Thèse n° 7357

Efficient Parsing with Derivatives and Zippers

Présentée le 9 juillet 2021

Faculté informatique et communications
Laboratoire d’analyse et de raisonnement automatisés
Programme doctoral en informatique et communications

pour l’obtention du grade de Docteur ès Sciences

par

Romain EDELMANN

École
polytechnique
fédérale
de Lausanne

2021

Correspondances

La Nature est un temple où de vivants piliers
Laissent parfois sortir de confuses paroles;
L'homme y passe à travers des forêts de symboles
Qui l'observent avec des regards familiers.

Comme de longs échos qui de loin se confondent
Dans une ténébreuse et profonde unité,
Vaste comme la nuit et comme la clarté,
Les parfums, les couleurs et les sons se répondent.

II est des parfums frais comme des chairs d'enfants,
Doux comme les hautbois, verts comme les prairies,
— Et d'autres, corrompus, riches et triomphants,

Ayant l'expansion des choses infinies,
Comme l'ambre, le musc, le benjoin et l'encens,
Qui chantent les transports de l'esprit et des sens.

Charles Baudelaire, Les Fleurs du mal

Correspondences

Nature is a temple in which living pillars
Sometimes give voice to confused words;
Man passes there through forests of symbols
Which look at him with understanding eyes.

Like prolonged echoes mingling in the distance
In a deep and tenebrous unity,
Vast as the dark of night and as the light of day,
Perfumes, sounds, and colors correspond.

There are perfumes as cool as the flesh of children,
Sweet as oboes, green as meadows
— And others are corrupt, and rich, triumphant,

With power to expand into infinity,
Like amber and incense, musk, benzoin,
That sing the ecstasy of the soul and senses.

Charles Baudelaire, Les Fleurs du mal
English translation by William Aggeler

Acknowledgements

First of all, I want to thank my advisor, Viktor Kunčak, for his ever-excellent advice, communicative energy, and most of all, for his trust in me and my research. This thesis would not have been possible if not for the unparalleled freedom and trust Viktor awarded me throughout these past years. I also want to thank the members of my thesis committee, Christoph Koch, Martin Odersky, François Pottier, and Matthew Might, for their time and input on the present thesis.

I thank my past and present fellow LARA researchers Jad Hamza, Nicolas Voirol, Georg Schmid, Dragana Milovancevic, Rodrigo Raya, Simon Guilloud, Romain Ruetschi, Manos Koukoutos, Régis Blanc, Ravichandhran Kandhadai Madhavan, Etienne Kneuss, Mikaël Mayer, Marco Antognini, Sarah Sallinger, Nataliia Stulova, and Andreas Pavlogiannis for the insightful discussions we had and the many laughs we shared. I am particularly grateful to Jad Hamza, who had the patience and trust to work with me on some the large formalisation efforts mentioned in this thesis. I also thank Jad for his insightful review of this thesis. I want to thank Sylvie Buchard and Fabien Salvi for the administrative, technical and otherwise general support they have offered me throughout the years. I also want to extend my thanks the people and friends in other labs I have had the chance to be in contact with as part of my PhD, most notably Joachim Hugonot, Ariane Staudenmann, Nicolas Stucki, Fengyun Liu, Olivier Blanvillain, Guillaume Martres, Natascha Fontana, Sébastien Doeraene, Darja Jovanovic and Julien Richard-Foy. I also thank Simon Bliudze and Joseph Sifakis for starting me on the path to research.

I would also like to thank my friends for their unfailing support during my time at EPFL. The many gaming sessions, philosophical discussions, and joyful events we shared over the years have allowed me to stay relatively sane!

I thank my parents, my sister, my sisters and brothers in law, and my parents in law, as well as all other members of my family, for their unconditional love and support throughout my years at EPFL. I consider myself lucky to have been given such a caring and nurturing environment. In the time that it took to complete my PhD, I have have had the great honour of becoming an uncle to Isaac, Séphora, Daphné, and Clément, and the even greater joy to become a father to Margot. Ever since she came into my life on 18th May 2020, Margot has been a source of immeasurable joy and pride.

Finally, I would like to thank my wife, Naomi, for all the sacrifices she made so that I could complete this thesis. Naomi provides the immovable foundation upon which I build my life. For that and countless other reasons, I thank you. I dedicate this thesis to you.

Abstract

Parsing is the process that enables a computer system to make sense of raw data. Parsing is common to almost all computer systems: It is involved every time sequential data is read and elaborated into structured data. The theory of parsing usually focuses on the binary recognition aspect of parsing and eschews this essential data-elaboration aspect. In this thesis, I present a declarative framework for value-aware parsing that explicitly integrates data elaboration.

Within the framework of the thesis, I present parsing algorithms that are based on the concept of Brzozowski's derivatives. Derivative-based parsing algorithms present several advantages: they are elegant, amenable to formal reasoning, and easy to implement. Unfortunately, the performance of these algorithms in practice is often not competitive with other approaches. In this thesis, I show a general technique inspired by Huet's *Zipper* to greatly enhance the performance of derivative-based algorithms, and I do so without compromising their elegance, amenability to formal reasoning, or ease of implementation.

First, I present a technique for building efficient tokenisers that is based on Brzozowski's derivatives and Huet's zipper and that does not require the usual burdensome explicit conversion to automata. I prove the technique is correct in Coq and present SILEX, a Scala lexing library based on the technique. I demonstrate that the approach is competitive with state-of-the-art solutions.

Then, I present a characterisation of LL(1) languages based on the concept of *should-not-follow* sets. I present an algorithm for parsing LL(1) languages with derivatives and zippers. I show a formal proof of the algorithm's correctness and prove its worst-case linear-time complexity. I show how the LL(1) *parsing with derivatives and zippers* algorithm corresponds to the traditional LL(1) parsing algorithm.

I then present SCALL1ON, a Scala parsing combinators library for LL(1) languages that incorporates the LL(1) *parsing with derivatives and zippers* algorithm. I present an expressive and familiar combinator-based interface for describing LL(1) languages. I present techniques that help precisely locate LL(1) conflicts in user code. I discuss several advantages of the *parsing with derivatives* approach within the context of a parsing library. I also present SCALL1ON's enumeration and pretty-printing features and discuss their implementation. Through a series of benchmarks, I demonstrate the good performance and practicality of the approach.

Finally, I present how to adapt the *LL(1) parsing with derivatives and zippers* algorithm to support arbitrary context-free languages. I show how the adapted algorithm corresponds to general parsing algorithms, such as Earley's parsing algorithm.

Keywords: Syntactic analysis, lexical analysis, parsing algorithms, formally verified parsing, context-free expressions, LL(1) expressions, Brzozowski's derivatives, Huet's zipper

Résumé

L'analyse syntaxique est le processus qui permet à un système informatique de donner du sens à des données brutes. L'analyse syntaxique est commune à presque tous les systèmes informatiques : elle intervient chaque fois que des données séquentielles sont lues et élaborées en données structurées. La théorie de l'analyse syntaxique se concentre généralement uniquement sur l'aspect de reconnaissance et néglige cet aspect essentiel d'élaboration des données. Dans cette thèse, je propose un cadre déclaratif pour l'analyse syntaxique qui intègre explicitement l'élaboration de données structurées.

Dans ce cadre, je présente des algorithmes d'analyse syntaxique qui sont basés sur le concept des dérivées de Brzozowski. Les algorithmes d'analyse syntactique basés sur les dérivées présentent plusieurs avantages : ils sont élégants, se prêtent au raisonnement formel et sont faciles à mettre en œuvre. Malheureusement, la performance en pratique n'est souvent pas compétitive avec d'autres approches. Dans cette thèse, je montre une technique générale inspirée du *zipper* de Huet pour améliorer considérablement les performances des algorithmes basés sur les dérivées, et ce sans compromettre leur élégance, leur affinité au raisonnement formel et leur facilité d'implémentation.

Tout d'abord, je présente une technique efficace basée sur les dérivées de Brzozowski et le *zipper* de Huet pour construire des analyseurs lexicaux. Cette technique ne nécessite pas l'habituelle et fastidieuse conversion explicite en automates. Je prouve que la technique est correcte dans Coq et je présente SILEX, une bibliothèque d'analyse lexicale en Scala basée sur cette technique. Je démontre que l'approche est compétitive par rapport aux solutions existantes.

Ensuite, je présente une caractérisation des langages LL(1) basée sur le concept des ensembles "*should-not-follow*". Je présente un algorithme d'analyse syntaxique des langages LL(1) basé sur les dérivées et les *zippers*. Je montre une preuve formelle de l'exactitude de l'algorithme et je prouve que sa complexité en temps est au pire linéaire. Je montre comment l'algorithme d'analyse syntaxique LL(1) avec dérivées et *zippers* correspond à l'algorithme d'analyse syntaxique LL(1) traditionnel.

Je présente ensuite SCALL1ON, une bibliothèque Scala d'analyse syntaxique pour les langages LL(1) à base de combinateurs. La bibliothèque incorpore l'algorithme d'analyse syntaxique LL(1) avec dérivées et *zippers* présenté dans cette thèse. Je présente une interface expressive et familière basée sur les combinateurs pour décrire les langages LL(1). Je présente des techniques qui aident à localiser précisément les conflits LL(1) dans le code utilisateur. Je discute de plusieurs avantages de l'approche d'analyse syntaxique avec dérivées dans le contexte d'une bibliothèque d'analyse syntaxique. Je présente également les fonctionnalités d'énumération et de sérialisation de SCALL1ON et je discute de leur mise en œuvre. À travers une série de benchmarks, je démontre les bonnes performances et l'aspect pratique de l'approche. Enfin, je présente comment adapter l'algorithme d'analyse syntaxique LL(1) avec dérivées et *zippers* pour supporter des langages non-contextuels arbitraires. Je montre comment l'algorithme adapté correspond aux algorithmes généraux d'analyse syntaxique, tels que l'algorithme d'analyse syntaxique de Earley.

Mots clés : Analyse syntaxique, analyse lexicale, algorithmes d'analyse syntaxique, analyse syntaxique vérifiée formellement, expressions non-contextuelles, expressions LL(1), dérivées de Brzozowski, *zipper* de Huet

Contents

1 Introduction

After more than half a century of theory and practice, parsing is still not a solved problem. Poorly behaved parsers can be security risks and cause performance bottlenecks, even in high-stakes projects. Recently, JSON parsing has been found to be the leading cause of the infamously long loading times of *Grand Theft Auto 5*'s online mode (t0st, 2021). At more than 140'000'000 copies sold worldwide, Grand Theft Auto 5 is the second best-selling video game of all times. Almost a decade after its initial release, its on-line mode still hosts around one hundred thousand daily users. This on-line mode has generated billions of dollars of revenue from micro-transactions over the years (Strickland, 2020). Due to a performance bug in the manually written parsing logic of the online mode's loading subsystem, parsing a 10MB JSON file, an operation that should generally take less than a second, takes several long minutes. Due to this bug, the online mode's loading time is reportedly more than doubled (t0st, 2021). Such long loading times are detrimental to the players' user experience and user engagement (Wiebe et al., 2014). We can only speculate about the financial impact this parser's performance bug has had over the years (Jin et al., 2017).

Parsers can also be security vulnerabilities. Recently, Barenghi et al. (2018) showed that widely used TLS libraries can accept syntactically invalid X.509 certificates due to a bug in their parsing logic. These authors show that this parsing bug is a serious attack channel and demonstrate how it can be exploited to perform impersonation attacks. In another recent parsing-related incident, a memory bug in an HTML parser used by the Cloudflare hosting company led to private data, including passwords, being leaked (Graham-Cumming, 2017).

Most programmers eventually face the problem of converting sequential data, be it a JSON file, an HTML webpage, an English sentence, or source code in a programming language, into a structured representation of the data that their program can operate on. Data represented in sequential form is often not fit for complex processing tasks, whereas a more structured representation of the data is often more appropriate. Programmers working on natural-language processing systems perform analyses much more efficiently on data represented as *parse trees* rather than on raw sequences of characters. Programmers working on formal-language tools, such as compilers, interpreters, and proof assistants, often spend considerable effort to convert their sequential input source code into *abstract syntax trees* before proceeding with their tool's later phases. Parsing is the task of converting sequential data into structured data, and the unit that performs this task is called a *parser*.

1.1 Writing Parsers

Programmers that seek to build a parser for their tool or system can adopt one of several competing approaches:

1. Programmers can use a parser generator, such as ANTLR (Parr, 2013), Yacc (Johnson et al., 1975), or Bison (Donnely and Stallman, 2015). Parser generators accept as input a description of the desired syntax, generally expressed in *grammar* form, and output executable code in some target language such as C or Java. Programmers that adopt this approach do not have to worry about implementing their parser: It suffices to *describe* the input language from which the parser generator automatically generates an executable parser. Additionally, the generated parsers can generally support error reporting features with little to no effort.

   However, this approach does not come without cost. Parser generators introduce a dependency on elaborate external tools, thus making the building process more complex and brittle. Grammars taken as input by parser generators are expressed in a domain-specific language that programmers working on the project must learn. Interfacing between the parser and the rest of the codebase can require substantial work, as the parse trees produced by the generated parsers might not match those used by the rest of the tool. Hence, a cumbersome conversion phase is often required. Keeping the parser and the rest of the codebase in sync as the two parts evolve could become challenging, made worse by the fact that the two are written in different languages.

   Finally, the domain-specific languages used by parser generators to describe syntaxes are often very limited. Such domain-specific languages lack the abstraction and code-reuse capabilities of general-purpose programming languages. Syntaxes expressed in these languages hence tend to be *low level* and verbose, thus making them harder to read and maintain than it is with syntaxes expressed using higher-level abstractions.

2. Alternatively, programmers can write their parser directly *by hand*, as they would for any other phase of their project; this alleviates the need to depend on an external tool. Often, programmers that adopt this technique write their parser as a collection of mutually recursive procedures that form a *recursive descent* parser (Burge, 1975). Although this gives programmers unparalleled freedom and flexibility, it also means that programmers are left alone to tackle the complexity of writing a parser. This complexity can lead to performance issues or downright design errors. And this can lead some designers to cripple their language to facilitate a parser's implementation. Certain syntactic constructs, such as infix binary operators with precedence levels, are notably complex

to handle, to the point that some ad hoc techniques, such as *Pratt parsing* (Pratt, 1973), have emerged to confront the complexity. Furthermore, adding orthogonal features, such as error reporting and recovery, is also tricky, as changes have generally to be made pervasively to support such features.

3. Finally, programmers can use a library to write their parser. Such libraries generally offer a limited set of simple primitive parsers and a collection of *parser combinators* (Frost and Launchbury, 1989; Hutton, 1992) to the programmers. Complex parsers are built by composing simpler parsers together, using the various combinators offered by the library. Due to the embedding in a general-purpose programming language, programmers are also free to build their own combinators and design abstractions specific to their language. Parser code expressed using parser combinators often looks like declarative grammars, which makes such code easier to read. The fact that combinators are often instances of algebraic structures, such as *applicative functors* (McBride and Paterson, 2008) and *monads* (Hutton and Meijer, 1996), also contributes to the ease of comprehension. The approach has been arguably well adopted, with many languages having access to well-maintained parsing combinators libraries (LAMP EPFL and Lightbend, Inc, 2019; Leijen and Meijer, 2001; Haoyi, 2021). Such parsing libraries generally implement parsing by recursive descent or by using Packrat parsing (Ford, 2004).

   This approach is, however, not without its shortcomings. Most parser combinators libraries do not support statically checking the code for ambiguities and design errors. To detect such issues, programmers have to test their parsers on various input strings, which is a time-consuming and incomplete process.

   Even though parser code written using parser combinators can superficially look like a declarative grammar, the parsing algorithm behaves quite differently from the programmers' expectations. For grammars, the order of alternatives is irrelevant. A grammar's alternative rules can be reordered without altering its meaning. For a recursive descent or Packrat parser, this is not the case. For such approaches, the order in which alternatives are processed often matters. In some instances, a partial match of one alternative can prevent another alternative from being tried.

   Although code written using parser combinators looks declarative on the surface, it is in fact imperative at its core. Most recursive descent libraries feature combinators that instruct the algorithm if and when to backtrack. This has no equivalent in the declarative setting of context-free grammars. In practice, controlling when and how to backtrack is essential for performance but is, unfortunately, error prone.

   It is worth noting that some parsing approaches offer a parser combinator interface that truly and closely corresponds to context-free grammars. One such approach is *Parsing with Derivatives* (Might et al., 2011; Adams et al., 2016), which relies on parser *derivation* (Brzozowski, 1964) for the basic operation behind parsing. This elegant and simple approach is the basis of the various techniques presented in this thesis. Despite its elegance, this approach has not yet seen frequent adoption. One of the possible reasons for this is that its performance is generally below that of more established approaches. In this thesis, I will show techniques for improving the performance of such approaches.

In this thesis, I will describe lexing and parsing approaches that can be classified in the third category, that is of parser combinators libraries. I will present several distinct approaches: one for regular languages, one tailored for LL(1) languages, and one for general context-free languages. Instead of relying on a *shallow embedding* (Gibbons and Wu, 2014) of parsers as functions, as is the case for most parser combinator libraries, the approaches I present are based on a *deep embedding* of parsers as directed graphs. I will rely on Brzozowski's derivatives (Brzozowski, 1964) to evolve this graph as input tokens are processed. I will show how adopting a data structure that is based on Huet's *zipper* (Huet, 1997) to represent parser states makes the approaches efficient in practice.

As I will demonstrate in this thesis, this deep embedding of parsers also facilitates the implementation of many features such as static checks of conflicts, precise parsing, error reporting and recovery, enumeration and code completion, as well as correct-by-construction serialisation.

1.2 Writing Pretty Printers

Some tools not only have to accept strings as input, but also to respond by showing strings back to the user; they generally must do this using the *same language*. For example, consider an interactive theorem-prover system. Such systems keep track of the current assumptions and current goals in their state and communicate them to the user. To facilitate the interaction between the user and the system, the assumptions and goals in the state are displayed using the same language as the users use to input their lemmas and theorems. Tools such as code synthesisers and code repair tools are another example of such systems. In such tools, not only must code be parsed, but it must also be produced.

Whereas parsing is the task of converting sequences into structured data, the task of *pretty printing* is the inverse: displaying structured data as sequences. Although the two tasks are generally closely related, they are (more often than not) defined in separate functional units, even though they often share the same structure. This separation leads to duplicated code and extra development and maintenance work. Furthermore, having two separate units increases the risk of inconsistencies significantly. In the context of proof assistants, one such type of inconsistency is the so-called Pollack's inconsistency (Wiedijk, 2012), in which a user can be led to believe a false statement due to the parser assigning a different meaning to a pretty printed theorem. Therefore, it is crucial in such tools that parsing a pretty printed value yields back the same value. This property is crucial for the confidence in the tool: without it, users could mistake theorems that they have shown for other propositions. However, in practice, this crucial property is often violated (Wiedijk, 2012).

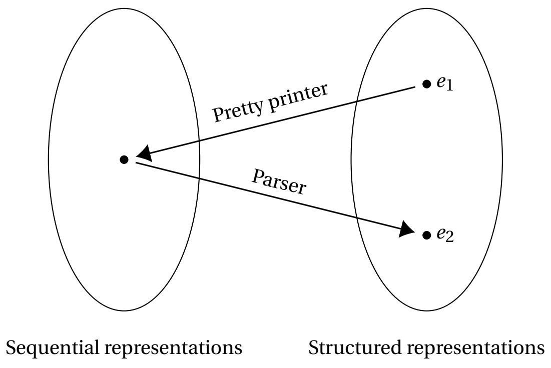


Figure 1.1 – Example inconsistency. Whereas the pretty printer has been given e_1 as input, the user is shown a representation that the parser would convert to e_2. When e_1 and e_2 are semantically different, an inconsistency arises. In the context of a theorem prover, this means that a user could be led to erroneously believe that e_2 is a true and proven proposition, when, in fact, the semantically different proposition e_1 has been shown.

In this thesis, I will show that the parsing combinators library presented here can ease the writing of a corresponding consistent pretty printer. By merely providing partial inverses for user-defined functions locally at the places where they are applied, users of the libraries get a pretty printer for free, in addition to their parser. Given sound local inverses, the pretty printer is guaranteed to produce sequences that can be, with equivalent meaning, parsed back by the parser. As the parser and the pretty-printer share their definition, keeping them in sync as the language evolves is a much easier task.

1.3 Teaching about Parsing

Most *compiler construction* courses in schools and universities around the world introduce and use regular expressions as the de facto formalism for lexical analysis. Such courses then switch to context-free grammars for syntactic analysis. The techniques and algorithms that are taught, such as conversion to automata in the case of regular expressions, and LL, LR, or Earley in the case of grammars, seem utterly unrelated to each other. This apparent gap between the various techniques hinders their understanding. Moreover, the generation of parse trees or parse forests is an afterthought for many traditional parsing algorithms. Consequently, lecturers and researchers often brush over this aspect in their lectures and research papers.

In this thesis, I demonstrate that expression-based formalisms can form a cohesive approach to teaching both lexical and syntactic analysis. I will show, in Chapter 2 of this thesis, that efficient lexical analysis can be performed directly on regular expressions, without explicit conversion to automata. I also argue that, in educational and academic settings, context-free expressions are a suitable alternative to context-free grammars. Context-free expressions are easy and natural to transition into from regular expressions, as they are a simple extension of these kinds of expressions. The progression from regular expressions to context-free expressions follows the order in which the lexical analysis and the syntactic analysis phases occur in a typical compiler. As the order in which subjects are covered in compiler construction courses tends to mimic the order of phases in compilers, this transition from regular to context-free expressions is natural.

Even more importantly, the same simple technique, based on Brzozowski's derivatives (Brzozowski, 1964) and Huet's zipper (Huet, 1997), can be used to explain and understand the various algorithms used in the context of both lexical and syntactical analysis. As I will show in Chapter 2, in the context of regular expressions, combining derivatives with zippers yields an algorithm that corresponds to running a finite-state automaton created on the fly. Additionally, as I will show in Chapter 5, combining derivatives and zippers in the context of LL(1) expressions yields an algorithm that closely corresponds to the traditional table-based LL(1) parsing algorithm. In Chapter 7, I will show that applying the same techniques on general context-free expressions instead yields an algorithm reminiscent of general context-free parsing algorithms such as Earley's algorithm. Furthermore, due to the value awareness of context-free expressions that are used to represent the parser state in such *parsing with derivatives and zippers* algorithms, parse-value generation is no longer an afterthought and is deeply ingrained in the algorithms.

I will argue that most parsing algorithms can be viewed and understood under the unifying lens of derivatives and zippers. This common framework shows that the dichotomy between bottom-up and top-down parsing is not as relevant and fundamental as might be initially thought. This common ground abstraction could enable the transfer of techniques designed for one algorithm to other parsing algorithms. It could also be an excellent guiding principle towards supporting richer types of expressions, such as expressions with monadic sequencing combinator as commonly found in parser combinator libraries, in traditional parsing algorithms.

1.4 Formally Reasoning about Parsing

Expression-based formalisms such as regular expressions and context-free expressions, along with derivatives-based algorithms, also seem particularly suited to formal proofs in proof assistants such as Coq. The fact that the correctness of parsing with derivatives on regular expressions is offered as an exercise in an introductory book on Coq (Pierce et al., 2018, Chapter on Inductively Defined Propositions) testifies to this. To further demonstrate this, the correctness of lexical analysis technique presented in Chapter 2 is supported by a formalisation in Coq. Moreover, the framework and parsing algorithms presented in Chapters 3 to 5 have also been implemented in Coq by Jad Hamza and myself (Hamza and Edelmann, 2019; Edelmann et al., 2020), thus resulting in a mechanised proof of correctness of LL(1) parsing with derivatives and zippers.

1.5 Thesis

In this thesis, I will demonstrate that combining derivatives and zippers leads to lexical and syntactic analysis algorithms that are amenable to formal reasoning, are easy to integrate, support expressive interfaces, and display excellent performance in practice. *Parsing with derivatives and zippers* approaches are the best choice for building fast and flexible parsers that can be trusted.

1.6 Contributions

The main contributions of this thesis are the following:

- A novel technique for building lexical analysis libraries in functional programming languages based on the concept of derivatives (Brzozowski, 1964) and zippers (Huet, 1997). The zipper-based representation provides a way to effectively classify expressions and their derivatives into a finite number of equivalence classes, in a manner that is reminiscent of the *partial derivatives* of Antimirov (1996). I show that, by employing memoisation in combination with the technique, a lazily built deterministic finite-state automata is obtained, without going through the burdensome traditional explicit conversion to automata. The approach is supported by a formalisation in Coq.
- A theoretical framework for value-aware context-free expressions, in which semantics and properties of context-free expressions are easily expressed using inductive predicates. I demonstrate how such properties can be efficiently computed using *propagation networks* (Radul, 2009).
- A novel characterisation of the LL(1) class for context-free expressions based on the concept of *should-not-follow* sets. *Should-not-follow* sets are an alternative to the traditional `FOLLOW` sets that have a more compositional nature. I arrive at this novel characterisation of the LL(1) class through the lens of Brzozowski's derivation (Brzozowski, 1964; Might et al., 2011).
- A novel and efficient parsing algorithm for LL(1) context-free expressions, called *zippy LL(1) parsing with derivatives*. The algorithm, which is based on Brzozowski's derivatives and Huet's zipper (Huet, 1997), is purely functional and incorporates value elaboration aspects.
- A proof of correctness of the zippy LL(1) parsing with derivatives. A formal and detailed proof is presented in this thesis. A mechanised version of the proof in Coq, written by Jad Hamza and myself (Hamza and Edelmann, 2019; Edelmann et al., 2020), is also available.
- A Scala implementation of an LL(1) parsing and pretty-printing combinators library called SCALL1ON (Edelmann, 2019). The library features an efficient implementation of

the zippy LL(1) *parsing with derivatives* algorithm.

- A novel parsing algorithm for general context-free expressions based on Brzozowski's derivatives and Huet's zippers. The algorithm is a generalisation of the zippy LL(1) parsing with derivatives.

Taken together, these contributions show that *parsing with derivatives and zippers* is a compelling alternative to more traditional parsing techniques. First of all, the technique is practical. The approach is easy to embed in functional programming languages and integrates well into projects that rely on parsing. With the support of combinator-based interfaces, the technique is highly expressive. The performance of the approach is competitive with state-of-the-art parsing techniques. Second, the approach is trustworthy. As demonstrated in this thesis, *parsing with derivatives and zippers* techniques are amenable to formal proofs. Additionally, *parsing with derivatives and zippers* is relatively easy to understand. Third, the approach is widely applicable. The same zipper-based optimisation can be applied with great effects on regular, LL(1), and general context-free expressions, thus leading to a unified presentation of parsing.

1.7 Overview

The remainder of this thesis is structured as follows:

1. In Chapter 2, I give an introduction to the well-known concept of regular expressions and of Brzozowski's derivatives (Brzozowski, 1964). I show that using Huet's zipper (Huet, 1997) to represent regular expressions and their derivatives provides a way to efficiently classify expressions into a finite number of equivalence classes. I show that, by introducing memoisation, we obtain a way to lazily build deterministic finite-state automata on the fly, as characters of input are processed. I show an implementation of a lexical analysis library in Scala based on the approach. The implementation is very concise, and its performance in practice is good. The correctness of the approach is backed by a formalisation in Coq. For a further example of the amenability of expression-based formalisms to formal proofs, I show a proof of the pumping lemma for regular languages operating directly on regular expressions.

2. In Chapter 3, I present a formalisation of value-aware context-free expressions and model their semantics as an inductive predicate. I present binary relations to semantically compare context-free expressions. Such relations will prove useful to talk about the correctness of transformations applied to expressions. I then introduce several properties of context-free expressions, such as nullability and first sets, and I show their relation to the semantics of expressions. I discuss how to efficiently compute such properties by using a technique based on *propagation networks* (Radul, 2009). I conclude

this chapter by presenting how the original *parsing with derivatives* algorithm (Might et al., 2011) can be expressed in the framework introduced.

3. In Chapter 4, I revisit the notion of LL(1) languages and apply it to context-free expressions. I introduce the notion of *should-not-follow* sets, a notion dual to the `FOLLOW` sets typically found in the literature on LL(1) grammars. I argue that the notion of *should-not-follow* sets presents several advantages over competing notions. I then present a simplified version of the *parsing with derivatives* algorithm tailored to LL(1) expressions. A complete proof of the correctness of the algorithm is presented. Through an example, I show that the parsing algorithm can unfortunately exhibit a quadratic running time, compared to the guaranteed linear running time of the traditional LL(1) parsing algorithm. I identify the cause of the issue and show that it also arises in the context of the original *parsing with derivatives* algorithm by Might et al. (2011).

4. In Chapter 5, I show that introducing a zipper turns the LL(1) parsing with derivatives algorithm presented in Chapter 4 into an efficient linear-time parsing algorithm. A complete proof of correctness of the algorithm, as well as a careful complexity analysis, is presented. Caching techniques are discussed to further accelerate the algorithm in practice. I conclude that chapter by showing that the LL(1) *parsing with derivatives and zippers* algorithm closely corresponds to the traditional table-based LL(1) parsing algorithm, thus shedding new light on this well-established algorithm.

5. In Chapter 6, I present SCALL1ON, a Scala parsing combinators library for LL(1) languages based on the theory developed in earlier chapters.

   I present techniques to enumerate recognised sequences of tokens from context-free expressions. I show that, combined with derivatives-based algorithms, this enables features such as code completion and error recovery. Finally, by presenting a technique that produces a shortest representation of a value from a context-free expression, I show that context-free expressions are also suitable tools for describing pretty printers.

6. In Chapter 7, I present a implementation of a *parsing with derivatives and zippers* algorithm generalised to arbitrary context-free expressions. I also make connections to well-known parsing algorithms.

7. In Chapter 8, I give references to works related to the topics covered in this thesis.

8. Chapter 9 concludes the thesis. Several pointers for future work are provided.

2 Regular Expressions

In this chapter, I present an introduction to regular expressions (Sipser, 2012). Not only are regular expressions an exciting topic on their own, but they also play a significant role in latter parts of the thesis. Indeed, the expressions at the foundation of the parsing techniques presented in this thesis, which are called *context-free expressions*, are *systems* of these regular expressions. The primary operation at the root of the various parsing techniques that I present throughout this thesis are an adaptation of the *derivation* operation on regular expressions proposed by Brzozowski (1964) and later transposed to context-free expressions by Might et al. (2011).

Brzozowski's derivatives offer an elegant way to build deterministic finite-state automata from regular expressions. States of the automaton correspond to regular expressions, whereas states' transitions are obtained from a *derivation* operation. Although extremely simple on the surface, to ensure the finiteness of states, the technique requires grouping expressions into a finite set of equivalence classes. Building equivalence classes is unfortunately non-trivial (Owens et al., 2009).

