponcheel [55] propose parser combinators for LL(1) languages. Due to their approach based on a shallow embedding of combinators, they are unable to check for LL(1) conflicts a priori. The parsing procedure they use is based on lookup tables, as opposed to our parsing approach based on derivatives.

Our implementation supports mutually inverse parsing and pretty printing, which is also present in Rendel and Ostermann [50] based on syntactic descriptions and using recursive descent parsing (instead of using derivatives).

Krishnaswami and Yallop [29] propose a type-system for LL(1) context-free expressions. They use the usual conversion to push-down automata for parsing, and rely on code-generation for good performance. In their approach, the various properties of context-free expressions (nullability, first sets, etc.) are obtained via fixpoint computations, as opposed to our approach based on propagator networks. They use a weaker definition of should-not-follow set (which they call follow-last set, abbreviated as FLAST). Their type system is more restrictive than ours as it does not allow nullable expressions to appear on the left of sequences.

Might et al. [40] present a parsing algorithm for context-free expressions based on derivatives. Compared to our paper, their approach is not restricted to only LL(1) expressions, but is applicable to a wider family of context-free expressions. The worse-case complexity of their approach is cubic in general [1], and can be shown to be (at least) quadratic for LL(1) expressions by following an argument similar to Section 6.1. Our approach is limited to LL(1) languages but has guaranteed linear time complexity thanks to the use of a zipper-like data structure.

Brachthäuser et al. [7] showcase how derivatives can be used to augment the language of parser combinators and gain fine-grained control over the input stream. The feed and done combinators they introduce can be straightforwardly implemented as derived combinators in our setting. However, most of the examples and patterns demonstrating the power of their approach require a monadic flatMap combinator, which we do not support.

Henriksen et al. [20] show a parsing technique based on derivatives for context-free grammars. They show that their approach is equivalent to Earley’s algorithm [14] and argue that parsing with derivatives has deep connections with traditional parsing techniques. In this paper, we reinforce such connection, linking traditional LL(1) parsing to efficient parsing with derivatives.

Acknowledgments

This work is supported by the Swiss National Science Foundation Project number 200021_175676 as well as EPFL. We thank the anonymous PLDI’20 and PLDI’20 Artifact Evaluation reviewers for their thorough and insightful reviews. The authors would also like to thank Joachim Hugonot, Maxime Kjaer, Dragana Milovančević, Romain Ruetschi, Georg Schmid, and Natalia Štulova for their feedback and help on this work.