In this chapter, I also introduce the concept of *zippers* proposed by Huet (1997). I show that zippers can be used to represent regular expressions and their derivatives. This zipper-based representation of regular expressions will prove advantageous because, as I will demonstrate, only a finite number of different zippers can be encountered through successive derivations. Hence, structural equality of zippers can be used as a way to efficiently classify regular expressions into a finite number of equivalence classes. I show a recognition procedure that, based on Brzozowski's derivatives and Huet's zipper, operates over a lazily built deterministic finite-state automaton. In this setting, the automaton can be seen as a mere memoisation optimisation. The technique has been implemented as part of the SILEX [1] Scala lexing library that is used in the *Computer Language Processing* class at EPFL.

I conclude this chapter by showing a proof of the regular pumping lemma on regular expressions. Typical proofs of the lemma operate over automata and invoke the pigeon-hole

[1] https://github.com/epfl-lara/silex

principle. Whereas the proof shown in this chapter operates by straightforward structural induction over regular expressions. This elegant proof is a testimony to the value of regular expressions as a formalism.

2.1 Definition

Regular expressions are an expression-based formalism for describing sets of sequences of characters taken from some alphabet A. The set of sequences of characters described by an expression is called its *language*. I will denote by $L(e)$ the language of the regular expression e. The various constructs of regular expressions are presented in Listing 19.

Empty expression	$\bot$
Empty word	ε
Single character	c
Sequencing	$e_1 \cdot e_2$
Disjunction	$e_1 \vee e_2$
Repetition	e^*

Figure 2.1 – Regular expressions.

Three constructs describe basic regular expressions. The expression $\bot$ represents the empty language $\emptyset$. The expression ε represents the singleton language containing only the empty sequence $\langle\rangle$. Finally, for every character c in the given alphabet A, the expression c represent the singleton language containing the singleton sequence $\langle c\rangle$:

$$\begin{aligned} L(\bot) &:= \emptyset \\ L(\varepsilon) &:= \{\, \langle\rangle \,\} \\ L(c) &:= \{\, \langle c\rangle \,\} \end{aligned}$$

Additionally, three combinator constructs allow the composition of small regular expressions into larger expressions. The sequencing combinator, denoted by $e_1 \cdot e_2$, represents the sequential composition of the two expressions e_1 and e_2. The language of $e_1 \cdot e_2$ is the product concatenation of the languages of e_1 and e_2, that is:

$$L(e_1 \cdot e_2) := \{\, cs_1 \mathbin{++} cs_2 \mid cs_1 \in L(e_1) \wedge cs_2 \in L(e_2) \,\}$$

The second combinator is disjunction. The language of the disjunction of two expressions e_1 and e_2 is the union of the languages of e_1 and e_2:

$$L(e_1 \vee e_2) := L(e_1) \cup L(e_2)$$

Finally, the last regular expression combinator is the repetition combinator, also called *Kleene* *star.* The expression e^* represents an arbitrary number of repetitions of the underlying expression e. Its language is defined as the infinite union:

$$\begin{aligned} L(e^*) &:= \bigcup_{i \in \mathcal{N}} U_e^i \\ U_e^0 &:= \{ \langle\rangle \} \\ U_e^{n+1} &:= \{ cs_1 ++ cs_2 \mid cs_1 \in L(e) \wedge cs_2 \in U_e^n \} \end{aligned}$$

The semantics of regular expressions can be expressed elegantly using an *inductive predicate* instead of set-based operations. The inductive predicate $e \vdash cs$ indicates that the expression e describes the sequence cs:

$$cs \in L(e) \iff e \vdash cs$$

The inductive rules that define the predicate $e \vdash cs$ are presented in Figure 2.2.

$$\frac{}{c \vdash \langle c \rangle} \qquad \frac{}{\varepsilon \vdash \langle\rangle} \qquad \frac{e_1 \vdash cs_1 \quad e_2 \vdash cs_2}{e_1 \cdot e_2 \vdash cs_1 ++ cs_2}$$

$$\frac{e_1 \vdash cs}{e_1 \vee e_2 \vdash cs} \qquad \frac{e_2 \vdash cs}{e_1 \vee e_2 \vdash cs} \qquad \frac{}{e^* \vdash \langle\rangle} \qquad \frac{e \vdash cs_1 \quad e^* \vdash cs_2}{e^* \vdash cs_1 ++ cs_2}$$

Figure 2.2 – Semantics of regular expressions.

Inductive predicates are an excellent tool for formal reasoning. Such predicates can be deconstructed to obtain their possible premises. Furthermore, proofs can be performed by induction over such predicates. In this thesis, I will generally make use of such inductive definitions to define predicates and various properties for this reason.

Note that such inductive properties do not immediately give rise to recursive definitions of computable functions. In the above example, the $e \vdash cs$ inductive predicate is merely a specification of the semantics of regular expressions: The definition can not directly be used as a definition for a recursive function. Indeed, there might be multiple valid ways to derive a proposition $e \vdash cs$, and knowing which to apply given only e and cs is non-trivial. For this reason, procedure to check membership in the language of a regular expression often look very different from the inductive definition.

2.2 Tree Representation

Regular expressions can very naturally expressed as an algebraic datatype in functional programming languages. Listing 1 presents a definition of regular expressions as such a datatype in Scala.

```
sealed trait RegExpr
case object Epsilon extends RegExpr
case object Failure extends RegExpr
case class Character(char: Char) extends RegExpr
case class Disjunction(left: RegExpr, right: RegExpr) extends RegExpr
case class Sequence(left: RegExpr, right: RegExpr) extends RegExpr
case class Repetition(inner: RegExpr) extends RegExpr
```

Listing 1 – Definition of the `RegExpr` algebraic datatype in Scala.

This datatype exhibits a tree-like structure, where the top-level combinator acts as the root of the tree, and basic regular expressions such as `Epsilon`, `Failure`, and `Character`, appear at the leaves. In this setting, all references flow from the top-level combinator down to the basic expressions. Combinators have a reference to their children expressions, but expressions have no way of referencing back to their parent (or parents). This representation of regular expressions has the advantage that procedures operating over regular expressions can be straightforwardly defined as recursive functions.

This canonical representation is however not the only way one can encode regular expressions. In the rest of this thesis, I will explore different kinds of representations of regular and context-free expressions (Chapter 3) inspired by Huet's *Zipper* (Huet, 1997), that will prove more appropriate in the context of derivatives-based approaches (Brzozowski, 1964; Might et al., 2011). I first demonstrate the approach in Section 2.6.

2.3 Conversion to Automata

It appears that checking whether a given sequence of characters cs is part of the language of a regular expression e is a task that can not be efficiently performed by a simple recursive function over the structure of regular expressions. The culprit is that some expressions have multiple ways of producing a sequence cs, many of which can not be immediately dismissed. For instance, it is not immediately clear which side of a disjunction produced the sequence cs. For sequencing and repetition, the situation is even worse: it is not immediately clear how to split the sequence of characters cs to proceed with the recursive calls.

The usual solution is to build an *automaton* out of the regular expression e as a pre-processing step. One way to build such an automaton is to first convert the regular expression into a *non-deterministic finite-state automaton* (NFA), which can later be converted to a *deterministic finite-state automaton* (DFA) through a process called determinisation. Once in automaton form, checking whether a given sequence cs is accepted is trivial as it suffices to follow transitions as described by the automaton.

NFA Definition

Given an alphabet A, non-deterministic finite state automata are composed of:

- A set of *states* S,
- A starting state $s_0 \in S$,
- A set of finite states $F \subseteq S$,
- A set of *transitions* Δ of the form $s_i \overset{c}{\mapsto} s_j$ for some character $c \in A$ and states s_i and $s_j \in S$. As the automaton is *non-deterministic*, in any given state s_i there might be any number of states s_j with an existing transition $s_i \overset{c}{\mapsto} s_j \in \Delta$ for the same character c.

  Additionally, the set delta may contain transitions $s_i \mapsto s_j$ that are not labeled by any character c. Such transitions are commonly known as ε-transitions.

NFA Construction Code

In Listing 2, I present an elegant way to convert regular expressions to non-deterministic finite-state automata in Scala. The code to convert the non-deterministic automaton into an efficient deterministic automaton is not shown. The code is expressed in a *continuation-passing* style: when visiting an expression node, a continuation state is given to the visitor function. This continuation-passing style allows for the definition to be rather concise and elegant.

The Cost of Conversion

Having to convert regular expressions to automata to execute them is not without cost. I argue that one of the main drawbacks is pedagogical. Indeed, the subject's traditional presentation forces teachers to introduce many different concepts: regular expressions, non-deterministic automata and deterministic automata. Switching back and forth between those concepts and showing how they relate can be time-consuming and may prove difficult for students to follow.

Although of similar expressiveness, regular expressions are often presented as a mere description tool that is unsuited for execution, whereas automata are seen as impractical to textually describe but very efficient to execute (Brüggemann-Klein, 1993). It would seem that you can not have your cake and eat it too. In the remainder of this chapter, I will attempt to hint at the contrary: Regular expressions, equipped with Brzozowski's derivation operation (Brzozowski, 1964) and represented using Huet's Zipper (Huet, 1997), can not only act as a declarative language description tool, but can also be efficiently executed. The presentation of the technique can be done without ever introducing automata theory. In Chapters 5 and 6, I will show that similar techniques can be applied to LL(1) and general context-free expressions to yield efficient parsing algorithms.

```scala
def convert(expr: RegExpr): NFA = {
  var states: Set[State] = Set.empty
  val transitions: Buffer[Transition] =
    new ArrayBuffer

  def createState(): State = {
    val state = freshState(states)
    states ++= Set(state)
    state
  }

  val finalState: State = createState()
  val errorState: State = createState()

  def go(expr: RegExpr, cont: State): State = expr match {
    case Epsilon => cont
    case Failure => errorState
    case Character(char) => {
      val state = createState()
      transitions += LabeledTransition(state, char, cont)
      state
    }
    case Disjunction(left, right) => {
      val state = createState()
      val leftState = go(left, cont)
      val rightState = go(right, cont)
      transitions += EpsilonTransition(state, leftState)
      transitions += EpsilonTransition(state, rightState)
      state
    }
    case Sequence(left, right) => {
      go(left, go(right, cont))
    }
    case Repetition(inner) => {
      val state = createState()
      val innerState = go(inner, state)
      transitions += EpsilonTransition(state, innerState)
      transitions += EpsilonTransition(state, cont)
      state
    }
  }

  val startState = go(expr, finalState)

  NFA(startState, Set(finalState), transitions)
}
```

Listing 2 – Conversion to NFA in continuation passing style.

2.4 Brzozowski's Derivatives

Regular expression derivatives were first described by Brzozowski (1964). In his influential paper, Janusz A. Brzozowski presents the operation as an elegant way to facilitate the conversion of regular expressions to state diagrams (i.e. automata). Interestingly, the operation is shown to be easily adapted to handle many additional regular expression operators, such as negation or intersection.

The derivative of a regular expression e by a character c, which I will denote by $\delta^c(e)$, is itself a regular expression whose language satisfies the following equality:

$$L(\delta^c(e)) = \{ \, w \mid c :: w \in L(e) \, \}$$

Or, expressed using the inductive predicate $\cdot \vdash \cdot$:

$$\forall w.\, \delta^c(e) \vdash w \iff e \vdash c :: w$$

Derivation is very naturally defined as a recursive function over the structure of the regular expression (see Figure 2.3a). The definition of derivation makes use of the concept of *nullability*: An expression is *nullable* if it accepts the empty sequence of character as part of its language. As seen in Figure 2.3b, nullability itself can be computed recursively.

$$\begin{aligned}
\delta^c(\bot) &:= \bot \\
\delta^c(\varepsilon) &:= \bot \\
\delta^c(c') &:= \begin{cases} \varepsilon & \text{if } c = c' \\ \bot & \text{otherwise} \end{cases} \\
\delta^c(e_1 \cdot e_2) &:= \begin{cases} \delta^c(e_1) \cdot e_2 \vee \delta^c(e_2) & \text{if } \texttt{null}(e_1) \\ \delta^c(e_1) \cdot e_2 & \text{otherwise} \end{cases} \\
\delta^c(e_1 \vee e_2) &:= \delta^c(e_1) \vee \delta^c(e_2) \\
\delta^c(e^*) &:= \delta^c(e) \cdot e^*
\end{aligned}$$

(a) Brzozowski's derivatives.

$$\begin{aligned}
\texttt{null}(\bot) &:= \texttt{false} \\
\texttt{null}(\varepsilon) &:= \texttt{true} \\
\texttt{null}(c) &:= \texttt{false} \\
\texttt{null}(e_1 \cdot e_2) &:= \texttt{null}(e_1) \;\texttt{\&\&}\; \texttt{null}(e_2) \\
\texttt{null}(e_1 \vee e_2) &:= \texttt{null}(e_1) \;\texttt{||}\; \texttt{null}(e_2) \\
\texttt{null}(e^*) &:= \texttt{true}
\end{aligned}$$

(b) Nullability of regular expressions.

Figure 2.3 – Definition of regular expression nullability and Brzozowski's derivatives as recursive functions.

Intuitively, derivation of an expression consists of replacing basic nodes that appear in left-most position by either ε or $\bot$, depending on whether or not they match the parameter character. In case of sequence expressions $e_1 \cdot e_2$, such left-most basic nodes may appear either:

- in the left expression e_1 and,
- in the right expression e_2 if e_1 is nullable.

Thus, in case e_1 is nullable, a disjunction node is introduced in the derivative to cover both cases. In case of repetition, a single iteration of the repetition is unrolled and recursively derived.

Initially, Brzozowski's derivatives were envisioned as a tool for building automata from regular expressions. The initial expression and its successive derivative expressions correspond to the states of the resulting automaton. Crucially, expressions are to be grouped together in equivalence classes, so that equivalent expressions are represented by the same state in the resulting automaton. Without this grouping of expressions in equivalence classes, the technique is not guaranteed to lead to a finite number of states. Checking expressions for equivalence is unfortunately a non-trivial task, which considerably complicates the seemingly simple approach proposed by Brzozowski. To do so, most approaches resort to building minimal deterministic finite-state automata, which completely defeats the purpose. Fortunately, weaker equivalence classes based on algebraic properties of regular expressions can be used to obtain a finite, but possible larger, number of states (Brzozowski, 1964; Owens et al., 2009). Brzozowski showed that rewriting expressions into a normal form based on the associativity, commutativity, and idempotence properties of disjunctions leads to a finite number of equivalence classes. Unfortunately, applying global transformations to rewrite expressions in a normal form complicates the approach.

2.4.1 Interpreter-style Membership Checking

Interestingly, Brzozowski's derivation can be straightforwardly implemented in functional programming languages. This natural embedding leads to the use of Brzozowski's derivation as part of an interpreter-style procedure for membership checking: Given an initial regular expression and a sequence of characters, one can check if the expression accepts the sequence of characters by iteratively applying derivation, once per input character, and then checking nullability of the resulting expression (Might, 2010). Doing so does not require explicitly transforming the regular expression into an automaton beforehand. Furthermore, support for *character classes* is trivial: For the reminder of the section, I will assume that `Character` accepts a *predicate* over characters instead of simple characters. This interpreter-style recognition technique is shown in Listing 3.

Although very elegant and straightforward, this interpreter is unfortunately often inefficient compared to the automaton-based approach. In the automaton-based representation, transitions to the next state are local and immediate: Computing the next state is a constant-time operation. On the other hand, computing the derivative of an expression is a seemingly global operation: With the canonical representation of regular expression as trees, the recursive derivation operation must always start from the root of the expression, whereas the basic subexpressions to be converted to ε or $\perp$ expressions are found at the leaves of the tree, possibly under layers and layers of accumulated operator nodes. Computing a derivation takes time linear in the size of the regular expression.

```scala
sealed trait RegExpr {
  val isNullable: Boolean = this match {
    case Epsilon => true
    case Disjunction(left, right) => left.isNullable || right.isNullable
    case Sequence(left, right) => left.isNullable && right.isNullable
    case Repetition(_) => true
    case _ => false
  }

  def derive(char: Char): RegExpr = {
    def down(expr: RegExpr): RegExpr = expr match {
      case Character(pred) => if (pred(char)) Epsilon else Failure
      case Disjunction(left, right) => Disjunction(down(left), down(right))
      case Sequence(left, right) if left.isNullable =>
        Disjunction(Sequence(down(left), right), down(right))
      case Sequence(left, right) => Sequence(down(left), right)
      case Repetition(inner) => Sequence(down(inner), expr)
      case _ => Failure
    }
    down(this)
  }

  def accepts(word: Iterable[Char]): Boolean =
    word.foldLeft(this) {
      case (derivative, char) => derivative.derive(char)
    }.isNullable
}
```

Listing 3 – Scala implementation of the `derive` and `isNullable` methods inspired by Brzozowski (1964). An interpreter-style `accepts` method is also provided.

Furthermore, if one is not careful, the size of derivative expressions may blow up, as would be the case by naïvely applying Brzozowski's derivation. Figure 2.4 shows how the size (number of nodes) of derivatives evolves with the number of derivations for a few selected regular expressions.

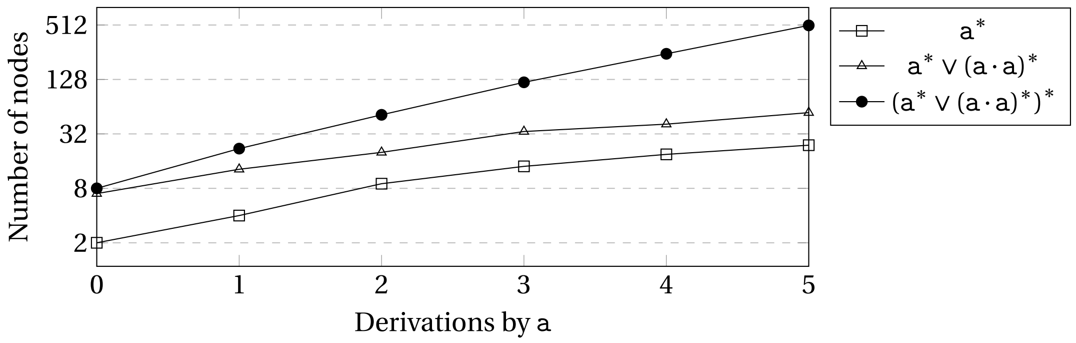


Figure 2.4 – Size of derivatives after a variable number of derivations by the character a. The y-axis is in logarithmic scale.

Figure 2.5 shows the results obtained by applying smart constructors to eliminate or propagate `Failure` and `Epsilon` expressions during derivation; a technique called *compaction* in some works (Might et al., 2011). The code for derivation with compaction is available in Appendix A. The figures show that, even with compaction, the size of derivatives can rapidly grow to be problematic. Furthermore, although compaction can mitigate the issue in certain cases, it is not able to completely eliminate it. Due to the possibly unbounded number of different derivative expressions, optimisation techniques such as memoisation become impractical. Later in this chapter, I will show how to completely do away with the issue by adopting a different representation of expressions and derivatives based on Huet's zipper.

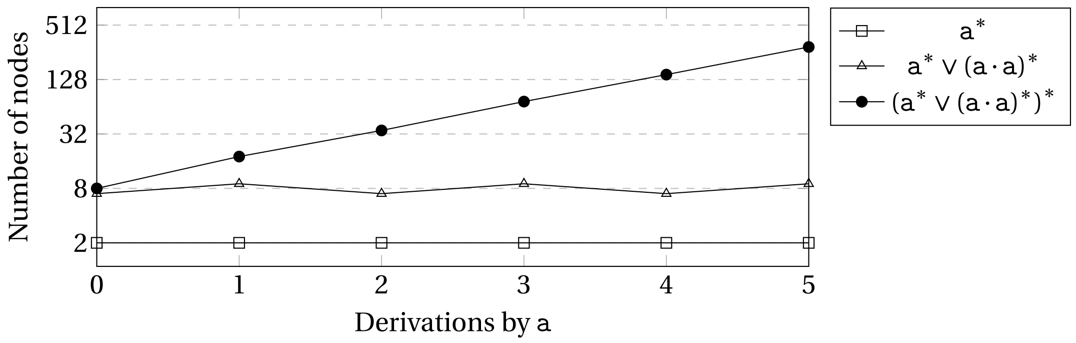


Figure 2.5 – Size of derivatives when applying a local *compaction* optimisation. The y-axis is in logarithmic scale.

2.5 Huet's Zipper

Given that the tree structure of regular expressions makes derivation behave badly, it is worth investigating different ways of representing regular expressions apart from top-down trees (Section 2.2). The *zipper* technique presented by Huet (1997) is such a way.

Huet's zipper is a simple technique that consists of adding a *focus* into a tree. The focus is a node in the data structure from which all references flow. In traditional tree-like structures, the focus is always located at the tree's root: All references always flow from that root. With a zipper, the focus can be moved around to child nodes or parent nodes.

```scala
sealed trait Tree[+A]
case class Branch[+A](left: Tree[A], right: Tree[A]) extends Tree[A]
case class Leaf[+A](value: A) extends Tree[A]

sealed trait Context[+A]
case object Empty extends Context[Nothing]
case class InLeft[+A](right: Tree[A], parent: Context[A]) extends Context[A]
case class InRight[+A](left: Tree[A], parent: Context[A]) extends Context[A]

case class Zipper[+A](focus: Tree[A], context: Context[A]) {
  def unfocus: Tree[A] = context match {
    case Empty => focus
    case InLeft(right, parent) => Zipper(Branch(focus, right), parent).unfocus
    case InRight(left, parent) => Zipper(Branch(left, focus), parent).unfocus
  }

  def moveUp: Zipper[A] = context match {
    case Empty => this
    case InLeft(right, parent) => Zipper(Branch(focus, right), parent)
    case InRight(left, parent) => Zipper(Branch(left, focus), parent)
  }

  def moveLeft: Zipper[A] = focus match {
    case Leaf(_) => this
    case Branch(left, right) => Zipper(left, InLeft(right, context))
  }

  def moveRight: Zipper[A] = focus match {
    case Leaf(_) => this
    case Branch(left, right) => Zipper(right, InRight(left, context))
  }

  def replaceFocus[B >: A](newFocus: Tree[B]): Zipper[B] =
    Zipper(newFocus, context)
}

def focus[A](tree: Tree[A]): Zipper[A] = Zipper(tree, Empty)
```

Listing 4 – Example implementation of a Zipper in Scala

Listing 4 shows a simple Scala implementation of a `Zipper` operating over a generic binary `Tree`. The zipper consists of a pair of a focused tree and a context. A `Context` is a stack of layers, each of which representing an ancestor of the currently focused subtree. The first layer corresponds to the tree's direct parent in the original tree structure, while the next layer corresponds to the parent's parent, and so on. Each context represents a path from the focus subexpression to the root of the expression. Layers contain information about the side of the child in which the focus is located, and keep a reference to the other child subtree. In case the focus is located in the left subtree, the layer keeps track of the right subtree, and conversely for the case when the focus is located in the right subtree.

Zippers support constant-time operations to move the focus to the parent node (`moveUp`), to the left child (`moveLeft`) or to the right child (`moveRight`). Trees can be converted to zippers, with a focus on the root, using the `focus` function. The function simply pairs the tree with the empty context `Empty`. Conversely, Zippers can be converted back to regular top-down trees by the recursive `unfocus` method.

Notably, the focus of a zipper can be replaced in constant-time. The function which implements this functionality in the example code is `replaceFocus`. The function discards the old focus and pairs the new focus with the current context. This focus replacement function can be used to remove or otherwise arbitrarily modify subtrees in focus.

McBride's Derivatives

Interestingly, McBride (2001) remarked that the *context* part of Huet's zippers corresponds to some sort of *derivative* of the original tree, hinting at some connection between datatypes and calculus. Thanks to Brzozowski's derivatives (Brzozowski, 1964), one can make a similar connection between regular expressions and calculus. "Like prolonged echoes mingling in the distance" (Baudelaire, 1855)[2], McBride's derivatives and Brzozowski's derivatives correspond. This correspondence suggests that adopting Huet's zipper to represent regular expressions and their derivatives might be appropriate, or at the very least worth trying!

In a later work, McBride (2008) generalised the technique to support different types for the left and the right elements appearing in the context. Having a separate type allows context elements to contain not only normal subtrees, but also arbitrary *views* of those subtrees. In this thesis, I will present data structures for representing regular expressions, as well as context-free expressions in Chapters 5 and 6, based on this idea. In such data structures, the elements appearing on the left of the focus will correspond to subtrees that have already been matched against the input.

[2]See the poem *Correspondances* by Charles Beaudelaire in the preface of this thesis.

2.6 Revisiting Brzozowski's Derivatives using Huet's Zipper

Zippers are, in essence, a way to record the context around nodes in a traversal of a tree-like structure. The `Zipper` data structure showed in Listing 4 is a specific example of this concept. It operates over a binary tree, supports a single focal point within that tree, and offers an interface to move this focus around the tree. I argue that the type of data structure operated over, the uniqueness of the focal point, as well as the interface to control the movement of the focal point, are not fundamental to the concept of zippers but are specific to that example. There exist instances of zippers which may vary some of those parameters.

In this section, I present an implementation in Scala of a simple lexical analysis library based on Brzozowski's derivatives and Huet's zipper. This implementation serves as the foundation to the SILEX library[3]. The implementation is backed by a formalisation in Coq[4].

The zipper that I will use differs in almost all aspects compared to the zipper of Listing 4: The zipper will operate on regular expressions, support multiple focal points, and will not offer an interface to arbitrarily move the focus. Instead of moving arbitrarily, focuses follow the recursive structure of Brzozowski's derivation operation, splitting into multiple focuses in case multiple recursive calls are made. As a result of following the structure of derivation, focal points will not appear at arbitrary points within regular expressions. Intuitively, focuses will always be on points located *at the front* of expressions. Indeed, derivation is only ever recursively called on subexpressions semantically located at the front of the expression. For this reason, there will not be any expressions to the left of the focal points. Contexts will represent *left-most* paths within the expression, and zippers will represent sets of such contexts. This set-based representation is reminiscent of the *partial derivatives* of Antimirov (1996).

2.6.1 A Zipper for Regular Expressions Derivatives

The type of zippers that I end up adopting is extremely simple. Each zipper is simply a *set* of contexts, each context being a *list* of regular expressions. The definition of `Zipper` and `Context` types is given in Listing 5. I will shortly discuss how one can arrive at this representation.

```
type Context = List[RegExpr]
type Zipper = Set[Context]
```

Listing 5 – Definition of the `Context` and `Zipper` types.

[3]https://github.com/epfl-lara/silex
[4]https://github.com/epfl-lara/silex-proofs

Giving Meaning to Zippers

Semantically, `Contexts` represent *sequences* of expressions, whereas `Zippers` represent *disjunctions* of such sequences. For instance, the zipper `Set(List(`e_1`, `e_2`), List(`e_3`))` is to be treated as semantically equivalent to the plain regular expression `Disjunction(Sequence(`e_1`, `e_2`), `e_3`)`.

This is reflected in the `unfocus` function (see Listing 6), which gives meaning to zippers in terms of regular expressions. Although the `unfocus` function is not used as part of the implementation, it is useful tool to reason about zippers.

```
def unfocus(zipper: Zipper): RegExpr = {
  def uncontext(context: Context): RegExpr =
    context.foldLeft[RegExpr](Epsilon)(Sequence)

  zipper.map(uncontext).foldLeft[RegExpr](Failure)(Disjunction)
}
```

Listing 6 – Definition of the `unfocus` function. The function is not used as part of the implementation, but helps providing meaning to zippers.

Arriving at the Definition of Zippers

Zippers are a way of representing the state of a traversal within a tree-like structure. Within zippers, focal values represent currently traversed nodes, whereas contexts represents the above nodes in the traversal. Such focal values and contexts are a faithful representation of the original data structure: from them, one can easily recover a corresponding unfocused tree.

In this chapter, the data structure that I operate on is `RegExpr`, and the traversal of interest is Brzozowski's derivation, or more precisely, successions of such derivations. During the execution of derivation on a zipper, the focus will be moved following the same recursive structure as Brzozowski's derivation. Instead of adding constructors on top of recursive calls, I will be recording those constructors and their already determined arguments in the contexts of zippers. Interestingly, Brzozowski's derivation only ever introduces two types of constructors:

- First, derivations can introduce sequences. Notice that when this happens, only the *left* side of the resulting sequence is further explored. Instead of building such sequences immediately on top of recursive calls, as would be the case in an unmodified version of derivation, our zippers will record the information needed to build such a sequence in the context as the focus is moved towards the left subexpression. In that case, the only information needed to reconstruct the sequence is its already determined right side.

- Secondly, derivations can also introduce disjunctions. When this happens, none of the sides of the resulting disjunctions is known a priori. The derivation function must be called recursively on the two subexpressions of the tree to determine both sides of the resulting disjunction. To support disjunctions, I will not be adding nodes to our contexts but instead *split* the focus. Each recursive call will be given a reference to the same context, upon which they can independently build. Thus, zippers will need to keep track of multiple pairs of focal values and contexts. Zippers represent *disjunctions* of such pairs.

Since disjunctions are commutative, associative and idempotent, representing zippers as *sets* is very natural. Furthermore, since, in the end, the only information contained within contexts is the right side of sequences, one can represent contexts using *lists* of regular expressions. Thus, contexts represent the list of all regular expressions that follow in sequence after the focal point.

Although typical zippers can represent arbitrary steps in a traversal of a tree, the zippers of this chapter will only be materialised in between each derivation. Add the end of a derivation, focal points will be only either `Epsilon` or `Failure` nodes, being the only possible results of applying Brzozowski's derivation on basic expressions. Interestingly, `Failure` nodes and their corresponding contexts can be entirely dropped from the zipper, leaving only `Epsilon` nodes. Since zippers will be materialised uniquely in between derivations, where the focal points are on `Epsilon` nodes, I will omit those values from the concrete representation of the present zippers. In the end, I obtain the very simple definition shown in Listing 5.

Converting Regular Expressions to Zippers

One can convert any regular expression into a zipper by using the `focus` function presented in Listing 7. Intuitively, the function focuses on a unique imaginary `Epsilon` node to the left of the expression. From that node's perspective, its `Context` consists of only one subsequent expression, which is the argument expression. Since the focal value of `Epsilon` bears no information, that node does not appear in the result: only the context is kept by the zipper.

```
def focus(expr: RegExpr): Zipper = Set(List(expr))
```

Listing 7 – Definition of the `focus` function, which converts a regular expression to an equivalent zipper.

Notice that `unfocus(focus(`e`))` is semantically equivalent to e for any regular expression e, ensuring that `focus` e is a faithful representation of e.

Theorem 2.1 (`unfocus_focus`)**.** *For any regular expression e and sequence of characters cs, the following holds:*

$$\mathtt{unfocus(focus(}e\mathtt{))} \vdash cs \iff e \vdash cs$$

Note on Proofs

I will not present proofs of the various lemmas and theorems that I state in this section. I instead refer interested readers to the proofs in Coq available online[5]. A reference to the name of the corresponding Coq theorem is given is parentheses before the theorem's statement. Given the order of lemmas and theorems, such proofs generally follow straightforwardly by induction.

2.6.2 Zipper-based operations

Now that I have an alternative way of representing regular expressions in the `Zipper` type, I can port operations that worked over regular expressions to this new zipper setting. With operations ported to zippers, I will only need to translate regular expressions to zippers once, using `focus`, and from then on be able to operate entirely on zippers.

Derivation

The first and most interesting operation to be ported is Brzozowski's derivation function. The zipper type that I adopt was designed with derivation in mind. Listing 8 presents the function `derive`, an implementation of the derivation function that operate on zippers instead of plain regular expressions. Intuitively, the function operates in two phases: an `up`-phase and a `down`-phase.

During the `up`-phase, the focuses are moved up the contexts to all regular expressions that can be reached without consuming any input. For this reason, the `up` helper function stops processing a context any further when it reaches a non-nullable expression.

During the `down`-phase, the focuses are moved down the regular expressions, following the same structure as Brzozowski's derivation. The contexts that end up reaching matching `Character` expressions are collected in a set and form the resulting zipper.

[5]https://github.com/epfl-lara/silex-proofs

```
def derive(zipper: Zipper, char: Char): Zipper = {
  def up(context: Context): Zipper = context match {
    case Nil => Set()
    case right :: parent if right.isNullable =>
      down(right, parent) ++ up(parent)
    case right :: parent => down(right, parent)
  }
  def down(expr: RegExpr, context: Context): Zipper = expr match {
    case Character(pred) if pred(char) => Set(context)
    case Disjunction(left, right) =>
      down(left, context) ++ down(right, context)
    case Sequence(left, right) if left.isNullable =>
      down(left, right :: context) ++ down(right, context)
    case Sequence(left, right) => down(left, right :: context)
    case Repetition(inner) => down(inner, expr :: context)
    case _ => Set()
  }
  zipper.flatMap(up)
}
```

Listing 8 – Definition of the `derive` function on zippers.

Note the striking similarity between the `down` helper function of Listing 8 and the `down` helper function of the original `derive` method as it was defined in Listing 3. The two functions share the same exact structure. Instead of representing the result as a normal `RegExpr`, the zipper-based version of `down` builds a set of `Context`s. The empty set plays the role of `Failure`, while the current context, wrapped in a set, plays the role of `Epsilon`. Set union is used instead of `Disjunction`, while `Sequence` is replaced by adding the subsequent expression to the context of recursive calls.

This zipper-based derivation function indeed computes a derivative in the sense of Brzozowski (1964), as shown by the following theorem.

Theorem 2.2 (`derive_correct_unfocus`)**.** *For any zipper z, character c and sequence of characters cs, the following holds:*

$$\texttt{unfocus(derive(z, c))} \vdash cs \iff \texttt{unfocus(z)} \vdash c :: cs$$

As shown in Listing 9, the derivation operation can be generalised from single characters to words of arbitrary length. The resulting function, `deriveWord`, successively applies derivation by all characters of the argument word.

```
def deriveWord(zipper: Zipper, word: Iterable[Char]): Zipper =
  word.foldLeft(zipper)(derive)
```

Listing 9 – Definition of the deriveWord function on zippers, which computes successive derivatives.

Theorem 2.3 (derive_word_correct_unfocus)**.** *For any zipper z and sequences of characters cs_1 and cs_2, the following holds:*

$$\texttt{unfocus(deriveWord(}z\texttt{,}\ cs_1\texttt{))} \vdash cs_2 \iff \texttt{unfocus(}z\texttt{)} \vdash cs_1 \mathbin{++} cs_2$$

Nullability

Nullability checks can also be easily ported to zippers. To check if a zipper is nullable, one simply checks if it contains a context that consists solely of nullable regular expressions. The resulting function, isNullable, is shown in Listing 10.

```
def isNullable(zipper: Zipper): Boolean =
  zipper.exists(_.forall(_.isNullable))
```

Listing 10 – Definition of the isNullable function.

Theorem 2.4 (nullable_correct_unfocus)**.** *For any zipper z, the following equivalence holds:*

$$\texttt{isNullable(}z\texttt{)} = \texttt{true} \iff \texttt{unfocus(}z\texttt{)} \vdash \langle\rangle$$

Membership Checking

Now that I have shown an implementation of the deriveWord and isNullable functions, everything is in place for the implementation of an accepts function operating on zippers. The function simply converts the argument regular expression into a zipper using focus, computes its derivative by the argument word using deriveWord, and then checks if that resulting zipper accepts the empty word using isNullable.

```
def accepts(expr: RegExpr, word: Iterable[Char]): Boolean =
  isNullable(deriveWord(focus(expr), word))
```

Listing 11 – An implementation of an accepts function using Brzozowski's derivatives on zippers.

Theorem 2.5 (accepts_correct)**.** *For any regular expression e and word cs, the following equivalence holds:*

$$\texttt{accepts(}e\texttt{,}\ cs\texttt{)} = \texttt{true} \iff e \vdash cs$$

```
def maxZipper(zipper: Zipper): Zipper = {
  def up(context: Context): Zipper = context match {
    case Nil => Set()
    case right :: parent => down(right, parent) ++ up(parent)
  }
  def down(expr: RegExpr, context: Context): Zipper = expr match {
    case Character(_) => Set(context)
    case Disjunction(left, right) =>
      down(left, context) ++ down(right, context)
    case Sequence(left, right) =>
      down(left, right :: context) ++ down(right, context)
    case Repetition(inner) => down(inner, expr :: context)
    case _ => Set()
  }
  zipper.flatMap(up)
}
```

Listing 12 – Definition of the `maxZipper` function. The function returns a zipper which is a superset of all successive derivatives of the argument zipper. The function is not used as part of the implementation.

2.6.3 Finiteness of Explorable Zippers

In the previous part, I have shown how one can represent regular expressions as zippers and how two crucial operations, Brzozowski's derivation and nullability checks, can be ported to zippers. At a first glance, operations on the zipper-inspired structure do not appear more efficient as on plain regular expressions. Although not inherently more efficient, the zippers from the previous part have a nice property: as I will show, the number of different zippers that can be visited by successive derivations is bounded. This fact opens the door to efficient memoisation.

From any zipper z, a *maximal zipper* `maxZipper(`z`)` can be constructed (see Listing 12). The structure of `maxZipper` resembles that of derivation, except that is it not subject to the restriction of only visiting subexpressions at the front of the regular expressions. As I shall prove, for any zipper z, the zipper `maxZipper(`z`)` is a superset of all the successive derivatives of z.

Lemma 2.1 (`derive_max_zipper_incl`)**.** *For any zipper z and character c:*

$$\mathtt{derive}(z,\ c) \subseteq \mathtt{maxZipper}(z)$$

Lemma 2.2 (`derive_max_zipper_mono`)**.** *For any zipper z and character c:*

$$\mathtt{maxZipper}(\mathtt{derive}(z,\ c)) \subseteq \mathtt{maxZipper}(z)$$

Lemma 2.3 (`derive_word_max_zipper`). *For any zipper z and non-empty sequences cs:*

$$\texttt{deriveWord(z, cs)} \subseteq \texttt{maxZipper(z)}$$

Theorem 2.6 (`finiteness`). *For any zipper z, their exists of set of contexts Z, such that for any (possibly empty) sequence of characters cs:*

$$\texttt{deriveWord(z, cs)} \subseteq Z$$

Counting Contexts

For any zipper z, the set Z of Theorem 2.6 can be constructed as the union of z with `maxZipper(`z`)`. The zipper z itself must be explicitly included, as it is not guaranteed to appear within `maxZipper(`z`)`, contrarily to all of its successive derivatives. This bounds the number of various zippers that can be encountered starting from any regular expression e to be at most:

$$1 + 2^{|\texttt{maxZipper(focus(}e\texttt{))}|}$$

The constant 1 represents the unique context of `focus(`e`)`, while $2^{|\texttt{maxZipper(focus(}e\texttt{))}|}$ returns the maximal number of different subsets of `maxZipper(focus(`e`))`.

Furthermore, one can easily see that the size of `maxZipper(focus(`e`))` is bounded by the number of `Character` nodes in the original expression e. I will denote by C this number. As expressed with this number, the maximal number of different zippers becomes:

$$1 + 2^C$$

When character classes are disjoint, we obtain a much better bound. I will denote by C_p the number of instances of `Character(`p`)` within e. Assuming no two different p accept the same character, we obtain the bound:

$$2 + \sum_p (2^{C_p} - 1)$$

The constant 2 accounts for both the unique context of `focus(`e`)` and the empty zipper $\emptyset$. This empty zipper is only counted once, and removed from all elements of the sum.

The finite number of different derivatives opens the door to efficient memoization, as I will discuss in Section 2.6.5.

2.6.4 Going Beyond Membership Checking

In practice, lexical analyzers go beyond simply checking if the input can be recognized in its entirely by a regular expression. Lexers generally repeatedly apply a procedure matching prefix of the input against a collection of regular expressions in a greedy fashion. Each time such a prefix is found an action is performed. This action generally consists of the emission of zero or more tokens. After the action is done, the process restarts on the remaining input. In this section, I explore the features required to enable this use case.

Stopping Early

In order to efficiently support prefix matching, it is primordial that processing stops as soon as possible. Each processing step should end as soon as all candidate regular expressions can no longer accept further input characters. To support this feature, I introduce a function to check if a zipper admits a least one non-empty word in its language (see Listing 13). The function makes use of two additional methods on regular expressions. The two methods respectively check if the expression accepts at least one word (`isProductive`), or at least one non-empty word (`hasFirst`).

Lemma 2.4 (`has_first_complete`)**.** *For any regular expression e:*

$$(\exists c, cs.\ e \vdash c :: cs) \implies \texttt{e.hasFirst}$$

Lemma 2.5 (`productive_complete`)**.** *For any regular expression e:*

$$(\exists cs.\ e \vdash cs) \implies \texttt{e.isProductive}$$

Theorem 2.7 (`has_first_zipper_complete_unfocus`)**.** *For any zipper z:*

$$(\exists c, cs.\ \texttt{unfocus(z)} \vdash c :: cs) \implies \texttt{hasFirst(z)}$$

Note that the converse of the above implications do not always hold. The culprit is the presence of empty character classes, that is expressions `Character(`p`)` where the predicate p is `false` for all possible characters. Such expressions are considered productive, even though they accept no sequences of characters. However, this unique direction is sufficient for our purposes. At the point where the lexer decides to stop processing further characters of input, this unique direction is sufficient to guarantee that the lexer indeed has found the longest possible match.

Specifying Rules

The behaviour of lexers is generally specified given a sequence of *rules*. Each rule associates a regular expression with an *action* to be executed when the input matches the expression. The action generally consists in generating tokens based on the matched string.

In the implementation, rules are simply pairs of a regular expression and an action (see Listing 14). The action is a function from an `ActionContext` to `Unit`. The `ActionContext` object provides the capability of emitting tokens, as well as retrieving the matched sequence of characters.

```
sealed trait RegExpr {
  val isNullable: Boolean = ... // As previously shown.

  val hasFirst: Boolean = this match {
    case Character(_) => true
    case Disjunction(left, right) => left.hasFirst || right.hasFirst
    case Sequence(left, right) => left.hasFirst && right.isProductive ||
                                  left.isNullable && right.hasFirst
    case Repetition(inner) => inner.hasFirst
    case _ => false
  }

  val isProductive: Boolean = isNullable || hasFirst
}

def hasFirst(zipper: Zipper): Boolean =
  zipper.exists(context => context.exists(_.hasFirst) &&
                           context.forall(_.isProductive))
```

Listing 13 – Definition of the `hasFirst` function on zippers, along with helper methods `hasFirst` and `isProductive` on regular expressions.

```
case class Rule[+T](expr: RegExpr, action: Action[T])

type Action[+T] = ActionContext[T] => Unit

trait ActionContext[-T] {
  def emit(token: T): Unit
  def content: String
}
```

Listing 14 – Definition of rules and actions. Each rule is a pair of a regular expression and an action. The action takes as parameter an `ActionContext`, which provides operations to emit tokens and retrieve the matched string.

Tokenisation

Everything is now in place for the introduction of the `tokenize` function (see Listing 15). The function is given as arguments a list of rules and a sequence of input characters. The function repeatedly matches a prefix of the remaining input characters with any of the given regular expressions, and executes the associated action.

To define which rule is to be applied, *maximal munch* is enforced: The rule that accepts longest possible prefix is to be selected. In addition, a *priority* scheme is enforced to break ties: Rules appearing earlier in the sequence are given priority over rules appearing later.

```
def tokenize[T](rules: List[Rule[T]], chars: String): Iterable[T] = {
  val tokens = new ArrayBuffer[T]()
  val zippers = rules.map(focus(_.expr))
  val actions = rules.map(_.action)
  val charsCount = chars.length
  var index = 0
  def context(result: String) = new ActionContext[T] {
    override def emit(token: T): Unit = tokens += token
    override val content: String = result
  }
  while (index < charsCount) {
    var current = zippers
    val startIndex = index
    var lastAction: Option[Action[T]] = None
    var lastIndex: Int = index
    while (index < charsCount && current.exists(hasNext)) {
      val nextChar = chars.charAt(index)
      current = current.map(derive(_, nextChar))
      val candidate = current.zip(actions).collectFirst {
        case (zipper, action) if isNullable(zipper) => action
      }
      if (candidate.nonEmpty) {
        lastIndex = index
        lastAction = candidate
      }
      index += 1
    }
    lastAction match {
      case None =>
        index = startIndex + 1
      case Some(action) =>
        val content = chars.substring(startIndex, lastIndex + 1)
        action(context(content))
        index = lastIndex + 1
    }
  }
  tokens
}
```

Listing 15 – Definition of the `tokenize` function, which processes sequences of characters according to a list of rules.

In case no accepting rule is found, the first remaining character of input is discarded, and the process resumes one character later. It is possible to change this behaviour by introducing a rule accepting any single character at the lowest priority level. The action associated with such a rule is guaranteed to be executed in case no other rule matches.

Note on Backtracking

Note that *backtracking* may take place: The function might need to place its read pointer at an earlier position in the input sequence after matching a rule, leading to some characters being processed multiple times. Indeed, reading a character of input does not guarantee eventual acceptance. The tokeniser might be required to *lookahead* some number of characters. This behaviour is exhibited by most lexical analysis tools (Levine, 2009; Lesk and Schmidt, 1975; Klein, 2010) and is largely unproblematic as the required lookahead is often small.

However, it is possible to devise regular expressions that require arbitrarily many characters of lookahead. As a pathologic example of this, consider the behaviour of the tokeniser on regular expression $\texttt{a}^* \cdot \texttt{b}$ (any number of `a`'s followed by a single `b`) and input consisting of n `a` characters. After processing the entire input of `a`'s for the first time, and realising that there are no `b`'s in sight, the tokeniser function resumes its execution, this time starting at the second character of input. The process continues and on and, each time resuming one character later in the input. In the end, the function exhibits quadratic running time. In practice, this problematic behaviour is often easily avoided. It generally suffices to add rules that match against prefixes of problematic rules (Klein, 2010). In this specific example, adding a rule for $\texttt{a}^*$ suffices to fix the issue.

Reps (1998) showed a technique which completely avoids this quadratic behaviour arising from backtracking, and so without requiring user intervention. The technique is orthogonal to the zipper-based approach that I present in this chapter, and thus could be easily incorporated should it be deemed appropriate.

Note on Streaming

In the interest of simplicity, the function shown in Listing 15 operates on an input `String`. The function thus requires the entire input to be available in its entirety before processing starts. However, the function can be straightforwardly adapted to operate on lazy streams of input characters, represented for instance using Java's `Reader` class[6], instead. In such a case, the methods `mark` and `reset` can be used in case of backtracking to return the reader head to the last accepted position.

In the same vein, the resulting tokens can also be made available as soon as they are emitted. Instead of returning an explicit collection of tokens, the function can be adapted to return

[6]https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html

an iterator over emitted tokens. In this fashion, the input can be processed on demand as elements are queried from the iterators. Those features are available in `silex`[7], the full-fledged implementation of the library.

2.6.5 To Automata via Memoization

The fact that only a finite number of different zippers can be encountered by successive derivations creates an opportunity to effectively apply memoization. Indeed, the list of zippers, as manipulated by the `tokenize` function of Listing 15, can only range over a finite number of different values. In this section, I define a `State` class (see Listing 16) to record data associated with each such list of zippers.

```
trait State[+A] {
  def next(char: Char): State[A]
  val hasNext: Boolean
  val value: Option[A]
}
```

Listing 16 – Definition of the `State` datatype, which holds the state associated with a list of zippers.

Each `State` object consists of:

- A transition function to other states.
- A boolean indicating if the state can lead to accepting states following the transition function.
- An optional value, which is set in case the set is accepting. In the present setting, the value associated to accepting states is a unique *action*.

The `build` function (see Listing 17) converts a list of rules into a finite-state automaton represented by its initial state. The automaton is lazily built as transitions from the initial and subsequent states are explored.

The `build` function first converts the list of rules given as parameters into a list of zippers, which it feeds to the `getState` helper function. The `getState` helper function returns the state associated with a given list of zippers. Importantly, the function is memoized using the `states` hash map. The function is guaranteed to return the same exact state given equal lists of zippers. In case the entry has not been already memoized, a new state is created using the aptly named `newState` helper function. That function creates a new state with the appropriate properties as computed from the list of zippers given as arguments.

[7]https://github.com/epfl-lara/silex

```
def build[A](rules: List[Rule[A]]): State[Action[A]] = {
  val states: HashMap[List[Zipper], State[Action[A]]] = new HashMap()
  def newState(zippers: List[Zipper]): State[Action[A]] = new State {
    val nexts: LongMap[State[Action[A]]] = new LongMap()
    override def next(char: Char): State[Action[A]] =
      nexts.getOrElseUpdate(char.toLong, getState(zippers.map(derive(_, char))))
    override val hasNext = zippers.exists(hasFirst)
    override val value: Option[Action[A]] = zippers.zip(rules).collectFirst {
      case (zipper, rule) if isNullable(zipper) => rule.action
    }
  }
  def getState(zippers: List[Zipper]): State[Action[A]] =
    states.getOrElseUpdate(zippers, newState(zippers))
  getState(rules.map(rule => Set(List(rule.expr))))
}
```

Listing 17 – Definition of the `build` function, which lazily builds a deterministic finite-state automaton corresponding to a collection of rules.

For each state, the transition function computes the derivatives of the associated list of zippers and then queries the state of the resulting list of derivative zippers using the previously defined `getState` helper function. Importantly, the transitions functions of the various states are also memoised: Results of the transitions functions are stored in a map local to each state and indexed by characters.

Note the use of two different memoised sets of functions. On the one hand, the unique `states` map, which is indexed by lists of zippers, ensures the uniqueness of the states associated with each such list. On the other hand, the `nexts` maps local to each state, which significantly decrease the cost of taking transitions in the resulting automaton.

The `tokenize` function can be easily adapted to operate on `State` automata. Listing 18 presents the adapted implementation. The only difference compared to the previous implementation of Listing 15 is that `tokenize` no longer manipulates lists of zippers but `State` objects.

Note that the `States`' transition function is called in the tight main loop of the tokeniser function. Thanks to memoisation, taking transitions will generally be very fast: In the best case, transitioning between states only consists of a single lookup in a character-indexed map.

```
def tokenize[T](initial: State[Action[T]], chars: String): Iterable[T] = {
  val tokens = new ArrayBuffer[T]()
  val charsCount = chars.length
  var index = 0
  def context(result: String) = new ActionContext[T] {
    override def emit(token: T): Unit = tokens += token
    override val content: String = result
  }
  while (index < charsCount) {
    var current: State[Action[T]] = initial
    val startIndex = index
    var lastAction: Option[Action[T]] = None
    var lastIndex: Int = index
    while (index < charsCount && current.hasNext) {
      val nextChar = chars.charAt(index)
      current = current.next(nextChar)
      if (current.value.nonEmpty) {
        lastIndex = index
        lastAction = current.value
      }
      index += 1
    }
    lastAction match {
      case None =>
        index = startIndex + 1
      case Some(action) =>
        val content = chars.substring(startIndex, lastIndex + 1)
        action(context(content))
        index = lastIndex + 1
    }
  }
  tokens
}
```

Listing 18 – Definition of the memoized `tokenize` function. The function runs the `State` automaton on the input until either the end of the input or a trap state, recording the last accepting state.

2.6.6 Evaluation

In this section, I evaluate the performance of the approach both in terms of speed and in terms of number of states of the resulting automata.

Speed

Figure 2.6 shows the time taken by three different implementations of lexical analysers that count the number of non-whitespace tokens in a collection of randomly generated JSON files (Omanashvili, 2019). The first implementation is a simple interpreter-style derivative-based tokeniser that implements compaction (Compaction). Note that this first implementation does *not* feature the zipper and memoisation techniques discussed in this chapter. The second implementation uses the library showcased in this chapter (SILEX). Finally, the last implementation was generated by the `JFlex` (Klein, 2010) generator (`JFlex`). All three approaches were given an equivalent specification of the token classes. The benchmarks were run on a 2018 MacBook Pro with a 2.2 GHz Intel Core i7 processor. I used Java 1.8 and Scala 3.0.0-RC1 running on top of the Java HotSpot™ virtual machine. Each data point corresponds to the mean result of a hundred runs on a hot virtual machine.

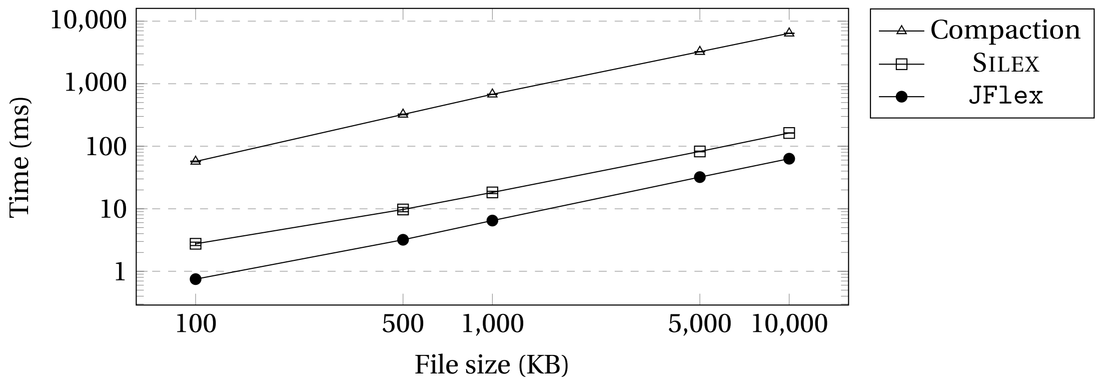


Figure 2.6 – Time required to count the number of valid tokens in randomly generated JSON files of various sizes. Both axes are in logarithmic scale.

The performance of the approach shown in this chapter is orders of magnitude better than the simple interpreter-style implementation featuring compaction. The approach is slower than the `JFlex`-generated analyser, but no more than by a factor three. This small factor is encouraging, especially given the speed advantage and optimisation opportunities offered by code generation.

On the JSON examples, the example implementation processes input at a rate of approximately 55-60MB per second, which seems reasonable for many practical applications.

Number of states

Table 2.1 reports the maximal number of states that the lazily built automata resulting from the approach admit in a variety of examples. I use example regular expressions taken from both JSON and ANSI C, as well as expressions corresponding to the full languages. I compare the number of states encountered with the number of states created by the `JFlex` (Klein, 2010) parser generator. The `JFlex` generator first computes an NFA from the given regular expressions, then determinises the NFA into a DFA, which is finally minimised. I report the size of all three types of automata in the table. Automata minimised by `JFlex` have sizes that are as small as theoretically possible. Both the `JFlex` lexers and the lexers following the approach shown in this chapter have been written by myself and correspond closely to one another. In the case of ANSI C, the lexers' implementations follow that of Degener (1995).

	Expressions	`JFlex`			SILEX
		NFA	DFA	min. DFA	
JSON	Keyword `true`	9	7	6	6
	Keyword `false`	10	8	7	7
	Keyword `null`	9	7	6	6
	White spaces	8	5	3	3
	Number	30	14	10	10
	String	32	13	10	10
	Full JSON	118	47	38	43
ANSI C	Comment	26	9	6	6
	Integer	14	8	7	7
	Float	58	25	19	23
	Identifier	10	5	3	3
	Full ANSI C	483	248	228	240

Table 2.1 – Number of states of the various automata. The `JFlex` entries correspond to the numbers reported by the lexer generator itself during code generation. To obtain the entries corresponding to SILEX, I forced the evaluation of the lazily built automata and reported the number of entries in the state memoization table.

2.6.7 Building a Domain-Specific Interface

In the previous parts of this chapter, I have shown how to design an efficient lexical analysis library in Scala based on derivatives and zippers. Thanks to the use of memoization, I arrived at a highly efficient library. The interface to the library, although serviceable, is not very user-friendly at this point. In this section, I show how one can take advantage of the excellent support of Scala for defining embedded domain-specific languages to offer a polished interface to the library.

An Interface for Regular Expressions

The first improvement is a friendlier interface for defining regular expressions. I add a collection of infix binary and postfix unary combinators as *extension methods* of regular expressions. Extensions methods are a mechanism of Scala for adding methods to a class after its definition. The combinators naming scheme corresponds to the usual notation for regular expressions Sipser (2012). I also add the `elem` and `word` functions to build regular expressions corresponding respectively to single character and sequences of characters.

```
extension (expr: RegExpr) {
  def ~(that: RegExpr): RegExpr = Sequence(expr, that)
  def |(that: RegExpr): RegExpr = Disjunction(expr, that)
  def * : RegExpr = Repetition(expr)
  def ? : RegExpr = expr | Epsilon
  def + : RegExpr = expr ~ expr.*
  def times(n: Int): RegExpr =
    if (n <= 0) Epsilon else expr ~ expr.times(n - 1)
}

def elem(pred: Char => Boolean): RegExpr = Character(pred)
def elem(char: Char): RegExpr = Character(_ == char)
def elem(chars: Iterable[Char]): RegExpr =
  chars.map(elem).foldLeft[RegExpr](Failure)(_ | _)
def word(chars: Iterable[Char]): RegExpr =
  chars.map(elem).foldLeft[RegExpr](Epsilon)(_ ~ _)
def inRange(low: Char, high: Char): RegExpr = elem(c => c >= low && c <= high)
```

Listing 19 – Combinators API for regular expressions in Scala.

An Interface for Rules

The second improvement is a polished interface for defining rules. In Listing 20, I show the definition of the `|>` infix operator as an extension method of `RegExpr`. The operator combines a regular expression and an action into a proper `Rule` object. The left-hand side of the operator is a regular expression, which can be written using the combinators discussed in the previous paragraphs. The `|>` operator binds more weakly than all defined operators.

The second argument of the `|>` operator is an action. In the latest released version of Scala at the time of writing[8], one can take advantage of *context functions* (Odersky et al., 2017) to offer a clutter-free interface for defining such actions. The parameter action of the `|>` method expects a function with an *implicit* parameter of type `ActionContext[T]`. The type of such a function is denoted using an `?=>` arrow instead of the `=>` arrow of normal functions. I then

[8]Scala version 3.0.0-RC1

```
extension (expr: RegExpr) {
  def |>[T](action: ActionContext[T] ?=> Unit): Rule[T] =
    Rule(expr, (context: ActionContext[T]) => action(using context))
}

def emit[T](token: T)(using context: ActionContext[T]): Unit =
  context.emit(token)

def content[T](using context: ActionContext[T]): String =
  context.content
```

Listing 20 – Using context functions to provide an interface for defining rules.

define the `emit` and `content` functions which expect an implicit `ActionContext[T]` in the environment. In Listing 21, I give an example of user code making use of such an interface. The definitions of the `string`, `number`, and `spaces` regular expressions is omitted.

2.6.8 Flexibility

As any interpreter-style lexer, the approach that I have shown can be easily modified to support features such as positions, arbitrary character types, stream processing, or non-blocking interfaces (Might, 2010). I have implemented some of those features in SILEX[9], the full-fledged lexical analysis library building on the approach shown in this chapter.

Thanks to the embedding in Scala, the definition of the lexer can make use of the abstraction capabilities offered by a high-level general purpose language. In contrast, the abstraction capabilities offered in the input language of code generators such as `lex`, `flex`, or `JFlex`, are often very limited.

Furthermore, thanks to the flexible dynamic nature of our approach, features such as lexer reconfiguration at runtime are made possible. Such features are often harder to implement in static generator-based approaches.

2.6.9 Applicability

The approach can be easily implemented in most functional programming languages. In this chapter, I have presented a simple Scala implementation. To prove the correctness of the technique, I have also built an implementation in Coq[10]. To provide further evidence of the applicability of our approach, I show an implementation of a simple membership-checking procedure technique in 56 lines of Haskell in Appendix B.

In later chapters of this thesis, I will show how one can adapt this derivatives-and-zippers

[9]https://github.com/epfl-lara/silex
[10]https://github.com/epfl-lara/silex-proofs

```
val digit = elem(_.isDigit)
val nonZero = inRange('1', '9')
val hex = digit | inRange('A', 'F') | inRange('a', 'f')
val whiteSpace = elem(_.isWhitespace)
val encodedChar = elem('u') ~ hex.times(4)
val escapedChar = elem("\"\\/bfnrt")
val stringChar =
  elem(c => c != '"' && c != '\\' && !c.isControl) |
  elem('\\') ~ (escapedChar | encodedChar)
val wholePart = elem('0') | nonZero ~ digit.*
val fractPart = elem('.') ~ digit.+
val exponent = elem("eE") ~ elem("+-").? ~ digit.+

val jsonRules = List[Rule[String]](
  whiteSpace.+ |> { },
  word("null") |> { emit(NullToken) },
  word("true") |> { emit(BoolToken(true)) },
  word("false") |> { emit(BoolToken(false)) },
  elem('[') |> { emit(OpenSquareToken) },
  elem(']') |> { emit(CloseSquareToken) },
  elem('{') |> { emit(OpenCurlyToken) },
  elem('}') |> { emit(CloseCurlyToken) },
  elem(',') |> { emit(CommaToken) },
  elem(':') |> { emit(ColonToken) },
  elem('"') ~ stringChar.* ~ elem('"') |> {
    emit(StringToken(content)) },
  elem('"') ~ stringChar.* |> { emit(InvalidToken(content))) },
  elem('-').? ~ wholePart ~ fractPart.? ~ exponent.? |> {
    emit(NumberToken(content.toDouble)) },
  elem(_ => true) |> { emit(InvalidToken(content)) }
)
```

Listing 21 – Definition of the rules of a JSON lexer using the discussed interface.

mingling technique with great effectiveness to context-free expressions, and notably LL(1) context-free expressions.

2.7 Pumping Lemma on Regular Expressions

To conclude this chapter on regular expressions, I show a proof of the *pumping lemma* for regular languages that operates on regular expressions. The proof is shown here as further evidence that regular expressions form a sufficiently expressive framework on their own, without needing to introduce deterministic and non-deterministic automata. Traditional proofs of the pumping lemma for regular languages usually operate at the level of automata. In contrast, the following proof operates directly on regular expressions. The proof is short and fully constructive. Contrary to traditional proofs of the lemma, it does not make use of the pigeon hole principle and is easy to formalise in proof assistants such as Coq. Such a proof, with looser constants, is offered as an advanced exercise in Pierce et al. (2018)[Chapter on inductively defined propositions], further demonstrating the amenability of the approach to formal proofs. Ammann (2021) also presents a mechanised proof of the lemma in Coq that she developed as part of her bachelor project that I supervised.

Lemma 2.6 (Pumping)**.** *Given a regular expression e, there exists a constant number $p \geq 1$, called the* pumping constant*, such that for all words w of length at least p (that is, $|w| \geq p$) in the language of e (that is, $e \vdash w$), there exists three sequences xs, ys, and zs, with the following properties:*

1. $xs \mathbin{++} ys \mathbin{++} zs = w$
2. $|ys| \geq 1$
3. $|xs \mathbin{++} ys| \leq p$
4. $\forall n.\ e \vdash xs \mathbin{++} ys^n \mathbin{++} zs$

Where ys^n is the sequence that consists of the concatenation of n instances of ys.

Proof. The proof proceeds by structural induction on the parameter regular expression e.

1. In the base cases, that is $e = \bot$, $e = \varepsilon$ or $e = c$, there exists at most one sequence accepted by e. In those cases, it suffices to choose a pumping constant p larger than the size of that word, if any. The proposition, which is universally quantified over the words of at least size p, then vacuously holds. In the case of $e = \bot$ and $e = \varepsilon$, the pumping constant p is 1, and in case of $e = c$, the pumping constant p is 2.

2. Consider the case $e = e_1 \vee e_2$. By induction hypothesis, one gets pumping constants p_1 for e_1 and p_2 for e_2. I assert that the maximum of p_1 and p_2 is a valid pumping constant p for e. The goal is now to show that any word w of sufficiently large size can be appropriately decomposed. Let w be a word accepted by e of size at least p. By construction, it must be the case that either:

   - $e_1 \vdash w$, or
   - $e_2 \vdash w$.

   Without loss of generality, let us assume that $e_1 \vdash w$. By induction hypothesis on e_1, since $|w| \geq p = \max(p_1, p_2) \geq p_1$, one gets three sequences xs_1, ys_2, zs_2 with the following properties:

   (a) $xs_1 \mathbin{++} ys_1 \mathbin{++} zs_1 = w$

   (b) $|ys_1| \geq 1$

(c) $|xs_1 \mathbin{++} ys_1| \leq p_1$

(d) $\forall n.\ e_1 \vdash xs_1 \mathbin{++} ys_1^n \mathbin{++} zs_1$

I assert that the sequences $xs = xs_1$, $ys = ys_1$, and $zs = zs_1$ form a valid decomposition of w with respect to e. Indeed:

(a) $xs \mathbin{++} ys \mathbin{++} zs = xs_1 \mathbin{++} ys_1 \mathbin{++} zs_1 = w$

(b) $|ys| = |ys_1| \geq 1$

(c) $|xs \mathbin{++} ys| = |xs_1 \mathbin{++} ys_1| \leq p_1 \leq \max(p_1, p_2) = p$

(d) $\forall n.\ e \vdash xs \mathbin{++} ys^n \mathbin{++} zs$. Indeed, for any n, $e_1 \vdash xs_1 \mathbin{++} ys_1^n \mathbin{++} zs_1$ and thus, by construction, $e_1 \vee e_2 \vdash xs_1 \mathbin{++} ys_1^n \mathbin{++} zs_1$.

3. Consider the case $e = e_1 \cdot e_2$. By induction hypothesis, one gets pumping constants p_1 for e_1 and p_2 for e_2. I assert that $p_1 + p_2 - 1$ is a valid pumping constant p for e. Let w be a word accepted by e of size at least p. By construction, there must exist two sequences w_1 and w_2 such that:

   (a) $w_1 \mathbin{++} w_2 = w$

   (b) $e_1 \vdash w_1$

   (c) $e_2 \vdash w_2$

   Trivially, either $|w_1| \geq p_1$ or $|w_1| < p_1$.

   - Consider the case $|w_1| \geq p_1$. By induction hypothesis on e_1, one gets three sequences xs_1, ys_1, zs_1 with the following properties:

     (a) $xs_1 \mathbin{++} ys_1 \mathbin{++} zs_1 = w_1$

     (b) $|ys_1| \geq 1$

     (c) $|xs_1 \mathbin{++} ys_1| \leq p_1$

     (d) $\forall n.\ e_1 \vdash xs_1 \mathbin{++} ys_1^n \mathbin{++} zs_1$

     I assert that $xs = xs_1$, $ys = ys_1$ and $zs = zs_1 \mathbin{++} w_2$ form a valid decomposition of w with respect to e. Indeed:

     (a) $xs \mathbin{++} ys \mathbin{++} zs = xs_1 \mathbin{++} ys_1 \mathbin{++} zs_1 \mathbin{++} w_2 = w_1 \mathbin{++} w_2 = w$

     (b) $|ys| = |ys_1| \geq 1$

     (c) $|xs \mathbin{++} ys| = |xs_1 \mathbin{++} ys_1| \leq p_1 \leq p_1 + p_2 - 1 = p$

     (d) $\forall n.\ e \vdash xs \mathbin{++} ys^n \mathbin{++} zs$. Indeed, for any n, $e_1 \vdash xs_1 \mathbin{++} ys_1^n \mathbin{++} zs_1$ and $e_2 \vdash w_2$, thus by construction $e_1 \cdot e_2 \vdash xs_1 \mathbin{++} ys_1^n \mathbin{++} zs_1 \mathbin{++} w_2$.

   - Consider the case $|w_1| < p_1$. By $|w| \geq p$, $|w_1| + |w_2| = |w|$ and $p = p_1 + p_2 - 1$, one gets:

$$|w_1| + |w_2| \geq p_1 + p_2 - 1$$

Subtracting $|w_1|$ from the left size and the strictly larger value p_1 from the right size, one gets the following strict inequality:

$$|w_2| > p_2 - 1$$

Or, equivalently, $|w_2| \geq p_2$. Thus, by induction hypothesis on e_2, one gets three sequences xs_2, ys_2, zs_2 with the following properties:

(a) $xs_2 ++ ys_2 ++ zs_2 = w_2$
(b) $|ys_2| \geq 1$
(c) $|xs_2 ++ ys_2| \leq p_2$
(d) $\forall n.\ e_2 \vdash xs_2 ++ ys_2^n ++ zs_2$

I assert that $xs = w_1 ++ xs_2$, $ys = ys_2$ and $zs = zs_2$ form a valid decomposition of w with respect to e. Indeed:

(a) $xs ++ ys ++ zs = w_1 ++ xs_2 ++ ys_2 ++ zs_2 = w_1 ++ w_2 = w$
(b) $|ys| = |ys_2| \geq 1$
(c) $|xs ++ ys| = |w_1 ++ xs_2 ++ ys_2| = |w_1| + |xs_2 ++ ys_2| \leq (p_1 - 1) + p_2 = p$
(d) $\forall n.\ e \vdash xs ++ ys^n ++ zs$. Indeed, for any n, $e_1 \vdash w_1$ and $e_2 \vdash xs_2 ++ ys_2^n ++ zs_2$, thus by construction $e_1 \cdot e_2 \vdash w_1 ++ xs_2 ++ ys_2^n ++ zs_2$.

4. Consider the case $e = e_1{}^*$. By induction hypothesis, one gets pumping constants p_1 for e_1. I assert that p_1 is a valid pumping constant p for e. Let w be a word accepted by e of size at least p. Let w_1, w_2 be two sequences such that:

   (a) $e_1 \vdash w_1$
   (b) $e_1{}^* \vdash w_2$
   (c) $|w_1| \geq 1$

   The condition (c) ensures that the sequence matched by e_1 is non-empty. Although quite intuitive, such a decomposition of w into w_1 and w_2 would formally require an auxiliary lemma. In the interest of space, this small lemma is omitted. It however follows straightforwardly by induction on the derivation of $e_1{}^* \vdash w_2$.

   Trivially, either $|w_1| \geq p_1$ or $|w_1| < p_1$.

   - Consider the case where $|w_1| \geq p_1$. In this case, by induction hypothesis on e_1, one gets three sequences xs_1, ys_1, zs_1 with the following properties:

     (a) $xs_1 ++ ys_1 ++ zs_1 = w_1$
     (b) $|ys_1| \geq 1$
     (c) $|xs_1 ++ ys_1| \leq p_1$
     (d) $\forall n.\ e_1 \vdash (xs_1 ++ ys_1^n ++ zs_1)$

     I assert that $xs = xs_1$, $ys = ys_1$ and $zs = zs_1 ++ w_2$ form a valid decomposition of w with respect to e. Indeed:

(a) $xs \mathbin{++} ys \mathbin{++} zs = xs_1 \mathbin{++} ys_1 \mathbin{++} zs_1 \mathbin{++} w_2 = w_1 \mathbin{++} w_2 = w$
(b) $|ys| = |ys_1| \geq 1$
(c) $|xs \mathbin{++} ys| = |xs_1 \mathbin{++} ys_1| \leq p_1 = p$
(d) $\forall n.\ e \vdash xs \mathbin{++} ys^n \mathbin{++} zs$. Indeed, for any n, $e_1 \vdash xs_1 \mathbin{++} ys_1^n \mathbin{++} zs_1$ and ${e_1}^* \vdash w_2$, thus by construction ${e_1}^* \vdash xs_1 \mathbin{++} ys_1^n \mathbin{++} zs_1 \mathbin{++} w_2$.

- Consider the case where $|w_1| < p_1$. I assert that $xs = \langle\rangle$, $ys = w_1$, and $zs = w_2$ form a valid decomposition of w with respect to e. Indeed:

(a) $xs \mathbin{++} ys \mathbin{++} zs = \langle\rangle \mathbin{++} w_1 \mathbin{++} w_2 = w_1 \mathbin{++} w_2 = w$
(b) $|ys| = |w_1| \geq 1$
(c) $|xs \mathbin{++} ys| = |\langle\rangle \mathbin{++} w_1| = |w_1| \leq p_1 = p$
(d) $\forall n.\ e \vdash (xs \mathbin{++} ys^n \mathbin{++} zs)$.
This last fact is proven by induction on n. In the base case, one has to show:

$$e \vdash xs \mathbin{++} ys^0 \mathbin{++} zs$$

Which is trivially equivalent to the assumption ${e_1}^* \vdash w_2$.
For the inductive case, one has to show:

$$e \vdash xs \mathbin{++} ys^{n+1} \mathbin{++} zs$$

With the following induction hypothesis:

$$e \vdash xs \mathbin{++} ys^n \mathbin{++} zs$$

The induction hypothesis can be rewritten as:

$${e_1}^* \vdash w_1^n \mathbin{++} w_2$$

Finally, from this induction hypothesis and from $e_1 \vdash w_1$, one can construct:

$${e_1}^* \vdash w_1 \mathbin{++} w_1^n \mathbin{++} w_2$$

This proposition is trivially equivalent to the goal:

$$e \vdash xs \mathbin{++} ys^{n+1} \mathbin{++} zs$$

This concludes the proof of the pumping lemma by structural induction on regular expressions.

□

2.8 Conclusion

In this chapter, I have given an introduction to the well-known concept of regular expressions. I have shown a short and fast lexical analysis library based on Brzozowski's derivatives and

Huet's zippers. The library does not require any preprocessing on regular expressions, and ends up building a deterministic finite-state automaton as it memoises derivatives. The zipper-based representation allows for efficient grouping of expressions and their derivatives in equivalence classes. Interestingly, as the automaton is a mere artefact of memoisation, the technique can be understood without any prior knowledge of the theory of deterministic and non-deterministic automata. To further exemplify that regular expressions may not need to be transformed into automaton form to be useful, I presented a proof of the regular pumping lemma operating on regular expressions instead of automata as would be typically the case.

For the remainder of this thesis, we shall leave behind the realm of regular languages and enter that of context-free languages. The formalism I will be working on, *context-free expressions*, is an extension of regular expressions. Interestingly, as we will discover in later chapters, the same combination of Brzozowski's derivatives and Huet's zipper will prove fruitful also in that setting, though the reasons why will differ.

3 Value-Aware Context-Free Expressions

Since their description by Noam Chomsky in the 1950s (Chomsky, 1956), *context-free grammars* have become the de-facto standard for describing the syntax of programming languages. Context-free grammars, often represented in some flavour of Backus-Naur Form (Backus et al., 1960), are used as inputs to many parser generator tools and as the formal description of the syntax of most programming languages.

In practice, grammars are used only as the initial description of the language. A parser generator's purpose is to convert this description to some representation suitable for manipulation by a parsing algorithm. Parsers represent their states using ad hoc data structures such as tables and stacks that often appear quite remote from grammars. I argue that this disparity between grammars and parser states makes it more challenging to understand parsers and parsing algorithms, and it makes it more difficult to adapt their functionality.

In this chapter, I formalise the notion of *context-free expressions*. Context-free expressions, as opposed to context-free grammars, prove a suitable tool to both serve as a description of parsers and a description of parser states. This chapter serves as a formal foundation for the following chapters, in which I will highlight parsing algorithms that operate on context-free expressions.

The idea of context-free expressions dates back at least to Leiß (1991). Context-free expressions are an extension of regular expressions where the Kleene star operator has been replaced by an ability to refer to a finite set of named expressions, typically through a least-fixed-point operator. Contrary to grammars, context-free expressions, due to their expression-based nature, can be encoded rather naturally by using a hierarchical tree-like data structure. Context-free expressions also bear striking similarities with the *parser combinators* that are often used to write parsers in functional programming languages (Burge, 1975; Hutton, 1992; Hutton and Meijer, 1996; Danielsson, 2010).

The presentation of context-free expressions offered in this chapter differs however significantly from the one found in earlier works (Leiß, 1991; Krishnaswami and Yallop, 2019):

1. Instead of introducing a local least-fix-point combinator (μ-combinator), I rely on a global environment of expressions. Although the two representations are equivalent, the one adopted by this thesis is arguably simpler. The adopted version, compared to the version using local μ-combinators, is easier to convert to and from context-free grammars. Indeed, before a translation to grammars is possible, nested μ-combinators need to be converted to a single top-level fix-point (via Bekić's Lemma (Bekić, 1984)). This representation will be closer to the one adopted in the presented implementations.

2. The context-free expressions of this thesis are *value aware*. They describe not only sets of accepted sequences of tokens (*languages*) but also the *value* associated with each such sequence. This value awareness is typical of works on parser combinators, not context-free expressions. The fact that values can be described by and contained within expressions enables them to be both suitable parsers' descriptions and accurate parser states' representations.

   Theoretical works in parsing theory often treat parsing as a decision problem and relegate value generation to an afterthought worthy only of a quick mention. In practice, however, values are all-important. For instance, it is not sufficient for a compiler to know that its input represents a valid program; it must know what the program is! It is high time that parsing theory reflects this and starts treating parsing as the transformation process it is in practice. The approach in this thesis reflects this.

   In the realm of context-free grammars, *attribute grammars* (Deransart et al., 1988) have been introduced towards a similar goal of enabling value-awareness. The two approaches however differ in several aspects:

   - In their most general form, attribute grammars allow for *inherited attributes*, in which attributes of a node might depend on the attributes of their parent and siblings. For the context-free expressions of this thesis, the value associated with an expression can depend only on the values of its children. In this aspect, context-free expressions are most similar to S-attributed grammars that allow for only *synthesized attributes*, attributes that might depend only on the attributes of the child nodes.
   - For attribute grammars, attributes and their computations have to be explicitly specified: Each rule is to be decorated with a series of assignments that specify the values of the various attributes. In context-free expressions, the value of an expression is determined uniquely by its constructor and does not need to be explicitly stated. For instance, the value of a *sequence* expression $e_1 \cdot e_2$ is set to be the pair of the values of the children expressions e_1 and e_2. To allow for finer control, a *map* constructor is introduced so that arbitrary functions can be applied on values.

3. The formalism presented in this thesis makes a distinction between tokens and token kinds. Tokens represent actual values, with potential payload data, whereas token kinds are an abstraction of tokens that only preserves aspects relevant for recognition. This

distinction, though seldom made, is important in the light of value awareness. Tokens and kinds are there to reconcile the fact that we want to have infinitely many different tokens with the possibility to explicitly list ways to start valid sequences. For the latter task, I will resort to token kinds.

Expressions as Parsing States. I argue that context-free expressions form a good formalism not only to describe parsers but also to model the runtime state of parsers. Typically, traditional parsing algorithms need to convert the grammar description into some ad hoc collection of data structures. In this thesis, I will show that context-free expressions can act as both a language specification tool and as a representation of parser states. In Chapters 4 and 5, I will develop parsing algorithms, based around the idea of derivatives (Brzozowski, 1964; Might et al., 2011), whose states will be encoded as context-free expressions.

Context-free expressions, as defined in this chapter, will prove, due to their value-awareness, to be a suitable formalism for describing pretty printers. I will discuss this in more detail when discussing SCALL1ON, the parsing and pretty printing library, in Chapter 6.

3.1 Preliminaries

Recognition is the task of determining if a given sequence of tokens is part of a language. Parsing, on the other hand, not only encompasses recognition, but also requires a *value* to be produced in case of a successful parse. Contrary to formalisms such as context-free grammars, the formalism that I present in this chapter explicitly incorporates such values. Preliminary to the actual description of context-free expressions, I will introduce the notion of values and their properties, as well as the concept of tokens and token kinds.

3.1.1 Values and Types

Typically, the value produced by a parser will be of the form of a *parse tree*, or some other recursive data type. For the purpose of the current formalism, the type of values is not constrained to be some flavour of parse trees, and is instead left completely abstract. I will denote by $\mathbb{V}$ this set of values. The only constraint on $\mathbb{V}$ is that the set must be closed under cartesian product. In other words, for any two values $v_1, v_2 \in \mathbb{V}$, the pair of the two values, denoted by (v_1, v_2), should also be a value.

The formalism developed in this chapter is *typed*. I will denote by $\mathbb{T}$ the set of types. For a value $v \in \mathbb{V}$ and a type $T \in \mathbb{T}$, I will denote by $v : T$ the fact that the value v has type T. As for values, I will assume that types are cartesian closed. I will denote by $(T_1, T_2) \in \mathbb{T}$ the pair of types T_1 and T_2. I will assume $(v_1, v_2) : (T_1, T_2)$ if and only if $v_1 : T_1$ and $v_2 : T_2$.

The formalism also assumes the existence of functions from and to values. The set $T_1 \to T_2$ denotes total functions from values of type T_1 to values of type T_2. Functions may themselves be values, but the formalism makes no such assumption.

Finally, the notion of sequences is also of great importance to the formalism. As in Chapter 2, the notation $\langle\rangle$ represents the empty sequence, while $xs_1 +\!\!+\ xs_2$ represents concatenation of xs_1 and xs_2, and $x :: xs$ represents the prepending of x to xs.

3.1.2 Tokens and Token Kinds

The input to a parser is a sequence of *tokens*, generally produced by an independent lexer. For the purposes of this formalism, tokens are considered to be actual values. I will use $\texttt{Token} \in \mathbb{T}$ to denote the type of tokens. The values $v \in \mathbb{V}$ such that $v : \texttt{Token}$ are called *tokens*. I will use the lower case letter t to denote such tokens and variations of the name ts to represent sequences of tokens.

In practice tokens will often form an infinite set. Indeed, tokens often carry around data that ranges over an infinite domain of values. For example, an identifier token may contain the name of the identifier, while a number token may contain the actual number that was written. In addition, tokens may also contain meta data, such as the position of the token in the source file. Since this data may be used to build the actual parsed value, it is of the uttermost importance that it is preserved. However, this data might be completely irrelevant for the purposes of recognition.

I will use the concept of *token kinds* to abstract away details in tokens that are irrelevant for recognition.[1] Token kinds represent (potentially infinite) groups of tokens. I will denote by $\mathbb{K}$ the finite set of all kinds. Furthermore, I will assume a function $\texttt{kind}(\cdot)$ from tokens to token kinds. For any token t, the unique kind $k = \texttt{kind}(t)$ is called the kind of t. In chapter 4, kinds will prove also useful in the context of LL(1) checking, providing a convenient way to list sets of accepted tokens and check for conflicts.

As an example of tokens and kinds, the strings `"hello world"`, `"foo"` and `"bar"` could be considered tokens, while `string` would be their token kind. In the proposed formalism, a parser would only be able to state that it expects a token of kind `string` as the next token, and not a specific token such as `"too specific"`. The actual content of the token would not factor in whether or not it is accepted, only its kind. In the case of keywords tokens, such as `if`, `then` or `else`, the associated kind generally depends on the actual keyword, such that an `if` token can not be accepted in place of a `else` token for instance.

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 described by a context-free expression may depend on the actual tokens and their payload.

For the remainder of this thesis, I will assume a fixed set of values, types, tokens and token

[1]Aho et al. (2006) use *tokens* in place of the token kinds of this thesis, and use *lexemes* to refer to actual tokens.

kinds. As a simplifying assumption, I will assume each kind $k \in \mathbb{K}$ to have at least one token t such that `kind`$(t) = k$.

3.2 Context-Free Expressions

The main abstraction offered by the formalism is that of *context-free expressions*. Context-free expressions are an expression-based formalism for describing mappings between token sequences and values. Context-free expressions, as I define them in this thesis, can be seen as an extension of regular expressions (see Chapter 2) along two independent axes: context-freeness and value-awareness.

3.2.1 Expressions

For every type $T \in \mathbb{T}$, let $\mathbf{CFE}_T$ denote the set of context-free expressions that associates token sequences with values of type T. Those sets are inductively defined by the rules in Figure 3.1.

$$\frac{k \in \mathbb{K}}{elem_k \in \mathbf{CFE}_{\texttt{Token}}} \qquad \frac{T \in \mathbb{T}}{\bot \in \mathbf{CFE}_T} \qquad \frac{v : T}{\varepsilon_v \in \mathbf{CFE}_T}$$

$$\frac{e_1 \in \mathbf{CFE}_T \qquad e_2 \in \mathbf{CFE}_T}{e_1 \vee e_2 \in \mathbf{CFE}_T} \qquad \frac{e_1 \in \mathbf{CFE}_{T_1} \qquad e_2 \in \mathbf{CFE}_{T_2}}{e_1 \cdot e_2 \in \mathbf{CFE}_{(T_1, T_2)}}$$

$$\frac{e \in \mathbf{CFE}_{T_1} \qquad f \in T_1 \to T_2}{f \circledcirc e \in \mathbf{CFE}_{T_2}} \qquad \frac{x \in \mathbf{Id}_T}{var_x \in \mathbf{CFE}_T}$$

Figure 3.1 – Definition of context-free expressions.

The construct $elem_k$, $\bot$, and ε_v form the basic context-free expressions. Intuitively, $elem_k$ represents a single token of kind k, $\bot$ represents failure, and ε_v represents the empty string. The value v attached to ε_v represents the value associated to the empty string by the expression. This value is important because context-free expressions are meant not only to describe a language (a set of sequences of tokens), but also the value associated with each recognised sequence.

The construct $e_1 \vee e_2$ represents a disjunction, that is non-deterministic choice between the children expressions e_1 and e_2. Next, the construct $e_1 \cdot e_2$ represents the sequence of e_1 and e_2, in that specific order. The construct $f \circledcirc e$ (the *map* combinator) represents the application of the function f on values produced by e, and so without altering what is recognised. Finally, the construct var_x represents a reference to an expression defined in an environment. The variables and the environment enable mutually recursive expressions.

3.2.2 Environments

Environments are mapping from *identifiers* to context-free expressions of heterogenous types. Identifiers determine the type of context-free expressions that environments may associate to them.

I will assume, for every type T, an infinite set of distinct identifiers denoted by $\mathbf{Id}_T$. I assume that all sets $\mathbf{Id}_T$ are disjoint. I denote the disjoint union of all such sets by $\mathbf{Id} = \bigsqcup_{T \in \mathbb{T}} \mathbf{Id}_T$. By extension, I will call T the type of an identifier $x \in \mathbf{Id}_T$.

An environment Γ is a finite mapping that associates, for various types T, identifiers $x \in \mathbf{Id}_T$ with a unique context-free expression $e \in \mathbf{CFE}_T$ of matching type. I use $\Gamma(x)$ to denote the expression associated with x by Γ.

3.2.3 Semantics

Context-free expressions associate token sequences with values. The inductive predicate $e \vdash_\Gamma ts \rightsquigarrow v$ indicates that, in a given environment Γ, the expression e associates the token sequence ts with the value v. The inductive predicate is defined by the rules in Figure 3.2. Note that this remarkably simple definition gives a declarative specification of value-aware parsing.

$$\textsc{MElem}\,\frac{k = \texttt{kind}(t)}{elem_k \vdash_\Gamma \langle t \rangle \rightsquigarrow t} \qquad \textsc{MEps}\,\frac{}{\varepsilon_v \vdash_\Gamma \langle\rangle \rightsquigarrow v}$$

$$\textsc{MDisL}\,\frac{e_1 \vdash_\Gamma ts \rightsquigarrow v}{e_1 \vee e_2 \vdash_\Gamma ts \rightsquigarrow v} \qquad \textsc{MDisR}\,\frac{e_2 \vdash_\Gamma ts \rightsquigarrow v}{e_1 \vee e_2 \vdash_\Gamma ts \rightsquigarrow v}$$

$$\textsc{MSeq}\,\frac{e_1 \vdash_\Gamma ts_1 \rightsquigarrow v_1 \qquad e_2 \vdash_\Gamma ts_2 \rightsquigarrow v_2}{e_1 \cdot e_2 \vdash_\Gamma ts_1 \mathbin{+\!+} ts_2 \rightsquigarrow (v_1, v_2)}$$

$$\textsc{MMap}\,\frac{e \vdash_\Gamma ts \rightsquigarrow v}{f \circledcirc e \vdash_\Gamma ts \rightsquigarrow f(v)} \qquad \textsc{MVar}\,\frac{e = \Gamma(x) \qquad e \vdash_\Gamma ts \rightsquigarrow v}{var_x \vdash_\Gamma ts \rightsquigarrow v}$$

Figure 3.2 – Semantics of context-free expressions.

The semantic relation formalises the intuitive meaning of the context-free expressions constructs:

MELEM	The construct $elem_k$ assigns to the token sequences containing a single token t of kind k the value t, that is the token itself is the value.
MEPS	The expression ε_v assigns to the empty sequence of tokens the value v.
MDISL and MDISR	The expression $e_1 \vee e_2$ assigns to any sequence of tokens ts the value v if either e_1 or e_2 assigns the value v to ts.
MSEQ	The expression $e_1 \cdot e_2$ assigns to any sequences of tokens ts the pair of values (v_1, v_2) if ts can be decomposed as two sequences ts_1 and ts_2 such that $ts = ts_1 +\!+\ ts_2$ and e_1 assigns the value v_1 to ts_1 and e_2 assigns the value v_2 to ts_2.
MMAP	The construct $f \circledcirc e$ assigns the value $f(v)$ to ts provided that e assigns v to ts.
MVAR	Lastly, the construct var_x assigns to ts the value v, given that the environment Γ contains an entry $e = \Gamma(x)$ for the identifier x and that entry e assigns to the sequence of tokens ts the value v.

As the semantics are defined, for any sequence of tokens ts, the values v assigned to ts by a context-free expression in $\mathbf{CFE}_T$ have the type T.

Theorem 3.1 (Type correctness)**.** *For any environment Γ, type $T \in \mathbb{T}$, expression $e \in \mathbf{CFE}_T$, token sequence ts and value $v \in \mathbb{V}$, if $e \vdash_\Gamma ts \rightsquigarrow v$ then $v : T$.*

Proof. By induction on the derivation of $e \vdash ts \rightsquigarrow v$.

1. Consider the MELEM case. In that case, the expression e is bound to be $elem_k \in \mathbf{CFE}_{\texttt{Token}}$ for some kind k, and thus the type T has to be equal to `Token`. In addition, the value v is bound to be a token t. Therefore, it holds that $v : T$.

2. Consider the MEPS case. In that case, the expression e is bound to be $\varepsilon_v \in \mathbf{CFE}_T$. Since the value produced is v, it is the case that $v : T$.

3. The MDISL and MDISR trivially hold by induction hypothesis.

4. Consider MSEQ case. In that case, the expression e is bound to be $e_1 \cdot e_2 \in \mathbf{CFE}_{(T_1, T_2)}$ for some expressions $e_1 \in \mathbf{CFE}_{T_1}$ and $e_2 \in \mathbf{CFE}_{T_2}$. The value v is bound to be a pair (v_1, v_2), where v_1 is a value produced by e_1 and v_2 by e_2. By induction hypothesis, it is the case that $v_1 : T_1$ and $v_2 : T_2$. Therefore, it must be the case that $(v_1, v_2) : (T_1, T_2)$, and thus also that $v : T$.

5. Consider MMAP case. In that case, the expression e is bound to be $f \circledcirc e_1 \in \mathbf{CFE}_{T_2}$ for some expressions $e_1 \in \mathbf{CFE}_{T_1}$ and function $f \in T_1 \to T_2$. The value v is bound to be the application of f to v_1, where v_1 is a value produced by e_1. By induction hypothesis, it is the case $v_1 : T_1$. Therefore the application of f to v_1 results in a value $f(v_1)$ which by definition has type T_2. Therefore, $v : T$.

6. Finally, consider the MVAR case. In that case, the expression e is bound to be $var_x \in \mathbf{CFE}_T$ for some identifier $x \in \mathbf{Id}_T$. Thus, it must be the case that $\Gamma(x) \in \mathbf{CFE}_T$. By induction hypothesis, the value v produced by $\Gamma(x)$ has type T. Since the same value v is produced by var_x, it must be the case that $v : T$.

□

3.2.4 Language of a Context-free Expression

While reasoning about context-free expressions, it is sometimes useful to abstract away values and focus on recognised sequences only. The set of recognised sequences of an expression is called its *language*. Given an environment Γ, the language of a context-free expression e is the set of token sequences ts such that there exists a value v with $e \vdash_\Gamma ts \rightsquigarrow v$.

$$L_\Gamma(e) := \{\, ts \mid \exists v.\ e \vdash_\Gamma ts \rightsquigarrow v \,\}$$

3.3 Correspondence with Context-free Grammars

Context-free expressions, as I have defined them, are straightforward to convert to and from context-free grammars. This correspondence is useful as it allows us to transfer many theorems on the expressiveness of context-free expressions by reduction to grammars.

3.3.1 From Grammars to Expressions

Consider a context-free grammar $G = (V, \Sigma, R, S)$, where V is a finite set of non-terminals symbols, Σ is a finite set of terminal symbols, $R \subseteq V \times (V \cup \Sigma)^*$ is a set of rewrite rules, and finally S is the start non-terminal symbol. The rewrite rules of R are pairs assigning a non-terminal to a sequence of terminals and non-terminals symbols.

In the proposed translation to context-free expressions, the set of terminal symbols Σ act as the set of token kinds. The values are tokens and parse trees. Parse trees contain tokens at the leaves. At inner nodes, parse trees indicate which rule has been applied. Parse tree are defined as follows:

$$\frac{t \in \mathbb{V}}{\textsc{TreeLeaf}(t) \in \mathbb{V}} \qquad \frac{r = (s, s_1 \ldots s_n) \in R \quad v_1 \in \mathbb{V} \quad \ldots \quad v_n \in \mathbb{V}}{\textsc{TreeNode}_r(v_1 \ldots v_n) \in \mathbb{V}}$$

Figure 3.3 – Definition of parse trees. Tokens are considered values, as well as parse trees. Each node of a parse tree is tagged with the corresponding rule and has one child for each symbol appearing in the right-hand side of the rule.

$$\frac{t : \texttt{Token}}{\textsc{TreeLeaf}(t) : \textsc{Tree}} \qquad \frac{r = (s, s_1 \ldots s_n) \in R \quad v_1 : \textsc{Tree} \quad \ldots \quad v_n : \textsc{Tree}}{\textsc{TreeNode}_r(v_1 \ldots v_n) : \textsc{Tree}}$$

Figure 3.4 – Definition of the type TREE of parse trees.

For each non-terminal symbol $s \in V$, a corresponding identifier s of type TREE is created. The environment Γ is set to assign to every such identifier s an expression of type TREE that is the n-ary disjunction of all expressions e_r for all $r \in R$ where the left-hand side of r is s. The expressions e_r are themselves defined to be n-ary sequences of var_s (in case of a non-terminal s) and $\textsc{TreeLeaf}(\cdot) \circledcirc elem_k$ (in case of a terminal k), which are then enclosed in a *map* combinator that wraps the result in a $\textsc{TreeNode}_r$. Finally, the top level expression simply is a var_S node corresponding to the start non-terminal S.

3.3.2 From Expressions to Grammars

The translation from expressions to grammars is equally simple. Each node e of the expression and of each expression in the environment Γ, is assigned a fresh non-terminal symbol S_e. The collection of those non-terminals forms V. The set of kinds acts as the set of terminal symbols Σ.

Each node e contributes to the rewrite rules R. In each of the following cases, the left-hand side of added rules always correspond to the non-terminal of the considered node:

- For $\perp$ nodes, no rules are added.
- For ε_v nodes, a rule mapping to the empty sequence is added.

$$S_{\varepsilon_v} \mapsto \langle\rangle$$

- For $elem_k$ nodes, a rule mapping to the terminal k is added.

$$S_{elem_k} \mapsto \langle k \rangle$$

- For $e_1 \vee e_2$ nodes, two rules are added. One mapping to the non-terminal of e_1, and one to the non-terminal of e_2.

$$S_{e_1 \vee e_2} \mapsto \langle S_{e_1} \rangle$$
$$S_{e_1 \vee e_2} \mapsto \langle S_{e_2} \rangle$$

- For $e_1 \cdot e_2$ nodes, a single rule is added. That rule maps to the sequence of the two terminals corresponding to e_1 and e_2.

$$S_{e_1 \cdot e_2} \mapsto \langle S_{e_1}, S_{e_2} \rangle$$

- For $f \circledcirc e_1$ nodes, the rule mapping to the non-terminal corresponding to e_1 is added.

$$S_{f \circledcirc e_1} \mapsto \langle S_{e_1} \rangle$$

- For var_x nodes, the rule mapping to the non-terminal corresponding to $\Gamma(x)$ is added.

$$S_{var_x} \mapsto \langle S_{\Gamma(x)} \rangle$$

Finally, the non-terminal corresponding to the top-level expression e is assigned to be the start symbol S of the grammar.

$$S = S_e$$

3.4 Canonical Representation of Expressions

Context-free expressions are meant to be not only mathematical descriptions of parsers, but also tangible data structures representing parser states. As representations of parser states, it is important to specify how context-free expressions are to be represented in memory. The canonical way to represent context-free expressions is as directed binary trees. The expression is accessed through its top-most node, and can be traversed by following references to child nodes. In this setting, an environment is simply a collection of such trees. Variable expressions contain a key, an index, that refers to some entry in that collection.

Note that, in the context of this work, such a data-structure is meant to be immutable. Operations that would modify the expression instead create a new version of the expression tree. Thanks to immutability, subtrees can be freely shared by many expressions. Operations need not resort to copying and can freely point to already existing subtrees. This also opens the way to persistence, in which multiple versions of a data structure simultaneously exist.

Although this representation is the canonical way of representing expressions, it is not the only one. As I shall explain in Chapter 5, a different representation based on *zippers* (Huet, 1997) will prove to be especially adapted for parsing algorithms based on derivatives.

3.5 Example

As a simple example of the theoretical framework, consider the mapping $L = \{(\mathtt{a}^n\mathtt{b}^n, n) \mid n \in \mathbb{N}\}$, which assigns to sequences of n `a`'s followed by n `b`'s the integer value n. In this example, the tokens are `a` and `b`, while their respective kinds are A and B. Towards an expression describing this mapping, consider the singleton environment Γ that maps the identifier x to the expression:

$$f \circledcirc ((elem_A \cdot var_x) \cdot elem_B) \vee \varepsilon_0$$
$$\text{where } f(((t_1, n), t_2)) = n + 1$$

Intuitively, the expression $\Gamma(x)$ describes 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 token, with value 0.

See Figure 3.5 for the canonical representation of the expression $\Gamma(x)$ as a tree.

$$\begin{tikzcd}
 & & & \vee \arrow[dl] \arrow[dr] & \\
 & & f\circledcirc \arrow[d] & & \varepsilon_0 \\
 & & \cdot \arrow[dl] \arrow[dr] & & \\
 & \cdot \arrow[dl] \arrow[dr] & & elem_B & \\
elem_A & & var_x & &
\end{tikzcd}$$

Figure 3.5 – Canonical representation of the expression $\Gamma(x)$.

In this environment, the mapping L is simply described by the expression var_x. The following statements about the semantics of the expression var_x are all derivable:

$$var_x \vdash_\Gamma \langle\rangle \rightsquigarrow 0$$
$$var_x \vdash_\Gamma \langle \mathtt{a}, \mathtt{b} \rangle \rightsquigarrow 1$$
$$var_x \vdash_\Gamma \langle \mathtt{a}, \mathtt{a}, \mathtt{b}, \mathtt{b} \rangle \rightsquigarrow 2$$

Those three statements are witnessed by the following derivations D_0, D_1 and D_2:

$$D_0 = \textsc{MVar}\,\dfrac{\textsc{MDisR}\,\dfrac{\textsc{MEps}\,\dfrac{}{\varepsilon_0 \vdash_\Gamma \langle\rangle \rightsquigarrow 0}}{f \circledcirc ((elem_A \cdot var_x) \cdot elem_B) \vee \varepsilon_0 \vdash_\Gamma \langle\rangle \rightsquigarrow 0}}{var_x \vdash_\Gamma \langle\rangle \rightsquigarrow 0}$$

$$D_1 = \textsc{MVar}\,\dfrac{\textsc{MDisL}\,\dfrac{\textsc{MMap}\,\dfrac{\textsc{MSeq}\,\dfrac{\textsc{MSeq}\,\dfrac{\textsc{MElem}\,\dfrac{}{\mathit{elem}_A \vdash_\Gamma \langle \mathtt{a} \rangle \rightsquigarrow \mathtt{a}} \quad D_0}{\mathit{elem}_A \cdot \mathit{var}_x \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b} \rangle \rightsquigarrow (\mathtt{a},0)} \quad \textsc{MElem}\,\dfrac{}{\mathit{elem}_B \vdash_\Gamma \langle \mathtt{b} \rangle \rightsquigarrow \mathtt{b}}}{(\mathit{elem}_A \cdot \mathit{var}_x) \cdot \mathit{elem}_B \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b},\mathtt{b} \rangle \rightsquigarrow ((\mathtt{a},0),\mathtt{b})}}{f \circledcirc ((\mathit{elem}_A \cdot \mathit{var}_x) \cdot \mathit{elem}_B) \vdash_\Gamma \langle \mathtt{a},\mathtt{b} \rangle \rightsquigarrow 1}}{f \circledcirc ((\mathit{elem}_A \cdot \mathit{var}_x) \cdot \mathit{elem}_B) \vee \varepsilon_0 \vdash_\Gamma \langle \mathtt{a},\mathtt{b} \rangle \rightsquigarrow 1}}{\mathit{var}_x \vdash_\Gamma \langle \mathtt{a},\mathtt{b} \rangle \rightsquigarrow 1}$$

$$D_2 = \textsc{MVar}\,\dfrac{\textsc{MDisL}\,\dfrac{\textsc{MMap}\,\dfrac{\textsc{MSeq}\,\dfrac{\textsc{MSeq}\,\dfrac{\textsc{MElem}\,\dfrac{}{\mathit{elem}_A \vdash_\Gamma \langle \mathtt{a} \rangle \rightsquigarrow \mathtt{a}} \quad D_1}{\mathit{elem}_A \cdot \mathit{var}_x \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b} \rangle \rightsquigarrow (\mathtt{a},1)} \quad \textsc{MElem}\,\dfrac{}{\mathit{elem}_B \vdash_\Gamma \langle \mathtt{b} \rangle \rightsquigarrow \mathtt{b}}}{(\mathit{elem}_A \cdot \mathit{var}_x) \cdot \mathit{elem}_B \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b},\mathtt{b} \rangle \rightsquigarrow ((\mathtt{a},1),\mathtt{b})}}{f \circledcirc ((\mathit{elem}_A \cdot \mathit{var}_x) \cdot \mathit{elem}_B) \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b},\mathtt{b} \rangle \rightsquigarrow 2}}{f \circledcirc ((\mathit{elem}_A \cdot \mathit{var}_x) \cdot \mathit{elem}_B) \vee \varepsilon_0 \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b},\mathtt{b} \rangle \rightsquigarrow 2}}{\mathit{var}_x \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b},\mathtt{b} \rangle \rightsquigarrow 2}$$

On the other hand, the following statements are not derivable:

$$\begin{array}{ll} \mathit{var}_x \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b},\mathtt{b} \rangle \rightsquigarrow 17 & \mathit{var}_x \vdash_\Gamma \langle \mathtt{a},\mathtt{b},\mathtt{a},\mathtt{b} \rangle \rightsquigarrow 2 \\ \mathit{var}_x \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b} \rangle \rightsquigarrow 2 & \mathit{var}_x \vdash_\Gamma \langle \mathtt{a},\mathtt{a},\mathtt{b},\mathtt{b},\mathtt{b} \rangle \rightsquigarrow 2 \end{array}$$

3.6 Comparing Expressions

In this section, I will introduce several relations between context-free expressions. Firstly, I will introduce a strict equivalence binary relation that will denote that two environment and context-free expression pairs have exactly the same meaning. This relation will prove an important tool for parser designers as they reason about their parsers, as well as for parsing algorithm designers as they reason about transformations of parser states encoded as expressions. Secondly, I will introduce a weaker notion of equivalence, named *prefix equivalence*. Two environment and expression pairs are prefix-equivalent if and only if they behave the same on all token sequences starting with a given prefix. Finally, I will also introduce the *derivative* binary relation. This relation, as well as prefix equivalence, will prove useful to reason about successive states of parsers.

3.6.1 Equivalence

Two context-free expressions e_1 and e_2 with respective environments Γ_1 and Γ_2 are considered *equivalent* (denoted by $\Gamma_1, e_1 \equiv \Gamma_2, e_2$) if and only if they associate the same token sequences to the same values.

$$\begin{gathered}\Gamma_1, e_1 \equiv \Gamma_2, e_2 \\ \Longleftrightarrow \\ \forall ts, v. \quad (e_1 \vdash_{\Gamma_1} ts \rightsquigarrow v \iff e_2 \vdash_{\Gamma_2} ts \rightsquigarrow v)\end{gathered}$$

When two expressions e_1 and e_2 are equivalent in a common environment Γ, I will simply note $e_1 \equiv_\Gamma e_2$ to mean $\Gamma, e_1 \equiv \Gamma, e_2$.

$$\begin{gathered}e_1 \equiv_\Gamma e_2 \\ \Longleftrightarrow \\ \forall ts, v. \quad (e_1 \vdash_{\Gamma} ts \rightsquigarrow v \iff e_2 \vdash_{\Gamma} ts \rightsquigarrow v)\end{gathered}$$

This strict notion of equivalence offers a way for parser designers to reason about their code. Unfortunately, checking equivalence between arbitrary context-free expressions is undecidable. Indeed, the problem to context-free grammar equivalence, which is undecidable (Sipser, 2012, Chapter 5), can be reduced to context-free expressions equivalence. Although checking equivalence is undecidable in the general case, there exists many useful transformations which preserve equivalence. Such transformations are a useful tool for safely working with parser code.

3.6.2 Equivalence-Preserving Transformations

Programmers are often faced with the task of refactoring code, either initially while writing the code, or even some time afterwards while revisiting the code. Refactoring without introducing bugs or otherwise changing the intended behaviour is often a difficult task (Bavota et al., 2012). Fortunately, as I will show in this section, parser designers working with context-free expressions have at their disposition a plethora of useful equivalence-preserving transformations. When applying such transformations, programmers are ensured that the functionality of their parser remains unchanged.

In addition, the various equivalence-preservations transformations show that context-free expressions are instances of many interesting algebraic structures. Such structures are well-known in some communities, notably functional programming communities. Members of such communities can transpose their intuitions and knowledge about such abstract structures to context-free expressions. These laws generalise the properties of formal languages to value-aware context-free expressions.

Table 3.2 presents an overview of such equivalence-preserving transformations.

$\bot \cdot e$	$\equiv_\Gamma$	$\bot$	Left-zero of sequencing
$e \cdot \bot$	$\equiv_\Gamma$	$\bot$	Right-zero of sequencing
$\bot \vee e$	$\equiv_\Gamma$	e	Left-unit of disjunction
$e \vee \bot$	$\equiv_\Gamma$	e	Right-unit of disjunction
$e_1 \vee e_2$	$\equiv_\Gamma$	$e_2 \vee e_1$	Commutativity of disjunction
$e_1 \vee (e_2 \vee e_3)$	$\equiv_\Gamma$	$(e_1 \vee e_2) \vee e_3$	Associativity of disjunction
$(e \cdot e_1) \vee (e \cdot e_2)$	$\equiv_\Gamma$	$e \cdot (e_1 \vee e_2)$	Left-factoring
$(e_1 \cdot e) \vee (e_2 \cdot e)$	$\equiv_\Gamma$	$(e_1 \vee e_2) \cdot e$	Right-factoring
$f \circledcirc \bot$	$\equiv_\Gamma$	$\bot$	Zero of map
$id \circledcirc e$	$\equiv_\Gamma$	e	Identity of map
$f_1 \circledcirc (f_2 \circledcirc e)$	$\equiv_\Gamma$	$(f_1 \circ f_2) \circledcirc e$	Composition of map
$f \circledcirc \varepsilon_v$	$\equiv_\Gamma$	$\varepsilon_{f(v)}$	Homomorphism of map
$f \circledcirc (e_1 \vee e_2)$	$\equiv_\Gamma$	$(f \circledcirc e_1) \vee (f \circledcirc e_2)$	Distributivity of map over disjunctions
$(f_1 \mathbin{**} f_2) \circledcirc (e_1 \cdot e_2)$	$\equiv_\Gamma$	$(f_1 \circledcirc e_1) \cdot (f_2 \circledcirc e_2)$	Distributivity of map over sequences

Table 3.2 – Equivalence-preserving transformations.

Where id is the identity function and where the binary operation $\mathbin{**}$ on functions is defined as $(f_1 \mathbin{**} f_2)((v_1, v_2)) := (f_1(v_1), f_2(v_2))$.

Note that associativity does not always hold for sequences. Although the languages of sequences $e_1 \cdot (e_2 \cdot e_3)$ and $(e_1 \cdot e_2) \cdot e_3$ are the same, the actual values associated with each word may differ. The reason for this is that values of the form $(v_1, (v_2, v_3))$ may be distinct from values of the form $((v_1, v_2), v_3)$. Table 3.3 presents further equivalence-preserving relations given a way to pair up values that obey associativity.

$\varepsilon_{\mathbb{1}} \otimes e$	$\equiv_\Gamma$	e	Left-unit of monoid sequencing
$e \otimes \varepsilon_{\mathbb{1}}$	$\equiv_\Gamma$	e	Right-unit of monoid sequencing
$e_1 \otimes (e_2 \otimes e_3)$	$\equiv_\Gamma$	$(e_1 \otimes e_2) \otimes e_3$	Associativity of monoid sequencing

Table 3.3 – Equivalence-preserving transformations involving monoids.

Assuming a monoid $(T, \times, \mathbb{1})$, the binary combinator $\otimes$ is defined as $e_1 \otimes e_2 := f_\times \circledcirc (e_1 \cdot e_2)$, with $f_\times((v_1, v_2)) := v_1 \times v_2$.

Correspondence to Functional Programming Type Classes

In the area of functional programming, some algebraic structures have emerged as useful and general design patterns. Such algebraic structures are often referred to as *type classes* (Wadler and Blott, 1989). Some languages, such as Haskell (Marlow, 2010), have built-in support for type classes. Most functional programming languages incorporate those type classes in their standard library and/or in third-party libraries (Typelevel, 2020; Scalaz, 2020).

Context-free expressions are instances of many widely used type classes such as `Functor`, Monoid, Applicative/Monoidal (McBride and Paterson, 2008), and Alternative (Yorgey, 2009). Users of such type classes could easily become familiar with context-free expressions by the simple fact that they are instances of widely used type classes. Type classes also offer an opportunity of code reuse, as library functions that are expressed on those abstract classes can be applied to context-free expressions.

3.6.3 Prefix-Equivalence

Sometimes, it is also useful to talk about equivalence of two expressions for token sequences starting with a given prefix of tokens. This weaker notion will prove useful when it comes to reasoning about the state of parsing algorithms. Indeed, at the point when the next token is known, parsing algorithms that represent their state as an expression are free to modify them arbitrarily as long as they preserve the semantics for sequences starting with that particular token.

$$\begin{gathered}\Gamma_1, e_1 \equiv^{ts} \Gamma_2, e_2 \\ \Longleftrightarrow \\ (\forall ts', v.\ e_1 \vdash_{\Gamma_1} ts \mathbin{++} ts' \rightsquigarrow v \iff e_2 \vdash_{\Gamma_2} ts \mathbin{++} ts' \rightsquigarrow v)\end{gathered}$$

When both expressions share the same environment Γ, I will simply use $e_1 \equiv^{ts}_{\Gamma} e_2$ to denote the statement $\Gamma, e_1 \equiv^{ts} \Gamma, e_2$.

$$\begin{gathered}e_1 \equiv^{ts}_{\Gamma} e_2 \\ \Longleftrightarrow \\ (\forall ts', v.\ e_1 \vdash_{\Gamma} ts \mathbin{++} ts' \rightsquigarrow v \iff e_2 \vdash_{\Gamma} ts \mathbin{++} ts' \rightsquigarrow v)\end{gathered}$$

When the prefix is empty, prefix equivalence is equal to strict equivalence.

$$\equiv^{\langle\rangle} = \equiv$$

The following theorems show that $\equiv^{ts}$ is indeed an equivalence relation and how it may be further weakened. All theorems follow trivially from definition.

Theorem 3.2 (Reflexivity). *The relation $\equiv^{ts}$ is reflexive.*

$$\Gamma, e \equiv^{ts} \Gamma, e$$

Theorem 3.3 (Symmetry). *The relation $\equiv^{ts}$ is symmetric.*

$$\Gamma_1, e_1 \equiv^{ts} \Gamma_2, e_2 \iff \Gamma_2, e_2 \equiv^{ts} \Gamma_1, e_1$$

Theorem 3.4 (Transitivity)**.** *The relation $\equiv^{ts}$ is transitive.*

$$\begin{aligned}&\Gamma_1, e_1 \equiv^{ts} \Gamma_2, e_2 \implies\\&\Gamma_2, e_2 \equiv^{ts} \Gamma_3, e_3 \implies\\&\Gamma_1, e_1 \equiv^{ts} \Gamma_3, e_3\end{aligned}$$

Theorem 3.5 (Weakening)**.** *The relation $\equiv^{ts}$ can be weakened by enlarging the prefix ts.*

$$\Gamma_1, e_1 \equiv^{ts_1} \Gamma_2, e_2 \implies \Gamma_1, e_1 \equiv^{ts_1 \mathbin{++} ts_2} \Gamma_2, e_2$$

3.6.4 Derivatives

Another useful way of comparing context-free expressions is the notion of derivatives. Intuitively, when an expression e_1 admits an expression e_2 as a derivative by a sequence of tokens ts, the expression e_2 can be seen as the state of e_1 after processing the tokens ts. For environments Γ_1, Γ_2, expressions e_1 and e_2, and a token sequence ts, I will denote by $\Gamma_1, e_1 \gg^{ts} \Gamma_2, e_2$ the fact that the pair Γ_1, e_1 admits the pair Γ_2, e_2 as a derivative by ts.

$$\begin{gathered}\Gamma_1, e_1 \gg^{ts} \Gamma_2, e_2\\\iff\\(\forall ts', v.\ e_1 \vdash_{\Gamma_1} ts \mathbin{++} ts' \rightsquigarrow v \iff e_2 \vdash_{\Gamma_2} ts' \rightsquigarrow v)\end{gathered}$$

As shown by the following theorems, the relation exhibits interesting properties, notably in relation with prefix equivalence. Those properties will come in handy when reasoning about *parsing with derivatives* algorithms in the following chapters. All following theorems are straightforwardly proven by definition.

Theorem 3.6 (Reflexivity)**.** *For any environment Γ, expression e, and token sequence ts, the following holds:*

$$\Gamma, e \gg^{\langle\rangle} \Gamma, e$$

Theorem 3.7 (Transitivity)**.** *For any environments Γ_1, Γ_2, Γ_3, expressions e_1, e_2, e_3, and token sequences ts_1, ts_2, the following holds:*

$$\begin{aligned}&\Gamma_1, e_1 \gg^{ts_1} \Gamma_2, e_2 \implies\\&\Gamma_2, e_2 \gg^{ts_2} \Gamma_3, e_3 \implies\\&\Gamma_1, e_1 \gg^{ts_1 \mathbin{++} ts_2} \Gamma_3, e_3\end{aligned}$$

Theorem 3.8 (Left Unit)**.** *For any environments Γ_1, Γ_2, Γ_3, expressions e_1, e_2, e_3, and token sequence ts, the following holds:*

$$\begin{aligned}
&\Gamma_1, e_1 \equiv^{ts} \Gamma_2, e_2 \implies \\
&\Gamma_2, e_2 \gg^{ts} \Gamma_3, e_3 \implies \\
&\Gamma_1, e_1 \gg^{ts} \Gamma_3, e_3
\end{aligned}$$

Theorem 3.9 (Right Unit)**.** *For any environments* Γ_1, Γ_2, Γ_3, *expressions* e_1, e_2, e_3, *and token sequence* ts, *the following holds:*

$$\begin{aligned}
&\Gamma_1, e_1 \gg^{ts} \Gamma_2, e_2 \implies \\
&\Gamma_2, e_2 \equiv^{\langle\rangle} \Gamma_3, e_3 \implies \\
&\Gamma_1, e_1 \gg^{ts} \Gamma_3, e_3
\end{aligned}$$

3.7 Properties of Context-Free Expressions

In this section, I define several properties of context-free expressions. Basic properties, such as *productivity* (checking if the language of the expression is non-empty) or *nullability* (checking if the language contains the empty sequence) among others, will play an important role for both classifying expressions (for instance as LL(1) in Chapter 4) and guiding parsing algorithms (as shown in Chapters 4 and 5). Fortunately, such properties are decidable, as I will demonstrate.

Each property will be defined using a set of simple and intuitive inductive rules. In each case, I will show how the inductively-defined properties relate to the semantics relation $e \vdash ts \rightsquigarrow v$.

Finally, I will discuss how to efficiently compute such properties. Computing those properties in a top-down way will prove hard because of the mutually recursive nature of the environment. The technique I will discuss instead works in a bottom-up manner, propagating information about properties from children to parents.

3.7.1 Productivity

The first property I define is *productivity.* A context-free expression is said to be *productive* if it associates at least one sequence of tokens with a value. I derive productivity according to the inductive rules presented in Figure 3.6.

Not all expressions are productive. For instance $\bot$, $\bot \cdot elem_k$, $elem_k \cdot \bot$ are all non-productive, and so regardless of the environment. Non-productive expressions can also occur due to non-well-founded recursion, as in the expression var_x with an environment mapping x to $elem_k \cdot var_x$.

Unproductive expressions are trivially all equivalent to $\bot$. In practice, unproductive expressions arising from a non-well-founded recursion are often symptomatic of a design or programming error.

$$\textsc{PEps}\,\frac{}{\textsc{productive}_\Gamma(\varepsilon_v)} \qquad \textsc{PElem}\,\frac{}{\textsc{productive}_\Gamma(elem_k)}$$

$$\textsc{PDisL}\,\frac{\textsc{productive}_\Gamma(e_1)}{\textsc{productive}_\Gamma(e_1 \vee e_2)} \qquad \textsc{PDisR}\,\frac{\textsc{productive}_\Gamma(e_2)}{\textsc{productive}_\Gamma(e_1 \vee e_2)}$$

$$\textsc{PSeq}\,\frac{\textsc{productive}_\Gamma(e_1) \qquad \textsc{productive}_\Gamma(e_2)}{\textsc{productive}_\Gamma(e_1 \cdot e_2)}$$

$$\textsc{PMap}\,\frac{\textsc{productive}_\Gamma(e)}{\textsc{productive}_\Gamma(f \circledcirc e)} \qquad \textsc{PVar}\,\frac{e = \Gamma(x) \qquad \textsc{productive}_\Gamma(e)}{\textsc{productive}_\Gamma(var_x)}$$

Figure 3.6 – Rules for productivity.

Theorem 3.10. *For any environment* Γ *and context-free expression e:*

$$\textsc{productive}_\Gamma(e) \iff \exists ts, v.\ e \vdash_\Gamma ts \rightsquigarrow v$$

Proof. By induction on the derivation of $\textsc{productive}_\Gamma(e)$ for the first direction, and by induction on the derivation of $e \vdash_\Gamma ts \rightsquigarrow v$ for the second direction. □

3.7.2 Nullability

The next property I introduce is *nullability*. Given an environment Γ, an expression *e* is said to be *nullable* if it associates the empty sequence of tokens with at least one value. Rules for nullability are given by Figure 3.7.

$$\textsc{NEps}\,\frac{}{\textsc{nullable}_\Gamma(\varepsilon_v)} \qquad \textsc{NDisL}\,\frac{\textsc{nullable}_\Gamma(e_1)}{\textsc{nullable}_\Gamma(e_1 \vee e_2)} \qquad \textsc{NDisR}\,\frac{\textsc{nullable}_\Gamma(e_2)}{\textsc{nullable}_\Gamma(e_1 \vee e_2)}$$

$$\textsc{NSeq}\,\frac{\textsc{nullable}_\Gamma(e_1) \qquad \textsc{nullable}_\Gamma(e_2)}{\textsc{nullable}_\Gamma(e_1 \cdot e_2)}$$

$$\textsc{NMap}\,\frac{\textsc{nullable}_\Gamma(e)}{\textsc{nullable}_\Gamma(f \circledcirc e)} \qquad \textsc{NVar}\,\frac{e = \Gamma(x) \qquad \textsc{nullable}_\Gamma(e)}{\textsc{nullable}_\Gamma(var_x)}$$

Figure 3.7 – Rules for nullability.

Disjunctions are nullable when either side is nullable. In the case of sequences, both sides have to be nullable for the entire sequence to be nullable.

Theorem 3.11. *For any expression e and value v:*

$$\textsc{nullable}_\Gamma(s) \iff \exists v.\ s \vdash_\Gamma \langle\rangle \rightsquigarrow v$$

Proof. By induction on the derivation of $\textsc{nullable}_\Gamma(e)$ for the first direction, and by induction on the derivation of $e \vdash_\Gamma \langle\rangle \rightsquigarrow v$ for the second direction. □

3.7.3 First Set

Given an environment Γ, the *first set* of an expression e is the set containing the kinds of all tokens at the start of at least one sequence associated with some value by e. Rules for inclusion in the first set are given by Figure 3.8.

$$\textsc{FElem}\ \frac{}{k \in \textsc{first}_\Gamma(elem_k)}$$

$$\textsc{FDisL}\ \frac{k \in \textsc{first}_\Gamma(e_1)}{k \in \textsc{first}_\Gamma(e_1 \vee e_2)} \qquad \textsc{FDisR}\ \frac{k \in \textsc{first}_\Gamma(e_2)}{k \in \textsc{first}_\Gamma(e_1 \vee e_2)}$$

$$\textsc{FSeqL}\ \frac{k \in \textsc{first}_\Gamma(e_1) \qquad \textsc{productive}_\Gamma(e_2)}{k \in \textsc{first}_\Gamma(e_1 \cdot e_2)} \qquad \textsc{FSeqR}\ \frac{\textsc{nullable}_\Gamma(e_1) \qquad k \in \textsc{first}_\Gamma(e_2)}{k \in \textsc{first}_\Gamma(e_1 \cdot e_2)}$$

$$\textsc{FMap}\ \frac{k \in \textsc{first}_\Gamma(e)}{k \in \textsc{first}_\Gamma(f \circledcirc e)} \qquad \textsc{FVar}\ \frac{e = \Gamma(x) \qquad k \in \textsc{first}_\Gamma(e)}{k \in \textsc{first}\,(var_x)}$$

Figure 3.8 – Rules for inclusion in the first set.

The first set of an expression is an important information that parsing algorithms may take advantage of. The parsing algorithms that I will show in Chapters 4 and 5 all make use of this property to guide derivation, which is the process of computing the next parser state.

Importantly, observe that all derivations of $k \in \textsc{first}_\Gamma(e)$ follow a linear, single-threaded, path starting at $k \in \textsc{first}_\Gamma(e)$ and ending at $k \in \textsc{first}_\Gamma(elem_k)$. Indeed, there are no rules that have more than a single instance of $k \in \textsc{first}_\Gamma(\cdot)$ as a premise, and FElem is the only rule without such a premise. Note that, in the general case, due to the multiple ways membership in the first set can be derived for disjunctions and sequences, there might be multiple such derivation paths, even possibly infinitely many. But such paths are always linear and always end with $k \in \textsc{first}_\Gamma(elem_k)$.

Theorem 3.12. *The first set of a an expression e equals the set*

$$\{\, k \in \mathbb{K} \mid \exists t, ts, v.\ \mathit{kind}(t) = k \wedge e \vdash t :: ts \rightsquigarrow v \,\}$$

Proof. The proposition can be expressed as an equivalence:

$$k \in \textsc{first}_{\Gamma}(e) \iff \exists t, ts, v.\ \texttt{kind}(t) = k \wedge e \vdash t :: ts \rightsquigarrow v$$

The first direction is shown by induction on the derivation of $k \in \textsc{first}_{\Gamma}(e)$, while the second direction is shown by induction on the derivation of $e \vdash t :: ts \rightsquigarrow v$.

In both directions, in cases involving the rules FSEQR and FSEQR, Theorem 3.10 and Theorem 3.11 are used to relate the predicates $\textsc{productive}_{\Gamma}(\cdot)$ and $\textsc{nullable}_{\Gamma}(\cdot)$ with the semantics relation. □

3.7.4 Left-Recursivity

I will call entries in the environment *left-recursive* if they can be reentered without consuming any input tokens. Such left-recursive definitions are often problematic for recursive-descent based approaches, as it can lead to infinite recursion. Formally, an identifier x in an environment Γ is left-recursive if $x \in \textsc{visitable}_{\Gamma}(\Gamma(x))$. See Figure 3.9 for rules for *visitability* and *left-recursivity*.

$$\frac{}{x \in \textsc{visitable}_{\Gamma}(var_x)} \qquad \frac{x \in \textsc{visitable}_{\Gamma}(e_1)}{x \in \textsc{visitable}_{\Gamma}(e_1 \vee e_2)} \qquad \frac{x \in \textsc{visitable}_{\Gamma}(e_2)}{x \in \textsc{visitable}_{\Gamma}(e_1 \vee e_2)}$$

$$\frac{x \in \textsc{visitable}_{\Gamma}(e_1)}{x \in \textsc{visitable}_{\Gamma}(e_1 \cdot e_2)} \qquad \frac{\textsc{nullable}_{\Gamma}(e_1) \qquad x \in \textsc{visitable}_{\Gamma}(e_2)}{x \in \textsc{visitable}_{\Gamma}(e_1 \cdot e_2)}$$

$$\frac{x \in \textsc{visitable}_{\Gamma}(e)}{x \in \textsc{visitable}_{\Gamma}(f \circledcirc e)} \qquad \frac{e = \Gamma(y) \qquad x \in \textsc{visitable}_{\Gamma}(e)}{x \in \textsc{visitable}_{\Gamma}(var_y)}$$

$$\frac{e = \Gamma(x) \qquad x \in \textsc{visitable}_{\Gamma}(e)}{\textsc{Left-Recursive}_{\Gamma}(x)}$$

Figure 3.9 – Rules for inclusion in the visitable set and for left-recursivity.

3.8 Computing Properties

In this section, I discuss how to efficiently compute properties such as the ones defined previously in this chapter, and so not only for top-level expressions, but also for all their sub-expressions at the same time. Computing properties for all sub-expressions is useful for two reasons:

1. Some properties may rely on other properties being computed. For instance, in order to compute membership of a kind in a first set of an expression e, nullability of sub-expressions appearing on the left of a sequence within the expression e is needed (rule FSEQR in Figure 3.8). The same applies for productivity of sub-expressions appearing on the right of sequences within e (rule FSEQL in Figure 3.8).
2. Some properties will be used extensively by the parsing algorithms I will present in this thesis. Such algorithms will traverse expressions and will need to query properties such as membership in the first set or nullability on sub-expressions in order to guide the traversals.

On *regular expressions*, this task is straightforward. For such expressions, as properties of an expression only depends on the properties of its children, a simple recursive procedure following the structure of the regular expression suffices to accurately compute any such property. In this context, the recursive calls performed on child expressions are complete and accurate, thus their result can naturally be recorded as the value of the property for that particular subexpression. See Figure 3.11 for how productivity can be computed on regular expressions.

$$\begin{aligned}
\texttt{isProductive}(\bot) &:= \texttt{false}\\
\texttt{isProductive}(\varepsilon) &:= \texttt{true}\\
\texttt{isProductive}(elem_k) &:= \texttt{true}\\
\texttt{isProductive}(e_1 \vee e_2) &:= \texttt{isProductive}(e_1)\ \texttt{||}\ \texttt{isProductive}(e_2)\\
\texttt{isProductive}(e_1 \cdot e_2) &:= \texttt{isProductive}(e_1)\ \&\&\ \texttt{isProductive}(e_2)\\
\texttt{isProductive}(e^*) &:= \texttt{true}
\end{aligned}$$

Figure 3.10 – Computation of productivity on regular expressions using a simple recursive function. Recursive calls accurately compute the property for the sub-expressions on which they are applied. Such results can safely be recorded, for instance by memoisation or directly in the expression data structure.

Unfortunately, things are not so simple for *context-free* expressions. For context-free expressions, due to the existence of variables, recursive procedures strictly following the structure of expressions could end up in a loop. To avoid this issue, one must make sure entries in the environment can not be visited infinitely many times. To do so, one simple solution is remove the entry associated with the visited variables in the argument environment. Upon revisiting a variable, no recursive call could be made and a conservative approximation of the property would have to be returned.

$$
\begin{aligned}
\texttt{isProductive}_\Gamma(\bot) &:= \texttt{false}\\
\texttt{isProductive}_\Gamma(\varepsilon_v) &:= \texttt{true}\\
\texttt{isProductive}_\Gamma(elem_k) &:= \texttt{true}\\
\texttt{isProductive}_\Gamma(f \circledcirc e) &:= \texttt{isProductive}_\Gamma(e)\\
\texttt{isProductive}_\Gamma(e_1 \vee e_2) &:= \texttt{isProductive}_\Gamma(e_1) \texttt{ || } \texttt{isProductive}_\Gamma(e_2)\\
\texttt{isProductive}_\Gamma(e_1 \cdot e_2) &:= \texttt{isProductive}_\Gamma(e_1) \;\&\&\; \texttt{isProductive}_\Gamma(e_2)\\
\texttt{isProductive}_\Gamma(var_x) &:= \begin{cases} \texttt{isProductive}_{\Gamma - x}(\Gamma(x)) & \text{if } x \in \Gamma \\ \texttt{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.11 – Tentative definition of computation of productivity on context-free expressions using a simple recursive function. In case of variables, the identifier is removed from the environment before the recursive call is made.

Interestingly, this simple technique returns correct results. Indeed, for all properties P that I have introduced, the following holds:

$$\forall x \in \Gamma, P_\Gamma(\Gamma(x)) \iff P_{\Gamma - x}(\Gamma(x))$$

In other words, for all properties and identifiers x in the environment, if the property holds for $\Gamma(x)$, then there is a derivation of $P_\Gamma(x)$ that does not need to unfold any corresponding var_x node.

Although this simple recursive procedure would give back a correct result for the top-level expression, it would also mean that, as soon as entries are removed from the environment, the result of recursive calls to sub-expressions could potentially be incomplete.

As a example, consider a call to $\texttt{isProductive}_\Gamma(var_x)$ in the following environment Γ:

$$
\begin{aligned}
\Gamma(x) &:= var_y \vee \varepsilon_v\\
\Gamma(y) &:= var_x
\end{aligned}
$$

In this example, the sub-expression var_y in $\Gamma(x)$ would not be determined to be productive by the suggested `isProductive` procedure, even though it clearly is, as var_x is determined to be productive in the end.

To get an accurate result for sub-expressions, the computation would have to be restarted for entry in the environment. On top of this, the simple recursive procedure showcased exhibits terrible runtime complexity, which can be bounded by:

$$\mathcal{O}(\text{size}(e) \cdot \prod_{x \in \Gamma} \text{size}(\Gamma(x)))$$

Techniques such as memoisation could be used to somewhat address this complexity problem, but the issue of not being able to accurately compute the property for all sub-expressions in a single pass would remain. To fix this issue, I suggest that the computation of properties be performed not top-down, but in a bottom-up fashion, where child expressions inform their parent. The resulting technique is able to compute inductive properties with complexity:

$$\mathcal{O}(\text{size}(e) + \sum_{x \in \Gamma} \text{size}(\Gamma(x)))$$

In addition, the technique I will show is able to compute the property not only for the top-level expression, but also for all its subexpressions in a single pass.

3.8.1 Computing Properties with Propagation Networks

I suggest the use of *propagation networks* (Radul, 2009) as a way to efficiently and globally compute inductive properties of context-free expressions. Propagation networks are directed networks of *cells*. Cells store some kind of information, which they either are initially provided with or receive from other cells in the network. Upon reception of information, cells update their state and propagate *new information* to some other cells in the network.

Propagation networks can be used to compute properties of context-free expressions. The technique operates in three phases:

1. In the first phase, the context-free expression and all expressions in the environment are traversed in a top-down fashion to build the propagation network.
2. In the second phase, information is propagated through the network according to the rules local to each cell. Such rules reflect the inductive rules that define the various properties. In this setting, information flows from child to parent nodes in the expression trees.
3. In the third and final phase, the properties of each expression node is read from the corresponding cell.

As cells can only gain information, information spreads monotonically over the network. Furthermore, since cells only propagate new information, coupled with the fact that cells can only be in a finite number of states, the process is bound to terminate.

Interestingly, the technique supports the online addition of cells. Such an addition would happen when a new expression is created from existing expressions. In this case, properties of the new expression can be computed from a network of cells in which information has already been partially propagated.

3.9 Parsing with Derivatives

In this section, I give a short presentation of the *parsing with derivatives* algorithm originally described by Might et al. (2011). The algorithm uses a variant of the Brzozowski's derivation operation adapted to context-free expressions to iteratively process the sequence of input tokens for left to right. The presentation that I will give here however differs in two significant aspects:

1. First of all, the algorithm is adapted to fit the framework presented in this chapter. Instead of relying on laziness and memoization to build cyclic structures, I will use the explicit notion of variables and environments. This approach is most similar to the *symbolic* approach of Henriksen et al. (2019). An actual implementation may still use such techniques, but the following presentation makes this cyclic structure explicit.
2. Secondly, I will make use of the properties such as nullability and first sets as defined earlier in this chapter to guide the various functions, notably derivation. In the case of derivation, this will avoid creating non-productive derivative subexpressions. In the original approach by Might et al. (2011), a technique called *compaction* is used to prune those expressions after the fact. Such a technique is no longer necessary in this setting.

The presentation does not to aim to produce a practical algorithm, but instead aims to show how the *parsing with derivatives* approach can be transposed to the theoretical framework suggested by this thesis.

3.9.1 Nullifying Expressions

The first operation defined is an operation to *nullify*, or *close*, an expression. This operation will be used by the upcoming derivation function. When applied on a sequence, that derivation function will sometimes need to nullify the left expression. In the context of regular expressions, this consisted in replacing the nullable left expression e by an ε expression, or simply ignoring it. In the current setting, this operation is more involved due to value-awareness.

Given an expression e in an environment Γ, the resulting expression $\nu_\Gamma(e)$ is to be equivalent to the argument expression for the empty sequence of tokens, and equivalent to $\perp$ for all other sequences. The operation is only interesting on nullable expressions, therefore the definition assumes as a precondition that $\textsc{nullable}_\Gamma(e)$, and as such will only be defined for constructs that can be nullable.

$$
\begin{aligned}
\nu_\Gamma(\varepsilon_v) &:= \varepsilon_v \\
\nu_\Gamma(e_1 \vee e_2) &:= \begin{cases} \nu_\Gamma(e_1) & \text{if } \neg\textsc{nullable}_\Gamma(e_2) \\ \nu_\Gamma(e_2) & \text{if } \neg\textsc{nullable}_\Gamma(e_1) \\ \nu_\Gamma(e_1) \vee \nu_\Gamma(e_2) & \text{otherwise} \end{cases} \\
\nu_\Gamma(e_1 \cdot e_2) &:= \nu_\Gamma(e_1) \cdot \nu_\Gamma(e_2) \\
\nu_\Gamma(f \circledcirc e) &:= f \circledcirc \nu_\Gamma(e) \\
\nu_\Gamma(var_x) &:= var_{x_\nu}
\end{aligned}
$$

The operation can be understood as simply trimming away branches of disjunctions that are not nullable, and replacing references to variables in the environment with their nullified counterpart. Note that the operation introduces variables that refer to identifiers that may not be present in Γ. Such entries correspond to the nullified counterparts of the original entries, and need to be introduced. Therefore the resulting expression only makes sense in an environment that includes such entries. The following operation augments an environment with all nullified entries when applicable:

$$N(\Gamma) := \Gamma \cup \{\, x_\nu \mapsto \nu_\Gamma(e) \mid x \mapsto e \in \Gamma, \textsc{nullable}_\Gamma(e) \wedge x \neq x_\nu \,\}$$

The identifiers used to refer to the nullified version of an identifier x is x_ν. I will assume that such identifiers are not introduced anywhere apart from the previously defined $\nu(\cdot)$ operation. Additionally, one may use the simplifying fact that nullifying an expression is an idempotent operation and reflect that in the identifier naming scheme:

$$(x_\nu)_\nu = x_\nu$$

Furthermore, note that in practice elements could be added to the environment in a lazy fashion, only when the entry is actually needed. In the end, some of the entries may not need to be computed at all.

3.9.2 Derivation

Now that the *nullify* operation has been defined, I can present the adapted derivation function $\delta^t_\Gamma(\cdot)$. The following definition requires as a precondition that the kind of t is part of the first set of the derived expression. Derivatives by a token t of an expression that does not contain the kind of t in its first set is bound to be unproductive.

$$\delta^t_\Gamma(elem_k) := \varepsilon_t$$

$$\delta^t_\Gamma(e_1 \vee e_2) := \begin{cases} \delta^t_\Gamma(e_1) & \text{if } \texttt{kind}(t) \not\in \textsc{first}_\Gamma(e_2) \\ \delta^t_\Gamma(e_2) & \text{if } \texttt{kind}(t) \not\in \textsc{first}_\Gamma(e_1) \\ \delta^t_\Gamma(e_1) \vee \delta^t_\Gamma(e_2) & \text{otherwise} \end{cases}$$

$$\delta^t_\Gamma(e_1 \cdot e_2) := \begin{cases} \delta^t_\Gamma(e_1) \cdot e_2 & \text{if } \neg\textsc{nullable}_\Gamma(e_1) \vee \texttt{kind}(t) \not\in \textsc{first}_\Gamma(e_2) \\ \nu_\Gamma(e_1) \cdot \delta^t_\Gamma(e_2) & \text{if } \texttt{kind}(t) \not\in \textsc{first}_\Gamma(e_1) \\ \delta^t_\Gamma(e_1) \cdot e_2 \vee \nu_\Gamma(e_1) \cdot \delta^t_\Gamma(e_2) & \text{otherwise} \end{cases}$$

$$\delta^t_\Gamma(f \circledcirc e) := f \circledcirc \delta^t_\Gamma(e)$$

$$\delta^t_\Gamma(var_{x^{ts}}) := var_{x^{ts :+ t}}$$

Derivatives context-free expressions are paired with derivative environments, which are computed as follows:

$$\Gamma^{\langle\rangle} := \Gamma$$

$$\Gamma^{ts :+ t} := N(\Gamma^{ts}) \cup \{\, x^{ys :+ t} \mapsto \delta^t_{\Gamma^{ts}}(e) \mid x^{ys} \mapsto e \in \Gamma^{ts} \wedge \texttt{kind}(t) \in \textsc{first}_{\Gamma^{ts}}(e) \wedge \exists xs. xs +\!\!+\; ys = ts \,\}$$

The original expression is to be paired with the environment $\Gamma^{\langle\rangle}$, while derivatives of that expression by token sequences ts are to be paired with Γ^{ts}. Note that, in the above scheme, identifiers superscripted by $\langle\rangle$ are considered to refer the original identifier without the superscript.

$$x^{\langle\rangle} := x$$

As before, elements could be added to the environment lazily, as identifiers for such entries are encountered.

The algorithm retains the simplicity and elegance of the original *parsing with derivatives* algorithm by Might et al. (2011).

Complexity and Performance

Adams et al. (2016) showed that the complexity of parsing with derivatives is at worst cubic in the number of input tokens. This result relies on computations being memoised, and so would not directly apply to the algorithm showed here. I shall make no statement on the theoretical complexity of that algorithm.

Unfortunately, despite its simplicity and elegance, the *parsing with derivatives* approach has not seen wide adoption in practice. The performance of the approach has been widely seen as impractical: The original paper by Might et al. (2011) reports a parsing time of 3 minutes for 31 lines of Python code, which can be reduced to 2 seconds by applying a *compaction* optimisation. Over time, further optimisations have been suggested (Adams et al., 2016). Although performance has reportedly been getting better to the point of being a relatively small factor away from other general context-free parsing techniques, the approach also arguably lost in elegance.

In this thesis, I will show how one can adapt the *parsing with derivatives* technique to the class of LL(1) context-free expressions (Chapter 4). I will show that *parsing with derivatives* algorithms can exhibit quadratic behaviour on this class of expressions, even though traditional parsing algorithms exhibit worst-case linear running time. I will show how one can use a zipper to reduce the theoretical complexity down to linear and ameliorate the performance in practice (Chapter 5). I will also show how one can port the zipper technique from LL(1) expressions to general context-free expressions (Chapter 7). This last technique is reminiscent of an other zipper-based approach by Darragh and Adams (2020) that has been developed concurrently and independently.

4 LL(1) Context-Free Expressions

In this chapter, I will present LL(1) context-free expressions that are a class of context-free expressions where all ambiguities can be resolved using only a single lookahead token. I will present a novel formal characterisation of this class and present a procedure to efficiently and statically check the LL(1) property.

In the second part of this chapter, I will discuss how the derivation computation can be specialised to LL(1) expressions. From this observation, I will then present a top-down parsing algorithm for LL(1) context-free expressions. Through an example, I will show that the complexity of this algorithm is unfortunately (at least) a worst-case quadratic. However, this algorithm proves to be an excellent stepping stone towards an efficient algorithm. In Chapter 5, I will show that, by simply employing a zipper data-structure to represent the LL(1) context-free expressions, the simple parsing algorithm presented in this chapter becomes worst-case linear time.

Throughout this chapter and the next, I will prove lemmas and theorems about the correctness of the approach. The lemmas and theorems follow closely those of a formalisation in Coq and developed jointly by Jad Hamza and myself. Due to the encoding of the state as an expression by the algorithm, the proofs are rather straightforward and, hopefully, easy to follow.

4.1 Unambiguous Context-Free Expressions

Natural languages are very often riddled with syntactic ambiguities. Sentences in natural languages often have multiple meanings, and knowing which one is intended by the speaker frequently depends on the context. This ambiguity allows for terseness and efficient human communication. When parsing natural languages, handling ambiguities is primordial.

On the other hand, in the context of formal languages, non-ambiguity is the norm. One of the very first phases of formal language tools, such as compilers, is converting input text into a structured representation that the rest of the tool can then handle. This parsing process is entirely deterministic: Given the same input, the same structured representation is to be produced by the parser. In this setting, a parser is simply a function from input tokens to some sort of structured value. Unambiguous context-free expressions, as descriptions of exactly such functions, appear to be a perfect fit for such a task.

Given an environment Γ, unambiguous context-free expressions e are those that, for all tokens sequences ts, assign at most a single value v to ts. Conversely, ambiguous context-free expressions are those which are not unambiguous. Unambiguous context-free expressions are descriptions of deterministic functions from input tokens to values.

Unfortunately, deciding whether a context-free expression is ambiguous is impossible in the general case, as it is equivalent to the problem of context-free grammar ambiguity which is itself undecidable (Hopcroft et al., 2001a, Theorem 9.20 on pages 405–406.). Therefore, all hopes of precisely deciding whether a context-free expression is unambiguous vanish.

In practice, in addition to unambiguity, there are also performance requirements. The usual target for parsers in compilers is linearity: The parsing time should be only linearly proportional to the size of the input. In this chapter, I will focus on a class of context-free expressions called LL(1) context-free expressions. I will formally describe the class and demonstrate how to efficiently decide whether a context-free expression is part of that class. In the later parts of the chapter, I will show a parsing algorithm for such expressions, and prove its correctness. Although simple and elegant, the parsing algorithm will prove to be performing poorly. In Chapter 5, I will demonstrate that a simple change in the data structure used to represent expressions turns this algorithm into an efficient linear-time parsing algorithm.

Even though unambiguity is undecidable for context-free *grammars*, there still exists useful subclasses of unambiguous context-free grammars that are decidable. LL(1) context-free grammars are such a class. In addition to being unambiguous, such LL(1) grammars are know to support linear-time parsing (Aho et al., 2006). Many programming languages, most notably Python (Van Rossum and Drake, 2009), have their syntax described using LL(1) grammars.

The traditional LL(1) parsing algorithm, which operates on a table and a stack, runs in time linear in the size of the input. In this chapter, I will transpose the notion of LL(1) grammars to the realm of context-free expressions. The characterisation that I propose is not a direct translation of that of LL(1) grammars. Indeed, I will show a novel characterisation of the class based on the idea of *should-not-follow* sets as opposed to the traditional `FOLLOW` sets. I will arrive at such a characterisation through the lens of derivation.

4.2 Towards Unambiguity via Derivation

In this section, I will build a class of unambiguous context-free expressions that will end up corresponding to the class of LL(1) context-free grammars. I will arrive at this class, which I will also call LL(1) context-free expressions, in a very natural fashion thanks to context-free derivation.

In the introduction of this chapter, I defined a context-free expression e with associated environment Γ to be *unambiguous* if and only if, for all token sequences ts, there was at most one value v such that $e \vdash_\Gamma ts \rightsquigarrow v$.

A first observation to be made is that, in order for a context-free expression to be unambiguous, it should at least be unambiguous for the empty sequence of tokens. If an expression assigns two different values to the empty sequence of tokens, then trivially that expression is ambiguous. That observation, as trivial as it may seem, is an important one to make.

Building on that first observation, a second observation is that if, for any token sequence ts, there exists *at least one* derivative of the expression by ts that is unambiguous for the empty sequence of tokens, then the original expression is bound to be unambiguous. Typically, this one derivative can be the result of repeated application of a derivation function such as the one inspired by Might et al. (2011) and shown in Section 3.9.

To substantiate this second observation, consider an expression e in an environment Γ such that for all token sequences ts there exists a derivative by ts that is unambiguous for the empty sequence of tokens. Towards a contradiction, assume that the original expression is ambiguous. Therefore, let ts be a sequence of tokens and v_1, v_2 be two distinct values that witness an ambiguity.

$$e \vdash_\Gamma ts \rightsquigarrow v_1$$
$$e \vdash_\Gamma ts \rightsquigarrow v_2$$

Let e' in environment Γ' be a derivative of the first expression by the token sequence ts that is unambiguous for the empty sequence of tokens.

$$\Gamma, e \gg^{ts} \Gamma', e'$$

From the fact that it is a derivative of Γ, e by ts, we also have that:

$$e' \vdash_{\Gamma'} \langle\rangle \rightsquigarrow v_1$$
$$e' \vdash_{\Gamma'} \langle\rangle \rightsquigarrow v_2$$

Which contradicts the assumption that this particular derivative is unambiguous for the empty sequence of tokens. Therefore, the original expression Γ, e is bound to be unambiguous.

Thanks to this insight, the context-free expression unambiguity problem can be reframed as a problem on expressions and a derivation function. Given a derivation function, an expression is unambiguous if and only if it is unambiguous for the empty sequence of tokens, and remains so after an arbitrary number of applications of the derivation function.

The derivation function that I will use in this explanatory argument is the one from Section 3.9. In the actual formal development that follows this introduction, I will substitute it for another derivation function that is tailored to the class of expressions that I will be building. Introducing that specialised derivation function at this point would be inappropriate and rather abrupt, as it makes assumptions about the inhabitants of the class that I am trying to progressively ease into! In order to save ourselves from circular arguments, the derivation function from Section 3.9 adapted from Might et al. (2011) is used.

As previously mentioned, context-free unambiguity is unfortunately undecidable. The class that I will be describing therefore has to be an approximation of this class of unambiguous context-free expressions. Since we wish for all elements of our class to be unambiguous, the class should be a subclass of unambiguous expressions. In order to be decidable, the class must be a strict subclass. In other words, approximations must be made in order for the class to potentially be decidable. The criteria to be rejected from the class are therefore expected to be somewhat imprecise.

4.2.1 First Criterion

The first criterion to be rejected from the class is the presence of a disjunction subexpression where both sides are nullable. When an expression contain such a subexpression, I will say that it exhibits a *Nullable/Nullable conflict*. All expressions that assign different values to the empty sequence of tokens exhibit such a conflict, and are therefore trivially ruled out by this criterion. Additionally, the criterion rightfully rejects some expressions which, although they themselves are not nullable, would exhibit multiple nullable derivations under some number of applications of the derivation function. For instance consider the following expression e and the derivative e' resulting from the application of the derivation function with the token t of kind k as argument.

$$\begin{aligned} e &= elem_k \cdot (\varepsilon_0 \vee \varepsilon_1) \\ e' &= \varepsilon_t \cdot (\varepsilon_0 \vee \varepsilon_1) \end{aligned}$$

Although e itself is not nullable, it contains a subexpression which accepts multiple derivations of nullability, namely $\varepsilon_0 \vee \varepsilon_1$. Under application of the derivation function, this subexpression ends up in a place where it contributes to the nullability of the expression e', thereby making e' ambiguous for the empty sequence of tokens. In turn, this shows that e is ambiguous.

Note that this first criterion is already somewhat imprecise. It can rules out unambiguous expressions where both sides produce the same value, or where the disjunction appears in an unproductive context, such as in the following examples:

$$\varepsilon_0 \vee \varepsilon_0 \qquad or \qquad (\varepsilon_0 \vee \varepsilon_1) \cdot \bot$$

However, rejecting such expressions is not impactful as they can be trivially transformed into equivalent expressions that are part of the class.

Unfortunately, this first criterion alone is not sufficient to rule out all ambiguous expressions. There exist ambiguous expressions that satisfy this criterion, such as:

$$(elem_k \cdot \varepsilon_0) \vee (elem_k \cdot \varepsilon_1)$$

The expression contains only a single disjunction subexpression, of which neither side is nullable. Therefore, the expression would not be rejected by the first criterion. It is however ambiguous, since the expression assigns both the value $(t, 0)$ and the value $(t, 1)$ to the sequence of tokens $\langle t \rangle$, and so for any token t of kind k.

Therefore, we should not only rule out expressions that contain disjunction subexpressions where both sides are nullable, but also all expressions which, under an arbitrary number of applications of the derivation function, contain such a disjunction subexpression.

Observe that the disjunction subexpressions that appear in the result of the derivation function from Section 3.9 are either present in the original expression, or are introduced by the derivation function. The derivation function introduces disjunctions in the following cases:

1. When deriving a disjunction, a disjunction of the derivatives is introduced in case both sides admit the derived token in their first sets.
2. When deriving a sequence when the left expression is nullable, a disjunction is also introduced when both sides admit the derived token in their first sets.
3. Additionally, some disjunctions may be introduced when encountering a sequence subexpression $e_1 \cdot e_2$ as part of the potential call to $\nu_\Gamma(e_1)$. Such disjunctions do not introduce Nullable/Nullable conflicts by themselves, as disjunctions are introduced there only when both sides are already nullable. When the expression satisfies this first criterion, calls to $\nu(\cdot)$ on subexpressions are bound not to introduce disjunctions.

One irrefutable way to ensure that the disjunctions introduced by the derivation function are not problematic is to make sure that the derivation function never introduces disjunctions in the first place! In conjunction with the first criterion, the next two criteria impose restrictions on the expressions of the class to ensure that this holds.

4.2.2 Second Criterion

The second criterion imposed on all expressions of the class is the absence of disjunctions where both sides share a common kind in their *first sets*. When an expression does contain such a subexpression, I will say that the expression exhibits a *First/First conflict*. Given a disjunction $e_1 \vee e_2$, this criterion ensures that, for any token t of kind k, either $k \not\in \textsc{first}_\Gamma(e_1)$ or $k \not\in \textsc{first}_\Gamma(e_2)$, which ensures that the derivation function can not ever introduce a disjunction when encountering a disjunction.

Interestingly, this criterion is preserved by derivation. Indeed, by design the derivation function never introduces disjunctions for elements of the class. Therefore, conflicting disjunctions may not be created by derivation.

Note that this criterion rules out some unambiguous expressions such that:

$$(elem_k \cdot \varepsilon_t) \vee (elem_k \cdot elem_k)$$

In some cases however it is possible to obtain an equivalent expression that is part of the class, and so through a process called *left-factoring*. In this particular instance, the above expression is equivalent to the following expression which does not exhibit such a conflict:

$$elem_k \cdot (\varepsilon_t \vee elem_k)$$

4.2.3 Third Criterion

The third and final criterion aims to ensure that disjunctions are never introduced when encountering a sequence expression during derivation. The derivation function will introduce such a disjunction when encountering a sequence $e_1 \cdot e_2$ where e_1 is nullable and the first sets of e_1 and e_2 are not disjoint.

Therefore, one criterion that seems rather natural is to exclude from the class those expressions which contain sequence subexpressions where the left is nullable and both sides share a common kind in their first sets. When directly applied on an expression that is not excluded by this criterion, the derivation function will indeed not introduce a disjunction.

Unfortunately, this property is *not* preserved by derivation, as witnessed by the following example.

$$\begin{aligned} e &= (elem_k \cdot (\varepsilon_t \vee elem_k)) \cdot elem_k \\ e' &= (\varepsilon_t \cdot (\varepsilon_t \vee elem_k)) \cdot elem_k \end{aligned}$$

In this example, the expression e' is the result of the derivation of e by a token t of kind k. Observe that the expression e did not violate the suggested property. Although derivation did not introduce any disjunction, it did introduce a fresh sequence where the left is nullable and both sides share a common element in their first sets.

To circumvent this issue, the approach that I propose in this thesis is to introduce a property of expressions called the *should-not-follow* set. Intuitively, the should-not-follow set of an expression is the set of kinds that are prohibited to appear as part of the first set of any directly subsequent expression. I will shortly introduce this property formally.

With this property in place, the third and final criterion will be that, for all sequence subexpressions $e_1 \cdot e_2$ appearing in the expression, the should-not-follow set of e_1 should be disjoint from the first set of e_2. As I will later formally prove, this criterion ensures, for any sequence subexpression $e_1 \cdot e_2$, that when e_1 is nullable then the first set of e_1 is disjoint from the first set of e_2. Additionally, and importantly, this criterion is preserved by derivation. When an expression violates this criterion, I will say that it exhibits a *First/Follow conflict*.

4.3 The LL(1) Class

So far in this chapter, I have introduced three decidable criterions which characterise a subclass of unambiguous context-free expressions. Violations of those criterions are called conflicts, and can be of the following form:

Nullable/Nullable conflict: In which a context-free expression contains a disjunction where both sides are nullable. The absence of this conflict ensures that the expression can not be ambiguous for the empty sequence of tokens.

First/First conflict: In which the expression contains a disjunction where both sides have non-disjoint first sets. The presence of this conflict means the derivation function may introduce further disjunctions, which may exhibit conflicts. The absence of such conflicts rules out the introduction by the derivation function of any Nullable/Nullable or First/First conflicts.

First/Follow conflict: In which the expression contains a sequence $e_1 \cdot e_2$ where the *should-not-follow* set of the left side intersects with the first set of the right side. The presence of this conflict makes it so that repeated application of the derivation function is no longer guaranteed not to introduce disjunctions when encountering sequences. Absence of such conflicts rules out the possibility a sequence where the left side is nullable and both sides have non-disjoint first set ever being introduced. Such a sequence, when subsequently derived, would introduce a disjunction, which could then potentially exhibit Nullable/Nullable or First/First conflicts.

As further made evident by the name of the various conflicts, the class of expressions that I have described ends up corresponding to the class of LL(1) context-free grammars. LL(1) grammars are context-free grammars that are exempt from conflicts similar to those that have been introduced here. In this grammar setting, the absence of such conflicts makes it possible to always decide which grammar rule is to be applied looking at the next token of input alone. In turn, this allows for a conversion of the grammar to an efficient deterministic stack-based automaton, in which transitions are taken only looking at the immediately next token of input.

Interestingly, in the realm of context-free expressions, the absence of conflicts as defined in this thesis make it possible to immediately resolve all alternatives encountered by the derivation function during parsing, and so only looking at the next input token, that is the token argument to the derivation function.

In the next sections, I will formally introduce the notion of should-not-follow sets and of LL(1) conflicts through inductive predicates. Next, I will introduce a derivation function specialised to LL(1) expressions and show an adaptation of the *parsing with derivatives* algorithm based on that derivation function. In the subsequent chapter, I will present a technique based on Huet's zipper to turn this *parsing with derivatives* algorithm into an efficient algorithm that will bear striking similarities with the deterministic stack-based automaton employed to execute LL(1) grammars.

4.3.1 Should-Not-Follow Set

The should-not-follow set of an expression is the set of kinds that could introduce an ambiguity if an expression directly following in sequence was to contain that kind in its first set. Membership in the should-not-follow set is inductively defined in Figure 4.1.

This definition differs from the similar concept of `FLast` set used by Krishnaswami and Yallop (2019) and introduced in earlier works (Johnstone and Scott, 1998; Brüggemann-Klein and Wood, 1992). Instead of introducing elements to the set in the case of disjunctions, the previous works do so in the case of sequences. Although the two concepts share the same goal of detecting conflicts in sequences, their definition of `FLast` imposes additional restrictions on expressions: nullable expressions are disallowed on left part of sequences. This restriction is *not* needed in this approach, nor in conventional LL(1) definition for context-free grammars (Aho and Ullman, 1972, Theorem 5.3, page 343).

The concept of should-not-follow set is used as an alternative to the concept of `FOLLOW` set generally used in the context of traditional LL(1) parsing. Whereas the `FOLLOW` set is a global property of a grammar, the should-not-follow set of an expression enjoys a more local nature. The should-not-follow set of an expression is a property of the expression alone; it does not matter *where* the expression appears, only the expression itself induces the set.

Kinds k are added to the should-not-follow set in the case of disjunctions when one side starts with the given kind k and the other side is nullable (rules SDISFN and SDISNF). For all constructs, the should-not-follow sets are additionally inherited from children expressions. In case of sequences, the elements are only inherited from a side when the other side is productive, respectively nullable (rules SSEQL and SSEQR).

As expressed by the following theorem, membership of a kind k in the should-not-follow set of an expression e indicates that there exists a token t and two sequences of tokens ts_1 and ts_2 such that both ts_1 and the sequence $ts_1 \mathbin{++} t :: ts_2$ are part of the language of e. If the expression e was to be followed by an expression starting with a token of kind k, a parser could not directly decide if it should stay in e or leave e and move its attention to the subsequent expression.

$$\textsc{SDisFN}\ \frac{k \in \textsc{first}_\Gamma(e_1) \qquad \textsc{nullable}_\Gamma(e_2)}{k \in \textsc{sn-follow}_\Gamma(e_1 \vee e_2)} \qquad \textsc{SDisNF}\ \frac{\textsc{nullable}_\Gamma(e_1) \qquad k \in \textsc{first}_\Gamma(e_2)}{k \in \textsc{sn-follow}_\Gamma(e_1 \vee e_2)}$$

$$\textsc{SDisL}\ \frac{k \in \textsc{sn-follow}_\Gamma(e_1)}{k \in \textsc{sn-follow}_\Gamma(e_1 \vee e_2)} \qquad \textsc{SDisR}\ \frac{k \in \textsc{sn-follow}_\Gamma(e_2)}{k \in \textsc{sn-follow}_\Gamma(e_1 \vee e_2)}$$

$$\textsc{SSeqL}\ \frac{k \in \textsc{sn-follow}_\Gamma(e_1) \qquad \textsc{nullable}_\Gamma(e_2)}{k \in \textsc{sn-follow}_\Gamma(e_1 \cdot e_2)}$$

$$\textsc{SSeqR}\ \frac{\textsc{productive}_\Gamma(e_1) \qquad k \in \textsc{sn-follow}_\Gamma(e_2)}{k \in \textsc{sn-follow}_\Gamma(e_1 \cdot e_2)}$$

$$\textsc{SMap}\ \frac{k \in \textsc{sn-follow}_\Gamma(e)}{k \in \textsc{sn-follow}_\Gamma(f \circledcirc e)} \qquad \textsc{SVar}\ \frac{e = \Gamma(x) \qquad k \in \textsc{sn-follow}_\Gamma(e)}{k \in \textsc{sn-follow}_\Gamma(var_x)}$$

Figure 4.1 – Rules for inclusion in the should-not-follow set.

Theorem 4.1 (Soundness)**.** *For any environment* Γ*, expression* e*, and kind* k*, if* k *is part of the should-not-follow set of* e*, then there exist a token* t *of kind* k *and (possibly empty) sequences of token* ts_1 *and* ts_2 *such that:*

$$e \vdash_\Gamma ts_1 \rightsquigarrow v_1 \quad \wedge \quad e \vdash_\Gamma ts_1 \mathbin{++} (t :: ts_2) \rightsquigarrow v_2$$

Proof. Follows by induction on the derivation of $k \in \textsc{sn-follow}_\Gamma(e)$. □

The should-not-follow set is however not complete in the general case: It is possible to devise an environment Γ and an expression e, such that there exist a token t of kind k that is not part of the should-not-follow set of e, and two sequences ts_1 and ts_2 such that ts_1 and $ts_1 \mathbin{++} t :: ts_2$ are in the language of e.

For example, let Γ be an arbitrary environment and let t be an arbitrary token of kind k. Consider the expression $e = (elem_k \cdot \varepsilon_t) \vee (elem_k \cdot elem_k)$. Clearly, both the sequences $\langle t \rangle$ and $\langle t, t \rangle$ are part of the language of e. It is however not the case that $k \in \textsc{sn-follow}_\Gamma(e)$. Indeed, in this example, the should-not-follow set of e is empty.

The should-not-follow set will however be *complete enough* for our purposes. The following theorem shows that, when an expression e is nullable, its should-not-follow set is a superset of its first set. This ensures that, for a sequence $e_1 \cdot e_2$, if the should-not-follow set of e_1 is disjoint from the first set of e_2, it is impossible for e_1 both to be nullable and to have a first set that intersects with the first set of e_2. Furthermore, I will later on show that the set is indeed complete in the case of LL(1) expressions.

Theorem 4.2 (Nullable/First Completeness)**.** *For any expression e, environment Γ and kind k, if k is part of the first set of e and e is nullable, then k is part of the should-not-follow set of e.*

Proof. By induction on the derivation of $k \in \textsc{first}_\Gamma(e)$, I show that if e is nullable then it must be the case that $k \in \textsc{sn-follow}_\Gamma(e)$. The only interesting case is when e is a disjunction between an expression e_1 such that $k \in \textsc{first}_\Gamma(e_1)$ where e_1 is not nullable, as all other cases follow directly by induction hypothesis.

In that case, assuming e is nullable forces the other branch of the disjunction, e_2, to be nullable. Since e is a disjunction between an expression e_1 where $k \in \textsc{first}_\Gamma(e_1)$ and an expression e_2 where $\textsc{nullable}_\Gamma(e_2)$, it follows that $k \in \textsc{sn-follow}_\Gamma(e)$ by the rule SDisL or SDisR. □

4.3.2 LL(1) Conflicts

Finally, I formally introduce the notion of LL(1) conflicts. The presence of LL(1) conflicts is defined inductively in Figure 4.2.

$$\textsc{CDisNN}\ \frac{\textsc{nullable}_\Gamma(e_1) \qquad \textsc{nullable}_\Gamma(e_2)}{\textsc{has-conflict}_\Gamma(e_1 \vee e_2)} \qquad \textsc{CDisFF}\ \frac{k \in \textsc{first}_\Gamma(e_1) \qquad k \in \textsc{first}_\Gamma(e_2)}{\textsc{has-conflict}_\Gamma(e_1 \vee e_2)}$$

$$\textsc{CSeqSF}\ \frac{k \in \textsc{sn-follow}_\Gamma(e_1) \qquad k \in \textsc{first}_\Gamma(e_2)}{\textsc{has-conflict}_\Gamma(e_1 \cdot e_2)}$$

$$\textsc{CDisL}\ \frac{\textsc{has-conflict}_\Gamma(e_1)}{\textsc{has-conflict}_\Gamma(e_1 \vee e_2)} \qquad \textsc{CDisR}\ \frac{\textsc{has-conflict}_\Gamma(e_2)}{\textsc{has-conflict}_\Gamma(e_1 \vee e_2)}$$

$$\textsc{CSeqL}\ \frac{\textsc{has-conflict}_\Gamma(e_1)}{\textsc{has-conflict}_\Gamma(e_1 \cdot e_2)} \qquad \textsc{CSeqR}\ \frac{\textsc{has-conflict}_\Gamma(e_2)}{\textsc{has-conflict}_\Gamma(e_1 \cdot e_2)}$$

$$\textsc{CMap}\ \frac{\textsc{has-conflict}_\Gamma(e)}{\textsc{has-conflict}_\Gamma(f \circledcirc e)} \qquad \textsc{CVar}\ \frac{e = \Gamma(x) \qquad \textsc{has-conflict}_\Gamma(e)}{\textsc{has-conflict}_\Gamma(var_x)}$$

Figure 4.2 – Rules for existence of LL(1) conflicts.

The rules express that an expression exhibits LL(1) conflicts if it contains a disjunction where both sides are nullable (CDisNN) or start with the name token kind (CDisFF), or when it contains a sequence where the should-not-follow set of the left side intersects with the first set of the right side (CSeqSF). Conflicts rooted at CDisNN are called Nullable/Nullable conflicts, those rooted at CDisFF are called First/First conflicts and finally those rooted at CSeqSF are called First/Follow conflicts.

4.3.3 The LL(1) Property

Given an environment Γ, an expression e is LL(1) if and only if it has no LL(1) conflicts.

$$\text{LL1}_\Gamma(e) \iff \neg\textsc{has-conflict}_\Gamma(e)$$

Observe that the LL(1) property applies throughout the expression: When an expression is LL(1), then so are all its subexpressions.

4.3.4 Properties of LL(1) Expressions

LL(1) context-free expressions enjoy several interesting properties. The first of those properties is unambiguity. The following theorem states that there exists at most one value associated with each token sequence.

Theorem 4.3 (Unambiguity of LL(1) Expressions)**.** *For all LL(1) expressions e and environment* Γ*, token sequences* ts *and values* v_1 *and* v_2*:*

$$e \vdash_\Gamma ts \rightsquigarrow v_1 \ \land \ e \vdash_\Gamma ts \rightsquigarrow v_2 \implies v_1 = v_2$$

Proof. By induction on the derivation of $e \vdash_\Gamma ts \rightsquigarrow v_1$. In every case, the LL(1) property enforces that the rule that was applied for $e \vdash_\Gamma ts \rightsquigarrow v_1$ be applied for $e \vdash_\Gamma ts \rightsquigarrow v_2$, hence showing the equality between v_1 and v_2. □

In the same vein, as expressed by the following theorems, LL(1) expressions also have the property that all derivations of nullability are of the same size. The same holds for derivations of membership of a kind in the first set of an expression. Thanks to this property, derivations appearing inside other such derivations can be therefore seen as *smaller*. This will prove important as recursive procedures on LL(1) expressions following the exact same structure as either a $\textsc{nullable}_\Gamma(\cdot)$ or $k \in \textsc{first}_\Gamma(\cdot)$ derivation will be bound to terminate. Termination of recursive functions operating on context-free expressions is non-obvious in the general case, as the recursive argument expression may grow structurally larger in the case of variables if the associated expressions in the environment are simply unfolded. In the case of LL(1), thanks to the following theorems, termination will be guaranteed even when variables are unfolded.

Theorem 4.4 (Unicity of Derivations of Nullability)**.** *For environments* Γ *and all LL(1) expressions e such that* $\textsc{nullable}_\Gamma(e)$*, there exists a unique derivation of* $\textsc{nullable}_\Gamma(e)$.

Proof. Let Γ be an environment and e be an LL(1) expression. Let D_1 and D_2 be two derivations of the fact that $\textsc{nullable}_\Gamma(e)$. The proof proceeds by induction on D_1. In each case, the LL(1) property ensures that rule on top of D_1 also appears on top of D_2, thereby showing that D_1 is equal to D_2. □

Theorem 4.5 (Unicity of Derivations of First Set Membership)**.** *For environments* Γ*, all LL(1) expressions e and kinds k such that* $k \in \textsc{first}_\Gamma(e)$*, all derivations of* $k \in \textsc{first}_\Gamma(e)$ *are equal modulo derivations of productivity.*

Proof. Let Γ be an environment, e be an LL(1) expression, and k be a kind. Let D_1 and D_2 be two derivations of the fact that $k \in \textsc{first}_\Gamma(e)$. I will show that D_1 is equivalent to D_2 modulo derivations of productivity. The proof proceeds by induction on D_1. In each case, the LL(1) property ensures that rule on top of D_1 also appears on top of D_2. In the case when nullability is needed as a premise (rule FSEQR), Theorem 4.4 applies. □

Left-Recursivity

According to the definitions of left-recursivity and the LL(1) property given in Section 3.7.4, it is technically possible for entries in the environment to be left-recursive and LL(1). As an example, consider the singleton environment Γ_R where $\Gamma_R(x) = var_x$ for some identifier x. In this environment, the expression $\Gamma(x)$ does not exhibit any conflicts, and is therefore LL(1). It is also the case that x is left-recursive.

As shown by the following theorem, the precise restriction that the LL(1) property imposes is that LL(1) expressions may not contain, directly or indirectly, subexpressions of the form var_x such that both $\Gamma(x)$ is productive and x is left-recursive.

In practice, the presence of unproductive entries in the environment may strongly suggest a design or coding error. Their presence could be reported as an actual error or as a simple warning. This would be especially important when the expression associated with the unproductive entry is recursive, since the recursive structure of the expression may be the cause of the issue.

Theorem 4.6. *For any environment* Γ *and identifier x, if* $\Gamma(x)$ *is such that:*

$$\textsc{productive}_\Gamma(\Gamma(x)) \quad \wedge \quad x \in \textsc{visitable}_\Gamma(\Gamma(x))$$

Then $\Gamma(x)$ *is not LL(1). Furthermore, any expression e which would directly or indirectly contain* var_x *as a subexpression would then also not be LL(1).*

Proof. Let Γ be an environment, and x an identifier such that x is visitable in $\Gamma(x)$ and $\Gamma(x)$ is productive.

Since the expression $\Gamma(x)$ is productive, it must be the case that it is nullable or its first set is non-empty.

Consider that $\Gamma(x)$ is nullable. Let D_1 be a derivation of $\textsc{nullable}_\Gamma(\Gamma(x))$. Additionally, let D_2 be a derivation of $\textsc{nullable}_\Gamma(\Gamma(x))$ that contains D_1 as a subtree. Since $x \in \textsc{visitable}_\Gamma(\Gamma(x))$, D_2 is bound to exist. By Theorem 4.4, the expression $\Gamma(x)$ can not be LL(1).

Now consider that there exists a kind k such that $k \in \textsc{first}_\Gamma(\Gamma(x))$. Let D_1 be a derivation of $k \in \textsc{first}_\Gamma(\Gamma(x))$ and let D_2 be a different derivation of $k \in \textsc{first}_\Gamma(\Gamma(x))$ that contains D_1 as a subtree. Since $x \in \textsc{visitable}_\Gamma(\Gamma(x))$, D_2 is bound to exist. By Theorem 4.4, the expression $\Gamma(x)$ can not be LL(1), which concludes the proof.

□

4.4 LL(1) Parsing with Derivatives

In the earlier parts of this chapter, I have characterised the class of LL(1) context-free expressions and discussed some of its properties. In this section, I will proceed to describe a parsing algorithm for this class of expressions based on Brzozowski's derivatives. Through an example, the algorithm presented in this section will be shown to be quite inefficient. However, as I will show in the next chapter, through a simple change of data structure, the algorithm presented in this section can be turned into an efficient linear time parsing algorithm.

4.4.1 Values from Nullable LL(1) Expressions

Before I can proceed any further, I need to introduce the function $\texttt{null}_\Gamma(e)$, which, given an environment Γ and a *nullable LL(1)* expression e, computes the (unique) value associated with the empty sequence of tokens by the expression e. The function is defined recursively as follows:

$$\begin{aligned}
\texttt{null}_\Gamma(\varepsilon_v) &:= v \\
\texttt{null}_\Gamma(e_1 \vee e_2) &:= \begin{cases} \texttt{null}_\Gamma(e_1) & \text{if } \textsc{nullable}_\Gamma(e_1) \\ \texttt{null}_\Gamma(e_2) & \text{otherwise} \end{cases} \\
\texttt{null}_\Gamma(e_1 \cdot e_2) &:= (\texttt{null}_\Gamma(e_1), \texttt{null}_\Gamma(e_2)) \\
\texttt{null}_\Gamma(f \odot e) &:= f(\texttt{null}_\Gamma(e)) \\
\texttt{null}_\Gamma(var_x) &:= \texttt{null}_\Gamma(\Gamma(e))
\end{aligned}$$

The function $\texttt{null}_\Gamma(\cdot)$ is only defined for nullable LL(1) expressions. Although simple, it is worth checking the validity of this definition. The upcoming theorems ensure that $\texttt{null}_\Gamma(\cdot)$ is well-defined.

Theorem 4.7 (Match-completeness)**.** *The definition of* $\texttt{null}_\Gamma(\cdot)$ *handles all constructs that can appear at the top of* nullable *expressions e.*

Proof. The only two constructs which are not handled are $elem_k$ and $\perp$, both of which are not nullable. □

Theorem 4.8 (Preconditions)**.** *All arguments e' of recursive calls made in the body of* $\mathit{null}_\Gamma(e)$ *satisfy* $\text{LL1}_\Gamma(e')$ *and* $\text{NULLABLE}_\Gamma(e')$ *given that* $\text{LL1}_\Gamma(e)$ *and* $\text{NULLABLE}_\Gamma(e)$.

Proof. By simple case analysis. □

Theorem 4.9 (Well-foundedness)**.** *The computation of* $\mathit{null}_\Gamma(e)$ *terminates for any nullable LL(1) expression e.*

Proof. I show the well-foundedness of the $\texttt{null}_\Gamma(\cdot)$ recursive function by demonstrating that the size of the derivation of $\text{NULLABLE}_\Gamma(\cdot)$ is strictly decreasing for all recursive arguments.

Let Γ be an environment and e be a nullable LL(1) context-free expression. By Theorem 4.4, the derivation of $\text{NULLABLE}_\Gamma(e)$ is unique. Then, consider a call to $\texttt{null}_\Gamma(e)$. Observe that $\text{NULLABLE}_\Gamma(e)$ can be derived from the premises $\text{NULLABLE}_\Gamma(e')$ where the various e' are arguments of recursive calls. Therefore, by uniqueness of derivations of $\text{NULLABLE}_\Gamma(\cdot)$ for nullable LL(1) expressions, it must be the case that derivations of $\text{NULLABLE}_\Gamma(e')$ are all smaller than the derivation of $\text{NULLABLE}_\Gamma(e)$, as they are strictly contained within the (finite) derivation of $\text{NULLABLE}_\Gamma(e)$. As the size of the derivation strictly decreases in each recursive call, the function is bound to terminate. □

Theorem 4.10 (Correctness)**.** *For all environments* Γ*, context-free expression e, if e is nullable and e is LL(1), then:*

$$\mathit{null}_\Gamma(e) = v \iff e \vdash_\Gamma \langle\rangle \rightsquigarrow v$$

Proof. By induction on the derivation of $\text{NULLABLE}_\Gamma(e)$. □

4.4.2 An Induction Principle for LL(1) Expressions

Most proofs on LL(1) expressions rely on induction. Unfortunately, structural induction on expressions, although simple, will generally prove useless. The culprit is the var_x case, in which no induction hypothesis is available, making it impossible to further make progress. The solution is generally to apply induction on the derivations of some inductive properties such as $\text{NULLABLE}_\Gamma(e)$ or $k \in \text{FIRST}_\Gamma(e)$ given as assumption.

In order to simplify proofs on LL(1) expressions with a given k in their first set, I introduce an induction principle called the *LL(1) induction principle.* The following theorem states the induction principle.

Lemma 4.1 (The LL(1) Induction Principle). *For a given environment* Γ *and kind* k*, to show that a property* $P[e]$ *holds for all LL(1) context-free expression* e *with* $k \in \textsc{first}_\Gamma(e)$*, it suffices to show all of the following:*

1. $P[elem_k]$.
2. $P[e_1 \vee e_2]$ *assuming* $k \in \textsc{first}_\Gamma(e_1)$, $k \notin \textsc{first}_\Gamma(e_2)$, $\mathrm{LL1}_\Gamma(e_1 \vee e_2)$ *and* $P[e_1]$.
3. $P[e_1 \vee e_2]$ *assuming* $k \notin \textsc{first}_\Gamma(e_1)$, $k \in \textsc{first}_\Gamma(e_2)$, $\mathrm{LL1}_\Gamma(e_1 \vee e_2)$ *and* $P[e_2]$.
4. $P[e_1 \cdot e_2]$ *assuming* $k \in \textsc{first}_\Gamma(e_1)$, $\textsc{nullable}_\Gamma(e_1) \implies k \notin \textsc{first}_\Gamma(e_2)$, $\textsc{productive}_\Gamma(e_2)$, $\mathrm{LL1}_\Gamma(e_1 \cdot e_2)$ *and* $P[e_1]$.
5. $P[e_1 \cdot e_2]$ *assuming* $\textsc{nullable}_\Gamma(e_1)$, $k \notin \textsc{first}_\Gamma(e_1)$, $k \in \textsc{first}_\Gamma(e_2)$, $\mathrm{LL1}_\Gamma(e_1 \cdot e_2)$ *and* $P[e_2]$.
6. $P[f \circledcirc e]$ *assuming* $k \in \textsc{first}_\Gamma(e)$, $\mathrm{LL1}_\Gamma(e)$ *and* $P[e]$.
7. $P[var_x]$ *assuming* $k \in \textsc{first}_\Gamma(\Gamma(x))$, $\mathrm{LL1}_\Gamma(\Gamma(x))$ *and* $P[\Gamma(x)]$.

Proof. Let Γ be an environment and let k be a kind. Let e be a context-free expression such that $k \in \textsc{first}_\Gamma(e)$. Let P_1 to P_7 be proofs of $P[e]$ depending on assumptions as specified in the 7 different cases above in the theorem statement. The proof of $\mathrm{LL1}_\Gamma(e) \implies P[e]$ follows by induction on the derivation of $k \in \textsc{first}_\Gamma(e)$:

1. Consider the case `FElem`. In that case, one must show $P[elem_k]$, which is directly given by P_1.
2. Consider the case `FDisL`. In that case, one must show $P[e_1 \vee e_2]$ assuming $k \in \textsc{first}(e_1)$, $\mathrm{LL1}_\Gamma(e_1 \vee e_2)$ and $\mathrm{LL1}_\Gamma(e_1) \implies P[e_1]$ (IH). From $\mathrm{LL1}_\Gamma(e_1 \vee e_2)$, one gets $\mathrm{LL1}_\Gamma(e_1)$ and therefore $P[e_1]$. From $\mathrm{LL1}_\Gamma(e_1 \vee e_2)$ and $k \in \textsc{first}(e_1)$, one gets $k \notin \textsc{first}(e_2)$. Finally, as all its assumptions as satisfied, P_2 proves $P[e_1 \vee e_2]$.
3. Consider the case `FDisR`. In that case, one must show $P[e_1 \vee e_2]$ assuming $k \in \textsc{first}(e_2)$, $\mathrm{LL1}_\Gamma(e_1 \vee e_2)$ and $\mathrm{LL1}_\Gamma(e_2) \implies P[e_2]$ (IH). From $\mathrm{LL1}_\Gamma(e_1 \vee e_2)$, one gets $\mathrm{LL1}_\Gamma(e_2)$ and therefore $P[e_2]$. From $\mathrm{LL1}_\Gamma(e_1 \vee e_2)$ and $k \in \textsc{first}(e_2)$, one gets $k \notin \textsc{first}(e_1)$. Finally, as all its assumptions as satisfied, P_3 proves $P[e_1 \vee e_2]$.
4. Consider the case `FSeqL`. In that case, one must show $P[e_1 \cdot e_2]$ assuming $k \in \textsc{first}(e_1)$, $\textsc{productive}_\Gamma(e_2)$, $\mathrm{LL1}_\Gamma(e_1 \cdot e_2)$ and $\mathrm{LL1}_\Gamma(e_1) \implies P[e_1]$ (IH). From $\mathrm{LL1}_\Gamma(e_1 \cdot e_2)$, one gets $\mathrm{LL1}_\Gamma(e_1)$ and therefore $P[e_1]$. From $\mathrm{LL1}_\Gamma(e_1 \cdot e_2)$, $k \in \textsc{first}(e_1)$ and Theorem 4.2, one gets $\textsc{nullable}_\Gamma(e_1) \implies k \notin \textsc{first}_\Gamma(e_2)$. Finally, as all its assumptions as satisfied, P_4 proves $P[e_1 \cdot e_2]$.

5. Consider the case `FSeqR`. In that case, one must show $P[e_1 \cdot e_2]$ assuming $\textsc{nullable}_\Gamma(e_1)$, $k \in \textsc{first}(e_2)$, $\mathrm{LL1}_\Gamma(e_1 \cdot e_2)$ and $\mathrm{LL1}_\Gamma(e_2) \implies P[e_2]$ (IH). From $\mathrm{LL1}_\Gamma(e_1 \cdot e_2)$, one gets $\mathrm{LL1}_\Gamma(e_2)$ and therefore $P[e_2]$. From $\mathrm{LL1}_\Gamma(e_1 \cdot e_2)$, $\textsc{nullable}_\Gamma(e_1)$ and Theorem 4.2, one gets $k \notin \textsc{first}_\Gamma(e_1)$. Finally, as all its assumptions as satisfied, P_5 proves $P[e_1 \cdot e_2]$.

6. Consider the case `FMap`. In that case, one must show $P[f \circledcirc e]$ assuming $k \in \textsc{first}_\Gamma(e)$, $\mathrm{LL1}_\Gamma(f \circledcirc e)$ and $\mathrm{LL1}_\Gamma(e) \implies P[e]$ (IH). From $\mathrm{LL1}_\Gamma(f \circledcirc e)$, one gets $\mathrm{LL1}_\Gamma(e)$ and therefore $P[e]$. Finally, as all its assumptions as satisfied, P_6 proves $P[f \circledcirc e]$.

7. Consider the case `FVap`. In that case, one must show $P[var_x]$ assuming $k \in \textsc{first}_\Gamma(\Gamma(x))$, $\mathrm{LL1}_\Gamma(var_x)$ and $\mathrm{LL1}_\Gamma(\Gamma(x)) \implies P[\Gamma(x)]$ (IH). From $\mathrm{LL1}_\Gamma(var_x)$, one gets $\mathrm{LL1}_\Gamma(\Gamma(x))$ and therefore $P[\Gamma(x)]$. Finally, as all its assumptions as satisfied, P_7 proves $P[var_x]$.

□

4.4.3 Derivatives of LL(1) Expressions

Computing a derivative of an LL(1) expression by a token t is an almost risibly simple task, as expressed by the following definition of the LL(1) derivation function. Given a token t of kind k and an LL(1) expression e where $k \in \textsc{first}_\Gamma(e)$, the following function computes a derivative of e by the token t.

$$\begin{aligned}
\delta^t_\Gamma(elem_k) &:= \varepsilon_t \\
\delta^t_\Gamma(e_1 \vee e_2) &:= \begin{cases} \delta^t_\Gamma(e_1) & \text{if } \mathtt{kind}(t) \in \textsc{first}_\Gamma(e_1) \\ \delta^t_\Gamma(e_2) & \text{otherwise} \end{cases} \\
\delta^t_\Gamma(e_1 \cdot e_2) &:= \begin{cases} \delta^t_\Gamma(e_1) \cdot e_2 & \text{if } \mathtt{kind}(t) \in \textsc{first}(e_1) \\ \varepsilon_{\mathtt{null}_\Gamma(e_1)} \cdot \delta^t_\Gamma(e_2) & \text{otherwise} \end{cases} \\
\delta^t_\Gamma(f \circledcirc e) &:= f \circledcirc \delta^t_\Gamma(e) \\
\delta^t_\Gamma(var_x) &:= \delta^t_\Gamma(\Gamma(x))
\end{aligned}$$

Theorem 4.11 (Match-completeness). *The definition of $\delta^t_\Gamma(\cdot)$ handles all constructs that can appear at the top of expressions e such that $\mathtt{kind}(t) \in \textsc{first}_\Gamma(e)$.*

Proof. The only two constructs which are not handled are ε_v and $\perp$, both of which have empty first sets. □

Theorem 4.12 (Preconditions). *All arguments e' of recursive calls made in $\delta^t_\Gamma(e)$ satisfy $\mathrm{LL1}_\Gamma(e')$ and $\mathtt{kind}(t) \in \textsc{first}_\Gamma(e')$ given that $\mathrm{LL1}_\Gamma(e)$ and $\mathtt{kind}(t) \in \textsc{first}_\Gamma(e)$. The arguments of calls to $\mathtt{null}_\Gamma(\cdot)$ also satisfy the preconditions of $\mathtt{null}_\Gamma(\cdot)$, that is that the argument expression is LL(1) and nullable.*

Proof. By simple case analysis. □

Theorem 4.13 (Well-foundedness)**.** *The computation of $\delta^t_\Gamma(e)$ terminates for any LL(1) context-free expression e and environment Γ where $\mathit{kind}(t) \in \textsc{first}_\Gamma(e)$.*

Proof. Follows directly by the LL(1) induction principle introduced in Lemma 4.1. □

Compared to the derivative computation seen in Chapter 3, the definition specialised to LL(1) expressions is simpler in several ways:

- The definition is singly recursive. For binary constructs, the side which undergoes derivation is decided by nullability and first sets.
- Disjunctions are eliminated along the way, and are never introduced. By construction, the two sides of an LL(1) disjunction always have disjoint first sets. A similar property holds for sequences, for which a disjunction needed to be introduced in the general case. In the LL(1) case, no disjunction need to be introduced.
- The environment remains unchanged. In the general case, an entry in the environment needed to be added. This addition was crucial to handle left-recursive entries. This addition was also necessary to be able to explicitly share the representation of derived variables, were they to be encountered multiple times during a single derivation operation. In the case of LL(1), the situation is simpler. Indeed, left-recursive entries are never encountered. Furthermore, as derivation follows a single thread, it is impossible to encounter the same variable multiple times during a single derivation operation. For those reasons, the definition associated with the variable can be simply unfolded.

4.4.4 On the Correctness of LL(1) Derivation

In the previous two subsections, I have shown an induction principle for LL(1) expressions and have introduced a function to, supposedly, compute a derivative of LL(1) expressions. I will now proceed to show that LL(1) derivation function is indeed correct with respect to the semantics of context-free expressions. In addition, I will also show that the LL(1) property is preserved by LL(1) derivation.

Theorem 4.14 (Correctness)**.** *For any LL(1) expression e and environment Γ, token t of kind $k \in \textsc{first}_\Gamma(s)$, token sequence ts and value v, e associates the token sequence $t :: ts$ with the value v iff $\delta^t_\Gamma(e)$ associates the token sequence ts with the same value v, that is:*

$$e \gg^{\langle t \rangle}_\Gamma \delta^t_\Gamma(e)$$

Proof. By LL(1) induction (Lemma 4.1) on e. □

Lemma 4.2 (Should-not-follow monotonicity). *For any environment* Γ*, LL(1) expression* e*, and token* t *of kind* $k \in \textsc{first}(e)$*, the following holds:*

$$\textsc{sn-follow}_\Gamma(\delta^t_\Gamma(e)) \subseteq \textsc{sn-follow}_\Gamma(e)$$

Proof. By LL(1) induction (Lemma 4.1) on e. □

Theorem 4.15 (Preservation). *For any environment* Γ*, LL(1) expression* e*, and token* t *of kind* $k \in \textsc{first}(e)$*, the expression* $\delta^t_\Gamma(e)$ *is LL(1).*

Proof. By LL(1) induction (Lemma 4.1) on e. Only the fourth case of the induction is non-trivial.

In that case, one must show that $\text{LL1}_\Gamma(\delta^t_\Gamma(e_1 \cdot e_2))$ given $\texttt{kind}(t) \in \textsc{first}_\Gamma(e_1)$, $\textsc{nullable}_\Gamma(e_1) \implies \texttt{kind}(t) \notin \textsc{first}_\Gamma(e_2)$, $\textsc{productive}_\Gamma(e_2)$, $\text{LL1}_\Gamma(e_1 \cdot e_2)$ and $\text{LL1}_\Gamma(\delta^t_\Gamma(e_1))$. By those hypotheses, $\delta^t_\Gamma(e_1 \cdot e_2)$ reduces to the sequence $\delta^t_\Gamma(e_1) \cdot e_2$, which exhibits LL(1) conflicts in only three cases:

1. When the left expression $\delta^t_\Gamma(e_1)$ exhibits a conflict, which is prohibited by the induction hypothesis $\text{LL1}_\Gamma(\delta^t_\Gamma(e_1))$.
2. When the right expression e_2 exhibits a conflict, which is prohibited by $\text{LL1}_\Gamma(e_1 \cdot e_2)$.
3. When the should-not-follow set of $\delta^t_\Gamma(e_1)$ intersects the first set of e_2. By Lemma 4.2, the following holds:
$$\textsc{sn-follow}_\Gamma(\delta^t_\Gamma(e_1)) \subseteq \textsc{sn-follow}_\Gamma(e_1)$$
In addition, by $\text{LL1}_\Gamma(e_1 \cdot e_2)$, the set $\textsc{sn-follow}_\Gamma(e_1)$ is disjoint from $\textsc{first}_\Gamma(e_2)$, and therefore it must be the case that $\textsc{sn-follow}_\Gamma(\delta^t_\Gamma(e_1))$ is also disjoint from $\textsc{first}_\Gamma(e_2)$.

Therefore, the expression $\delta^t_\Gamma(e_1 \cdot e_2)$ is also LL(1). □

4.4.5 Should-Not-Follow Completeness

Previously in this chapter, the notion of should-not-follow set was shown to be incomplete in the general case. For LL(1) expressions however, the should-not-follow set exactly corresponds to set of kinds k that appear to continue valid sequences, that is:

$$\begin{aligned} \textsc{sn-follow}_\Gamma(e) = \{\, k \mid \exists t, ts_1, ts_2, v_1, v_2.\ & \texttt{kind}(t) = k \land \\ & e \vdash_\Gamma ts_1 \rightsquigarrow v_1 \land \\ & e \vdash_\Gamma ts_1 \mathbin{++} (t :: ts_2) \rightsquigarrow v_2 \,\} \end{aligned}$$

Theorem 4.16. *Given an environment* Γ*, the should-not-follow set of an LL(1) expression e equals the set*

$$\begin{aligned}\{\, k \mid \exists t, ts_1, ts_2, v_1, v_2.\ \mathit{kind}(t) = k \wedge{} \\ e \vdash_\Gamma ts_1 \rightsquigarrow v_1 \wedge{} \\ e \vdash_\Gamma ts_1 \mathbin{++} (t :: ts_2) \rightsquigarrow v_2 \,\}\end{aligned}$$

Proof. By Theorem 4.1, the only missing direction is completeness.

Let Γ be an environment. Let t be a token of kind k and let ts_1, ts_2 be sequences of tokens. The proof proceeds by structural induction on the sequence of tokens ts_1.

In both cases, let e be an LL(1) expression and assume that there exist two values v_1 and v_2 such that:

$$e \vdash_\Gamma ts_1 \rightsquigarrow v_1 \qquad \wedge \qquad e \vdash_\Gamma ts_1 \mathbin{++} (t :: ts_2) \rightsquigarrow v_2$$

1. Consider the case when ts_1 is $\langle\rangle$. By assumption, it is the case that e is nullable. Additionally, also by assumption, it is the case that $k \in \text{FIRST}_\Gamma(e)$. Therefore, by Theorem 4.2, it must be that $k \in \text{SN-FOLLOW}_\Gamma(e)$.
2. Consider the case when $ts_1 = t' :: ts_1'$ for some token t' and token sequence ts_1'. Consider the expression $\delta_\Gamma^{t'}(e)$. By Theorem 4.14, one derives that:

$$e \vdash_\Gamma ts_1' \rightsquigarrow v_1 \qquad \wedge \qquad e \vdash_\Gamma ts_1' \mathbin{++} (t :: ts_2) \rightsquigarrow v_2$$

   Thus, by induction hypothesis, one gets that $k \in \text{SN-FOLLOW}_\Gamma(\delta_\Gamma^{t'}(e))$. Finally, by Lemma 4.2, it must be the case that $\text{SN-FOLLOW}_\Gamma(\delta_\Gamma^{t'}(e)) \subseteq \text{SN-FOLLOW}_\Gamma(e)$, and therefore that $k \in \text{SN-FOLLOW}_\Gamma(e)$.

$\square$

4.4.6 Parsing Algorithm

The derivation function $\delta_\Gamma^t(\cdot)$ immediately yields a parsing algorithm for LL(1) context-free expressions that I call *simple LL(1) parsing with derivatives.*

$$\texttt{simple-ll1-parse}_\Gamma(e, \langle\rangle) := \begin{cases} \texttt{some}(\texttt{null}_\Gamma(e)) & \text{if } \text{NULLABLE}_\Gamma(e) \\ \texttt{none} & \text{otherwise} \end{cases}$$

$$\texttt{simple-ll1-parse}_\Gamma(e, t :: ts) := \begin{cases} \texttt{simple-ll1-parse}_\Gamma(\delta_\Gamma^t(e), ts) & \text{if } \texttt{kind}(t) \in \text{FIRST}_\Gamma(e) \\ \texttt{none} & \text{otherwise} \end{cases}$$

When the sequence of tokens to be parsed is empty, the algorithm simply checks if the expression is nullable, and in that case invokes the $\texttt{null}_\Gamma(\cdot)$ function to return the parsed value. In case the sequence of tokens is non-empty, the algorithm checks if the current expression admits the kind of the next token in its first set. If it is the case, then a derivative of the expression by that token is computed, and the algorithm is recursively applied. Otherwise, a parse error is returned.

Theorem 4.17 (Correctness of Simple LL(1) Parsing with Derivatives). *For all LL(1) context-free expressions e and environments Γ, for all sequences of tokens ts and for all values v:*

$$\texttt{simple-ll1-parse}_{\Gamma}(e, ts) = \textit{some}(v)$$
$$\Longleftrightarrow$$
$$e \vdash_{\Gamma} ts \rightsquigarrow v$$

Proof. By induction on the input token sequence. The proof follows immediately from the correctness of LL(1) derivation (Theorem 4.14), from preservation of the LL(1) property by LL(1) derivation (Theorem 4.15) and from the correctness of the $\texttt{null}_{\Gamma}(\cdot)$ function (Theorem 4.10). □

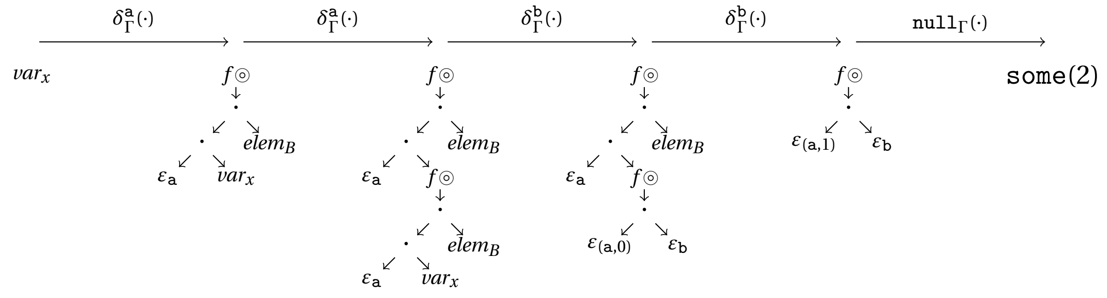
4.4.7 Example Execution

As an example execution of the algorithm, let us get back to the example context-free expression presented in Section 3.5. For this example, consider two tokens a and b of respective kinds A and B. Consider also the singleton environment Γ which maps the identifier x to the expression:

$$(f \circledcirc ((elem_A \cdot var_x) \cdot elem_B)) \vee \varepsilon_0$$

Where the function f takes as input values of the form $((t_1, n), t_2)$ and returns the value n. In this given environment, the expression var_x describes sequences of n a's followed by n b's, and so for any natural number n. The value associated with the sequence is the corresponding n.

Figure 4.3 presents the various phases the parsing algorithm goes through, and shows the state of the parsing algorithm before and after each token is processed.



$\delta^{a}_{\Gamma}(\cdot)$
$\delta^{a}_{\Gamma}(\cdot)$
$\delta^{b}_{\Gamma}(\cdot)$
$\delta^{b}_{\Gamma}(\cdot)$
$\texttt{null}_{\Gamma}(\cdot)$
var_x
$f\circledcirc$
$elem_B$
ε_a
var_x
$\varepsilon_{(a,0)}$
$\varepsilon_{(a,1)}$
ε_b
some(2)

Figure 4.3 – Example execution of the simple LL(1) *parsing with derivatives* algorithm.

At the start, the state of the parser is represented by the initial expression var_x. The algorithm then starts processing the first `a` token. Since the kind of that token is part of the first set of the current expression, a derivative of that expression by `a` is computed, which results in a new parser state. The following tokens `a`, `b`, and `b`, are all processed in a similar fashion. Once the entire sequence has been processed, the nullability of the state is queried. In this case, the expression is indeed nullable, and therefore the algorithm simply returns the unique value returned by $\texttt{null}_\Gamma(e)$ as its result.

4.4.8 Complexity Analysis

In this last section of the chapter, I will use build on the example I have just shown to demonstrate that the simple LL(1) *parsing with derivatives* algorithm presented in this chapter is *not* worst-case linear-time. This complexity class would be expected of LL(1) parsing algorithms, as the traditional LL(1) parsing algorithm has worst-case linear time complexity (Aho et al., 2006). The argument that I will make can be transposed to the original *parsing with derivatives* algorithm (Might et al., 2011; Adams et al., 2016), from which one can conclude that the original *parsing with derivatives* algorithm is also not worst-case linear-time on LL(1) expressions.

Consider the environment Γ of the previous example, which maps the identifier x to the expression $(f \circledcirc ((elem_A \cdot var_x) \cdot elem_B)) \vee \varepsilon_0$. Consider the following sequence of expressions:

$$e_0 := var_x \qquad e_{i+1} := \delta^{\mathtt{a}}_{\Gamma}(e_i)$$

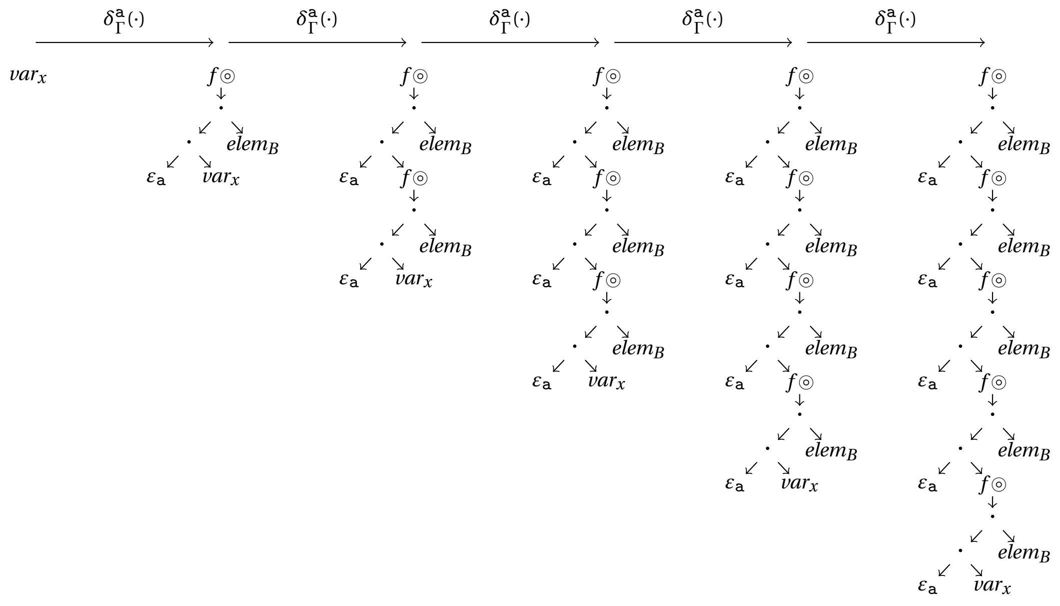
The expression e_i represents the state of the parsing algorithm after processing i `a` tokens.

In order to compute e_{i+1}, the LL(1) derivation function $\delta^{\mathtt{a}}_{\Gamma}(\cdot)$ must traverse all nodes until the leaf $elem_A$ node is found. During this traversal, the function will unfold the definition of x, making the resulting expression slightly larger and deeper than e_i. Figure 4.4 shows the successive parser states for the first few `a` tokens.

One can easily observe that the number of nodes that must be traversed until an $elem_A$ node can be found increases for each successive e_i. Since the derivation function always starts from the root of the expression, the time it takes to reach that $elem_A$ node and compute a derivative also increases with each successive parser state e_i.

Processing a sequence of n `a` tokens followed by n `b` tokens therefore takes at least time quadratic in n, even assuming all operations occurring after the last `a` token are free.

The subpar runtime complexity comes from the fact that derivation always starts at the root of the expression. With time, layers upon layers accumulate on top of the immediately interesting part of the expression. In the next chapter, I will present a simple solution to this issue: Instead of always starting from the root of the expression, the parsing algorithm should start the next derivation where the last ended. To do so, I will introduce a zipper-based (Huet, 1997) data



$\delta^{\mathtt{a}}_{\Gamma}(\cdot)$
var_x
$f\circledcirc$
$elem_B$
$\varepsilon_{\mathtt{a}}$

Figure 4.4 – Successive states of the parser after processing tokens of kind A.

structure to represent expressions. With this simple change of representation in place, I will show that the algorithm becomes worst-case linear time. I will also draw interesting parallels with the traditional LL(1) parsing algorithm.

5 Zippy LL(1) Parsing with Derivatives

In this chapter, I will build on the parsing algorithm described in the previous chapter and describe how a simple change to the data structure representing derived expressions yields an efficient linear-time parsing algorithm for LL(1) context-free expressions. So far, expressions have been represented in a rather direct way, with references from parent nodes to child nodes. In this setting, all references flowed from the root of the expression down to the leaves. This representation meant that, as derived expressions became increasingly deeper, derivation could become more and more expensive, as I showed with an example in Section 4.4.8.

The key insight to avoid this issue is to introduce a single *focus* in context-free expressions from which references flow, in the spirit of Huet's *zipper* data-structure (Huet, 1997). Nodes on the path from the root to the focal point will need to be given a representation that differs from the standard nodes. Indeed, such nodes will no longer have a reference to one of their children; instead it is the child node that will point to it. To represent such nodes, I will introduce the notion of *layers*. The path from the root of the expression to the focal point will be represented as a stack of such layers that I will call a *context*.

The derivation of a *focused LL(1) expression*, by a token t of kind k in its first set, will consist of a series of steps that progressively moves the focus towards the unique $elem_k$ leaf node that causes k to be a member of the first set of the expression. Such a node is guaranteed to be unique by the LL(1) property. The process of moving the focus down to this node will trim away unchosen branches along the way, such that not all expression nodes will need to be representable by layers. Once focused on this $elem_k$ node, derivation will be trivial, as it suffices to replace the node with an ε_t node. As all references at this point flow from the single node in focus, the rest of the structure can remain untouched.

While the LL(1) derivation function $\delta^t_\Gamma(\cdot)$ was defined in a purely downward (i.e. top-down) manner, the derivation function that I will present in this section will have both an upward and a downward phase. The downward phase of the algorithm will consist of a single recursive function called `pierce`. As I will show, that function will follow the exact same recursive structure as the $\delta^t_\Gamma(\cdot)$ function. Instead of directly building up a top-down expression to represent the result of derivation, that `pierce` function will build its result by adding layers to the context, and leave the focus at a leaf of the expression. The goal of the upward phase will be to traverse that stack of layers to locate a suitable continuation point for the downward phase.

The technique is reminiscent of the zipper-based technique I presented in Chapter 2. In that chapter, the zipper-based representation ensured that derivatives could only range over a finite set of predetermined values. In this chapter, due to the presence of variables, no such property can be obtained. As I shall demonstrate, the zipper-based representation of expression will play a different role: Employing a zipper will avoid the repeated traversals at the root of the quadratic behavior exhibited by the parsing algorithm presented in Chapter 4.

5.1 Zipper-based Representation of LL(1) Expressions

5.1.1 Layers

As explained in the introduction of this chapter, the key insight towards an efficient algorithm is to introduce a focus, a direct access, to a node in the expression. All references are to flow away from that focused node, such that this node becomes the new entry point to the data structure in place of the root of the expression. This change will require certain nodes, precisely those on the path from the root of the expression to the focused node, to be represented differently, as such nodes will have one less descendant. I introduce the *layers* to represent such altered nodes.

$$\frac{f \in T_1 \to T_2}{\texttt{apply}(f) \in \mathscr{L}^{T_1}_{T_2}} \qquad \frac{v : T_1}{\texttt{prepend}(v) \in \mathscr{L}^{T_2}_{(T_1,T_2)}} \qquad \frac{e \in \mathbf{CFE}_{T_2}}{\texttt{follow-by}(e) \in \mathscr{L}^{T_1}_{(T_1,T_2)}}$$

Figure 5.1 – Definition of layers.

Layers are parameterised by two types: an *exterior* type and an *interior* type. For a layer $l \in \mathscr{L}^{T_{in}}_{T_{ext}}$, the superscript T_{in} is the interior type, while the subscript T_{ext} is the exterior type. The interior type is the type ascribed to the subexpression below that node in the original expression, while the exterior type is the type ascribed to the subexpression rooted precisely at that node in the original expression.

The parents references are not explicitly present in the various layers, but will instead be given by the order in which they appear in the *context* stack. The layers are stored in the context from closest to the focal point to closest to the root of the expression, such that the parent of a layer node is always the next layer in the sequence.

Each layer corresponds to an expression with a hole denoted by $\square$:

- The `apply`(f) layer corresponds to the expression $f \circledcirc \square$.
- The `prepend`(v) layer corresponds to the expression $\varepsilon_v \cdot \square$.
- The `follow-by`(e) layer corresponds to the expression $\square \cdot e$.

Note that not all expressions constructs have corresponding layers. In particular, there are no layers for disjunctions, nor for variables. There are also no layers for sequences with a hole on the right and arbitrary expressions on the left.

In fact, one can observe that all expressions with holes that are given a corresponding layer appear in the definition of LL(1) derivation function $\delta^t_\Gamma(\cdot)$, with a recursive call to derivation function instead of the hole $\square$. This is not a coincidence. The derivation function that I will introduce in this section will, in its downward phase, follow the same recursive structure as $\delta^t_\Gamma(\cdot)$, but will materialise layers instead of building up a new expression.

5.1.2 Context

Layers are intended to be stacked, with the exterior type of a layer matching the interior type of the layer next in sequence. I will call such stacks of layers *contexts*. Contexts are formally defined in Figure 5.2.

$$\frac{}{\langle\rangle \in \mathscr{C}^T_T} \qquad\qquad \frac{l \in \mathscr{L}^{T_1}_{T_2} \qquad c \in \mathscr{C}^{T_2}_{T_3}}{l :: c \in \mathscr{C}^{T_1}_{T_3}}$$

Figure 5.2 – Definition of contexts.

Contexts $c \in \mathscr{C}^{T_{in}}_{T_{ext}}$, similarly to layers, are also parameterised by an interior type T_{in} and an exterior type T_{out}. Contexts a type-aligned stack of layers: the exterior-type of a layer always matches the interior-type of its successor in the stack, if any. In addition, for any non-empty context, the interior type of the context matches the interior type of its first element, while the exterior type of the context matches with the exterior type of its last element.

5.1.3 Weight of Layers and Contexts

Let each sort of layer be assigned a *weight*. Layers `apply`$(\cdot)$ and `prepend`$(\cdot)$ are assigned the weight 1, while `follow-by`$(\cdot)$ layers are assigned weight 2. The weight of a context is defined as the sum of the weights of its layers.

5.1.4 Focused Expressions

For any expression $e \in \mathbf{CFE}_{T_{in}}$ and context $c \in \mathscr{C}^{T_{in}}_{T_{ext}}$ of matching interior type T_{in}, I will call the pair (e, c) a *focused expression*. I will call the expression e the *focus*, or *focal point*, while c will simply be called the *context*.

Focusing

One can obtain a focused expression simply by associating an expression $e \in \mathbf{CFE}_T$ with the empty context $\langle\rangle \in \mathscr{C}^T_T$. I shall denote by $\texttt{focus}(e)$ such focused expression.

$$\texttt{focus}(e) := (e, \langle\rangle)$$

Note that the root of the expression is the only node that can be directly focused. In order to have a focus on a different part of the expression, the focus will need to be progressively moved there. The main goal of several of the operations that I will introduce later in this section will be to move the focus around.

Unfocusing

Focused expressions can be converted back to plain expressions using the `unfocus` function:

$$\begin{aligned}
\texttt{unfocus}((e, \langle\rangle)) &:= e \\
\texttt{unfocus}((e, \texttt{apply}(f) :: c')) &:= \texttt{unfocus}((f \circledcirc e, c')) \\
\texttt{unfocus}((e, \texttt{prepend}(v) :: c')) &:= \texttt{unfocus}((\varepsilon_v \cdot e, c')) \\
\texttt{unfocus}((e, \texttt{follow-by}(e') :: c')) &:= \texttt{unfocus}((e \cdot e', c'))
\end{aligned}$$

The `unfocus` function simply applies each layer of the stack recursively on top of the focal point. When the context is empty, the process is done. At that point, the focal point is at the root of the expression and the empty context can be discarded.

Note that this function will *not* be used by the parsing algorithm that I will develop it this section. It will however play an important role in arguments about the correctness of the algorithm.

5.1.5 LL(1) Property of Focused Expressions

Using `unfocus`, properties of (unfocused) expressions can be transposed to focused expressions. Among those is the LL(1) property:

$$\texttt{LL1}_\Gamma((e, c)) \iff \texttt{LL1}_\Gamma(\texttt{unfocus}((e, c))$$

Theorem 5.1 (LL(1) Preservation of `focus`$(\cdot)$)**.** *For any environment* Γ *and expression* e*, if* e *is LL(1), then so is the focused expression* $\texttt{focus}(e)$.

Proof. Trivial by the observation that $\texttt{unfocus}(\texttt{focus}(e)) = e$. □

Theorem 5.2 (LL(1) Focal Point)**.** *For any environment* Γ *and focused expression* (e, c)*, if* (e, c) *is LL(1), then the focal point* e *is also LL(1).*

Proof. By structural induction on the context c. □

5.1.6 The Essence of LL(1) Derivation

The zippy LL(1) derivation algorithm that I will introduce in this chapter relies on one crucial observation: If by any chance the focal point of a focused expression ends up being an $elem_k$ expression, then it suffices to swap that $elem_k$ with an ε_t expression for some token t of kind k to get a derivative of the original expression by the token t. This idea is formally stated in the following lemma.

Lemma 5.1 (Essential Derivation)**.** *For any environment* Γ*, token* t *of kind* k *and context* c*:*

$$(elem_k, c) \gg_{\Gamma}^{\langle t \rangle} (\varepsilon_t, c)$$

Proof. Straightforward by structural induction on the context c. □

From this observation, the goal of the game for the derivation function will now be to actually get the focal point to be an $elem_k$ node, and so while preserving the meaning of the expression. Towards this goal, I will now introduce operations that move or otherwise transform the focal point, while preserving semantics and the LL(1) property. I will first introduce simple movement and replacement operations, from which I will then be able to build more advanced operations such as `pierce`, `plug` and `locate`.

5.2 Zipper Operations

5.2.1 Focus Movement Operations

One of the essential aspects of focused expressions is that the focus can be moved around. Without this ability, the focus would always be on the same node, greatly limiting the usefulness of the technique. Fortunately, there are three basic operations that move the focus around. Those operations are bidirectional.

1. When the focus is on a $f \circledcirc e$ expression, the focus can be moved towards e by putting $\texttt{apply}(f)$ on the context stack. Similarly, when the context starts with an $\texttt{apply}(f)$ layer, the focus can be moved towards $f \circledcirc e$ by removing that layer from the stack.
2. When the focus is on $\varepsilon_v \cdot e$, the focus can be moved towards e by putting a $\texttt{prepend}(v)$ layer on top of the context stack. Likewise, when the context starts with a $\texttt{prepend}(v)$ layer, the focus can be moved to $\varepsilon_v \cdot e$ simply by removing that layer from the stack.
3. Finally, when the focus is on $e \cdot e'$, the focus can be moved to e by putting $\texttt{follow-by}(e')$ on top of the stack. Reversely, when the context starts with a $\texttt{follow-by}(e')$ layer, the focus can be moved to $e \cdot e'$ by removing that layer from the context.

One important property of those movement operations is that they preserve both the semantics of the expression as well as its LL(1) property. Indeed, both sides always are equal when viewed under $\texttt{unfocus}(\cdot)$, as schematised by Figure 5.3.

Figure 5.3 shows that, for all three focus movement operations, both sides are equal when viewed under $\texttt{unfocus}(\cdot)$. The bidirectional bold arrow represent the movement operations, while the solid arrows labeled by $\texttt{unfocus}(\cdot)$ represent the application of $\texttt{unfocus}(\cdot)$. The dashed bidirectional arrows represent the fact that the two unfocused expressions are related by a given relation, in this case =.

$$\begin{tikzcd}
u_0 \arrow[r, leftrightarrow, dashed, "="] & u_1 & u_0 \arrow[r, leftrightarrow, dashed, "="] & u_1 & u_0 \arrow[r, leftrightarrow, dashed, "="] & u_1 \\
(f \circledcirc e, c) \arrow[u, "\texttt{unfocus}(\cdot)"'] \arrow[r, leftrightarrow] & (e, \texttt{apply}(f) :: c) \arrow[u, "\texttt{unfocus}(\cdot)"'] & (\varepsilon_v \cdot e, c) \arrow[u, "\texttt{unfocus}(\cdot)"'] \arrow[r, leftrightarrow] & (e, \texttt{prepend}(v) :: c) \arrow[u, "\texttt{unfocus}(\cdot)"'] & (e \cdot e', c) \arrow[u, "\texttt{unfocus}(\cdot)"'] \arrow[r, leftrightarrow] & (e, \texttt{follow-by}(e') :: c) \arrow[u, "\texttt{unfocus}(\cdot)"']
\end{tikzcd}$$

Figure 5.3 – Available focus movement operations.

This chapter will showcase more similar looking schemas. In each case, the focused expressions will appear on the bottom row, while their unfocused counterparts will appear on the top row. The operations between focused expressions, as well as the relations between unfocused expressions, will differ.

Lemma 5.2 (Move Operations)**.** *For any expressions e, e', context c, function f and value v:*

$$\begin{aligned} \mathtt{unfocus}((f \circledcirc e, c)) &= \mathtt{unfocus}((e, \mathtt{apply}(f) :: c)) \\ \mathtt{unfocus}((\varepsilon_v \cdot e, c)) &= \mathtt{unfocus}((e, \mathtt{prepend}(v) :: c)) \\ \mathtt{unfocus}((e \cdot e', c)) &= \mathtt{unfocus}((e, \mathtt{follow\text{-}by}(e') :: c)) \end{aligned}$$

Proof. Straightforward by definition of $\mathtt{unfocus}(\cdot)$. □

5.2.2 Focus Replacement Operations

In addition to movement operations, focused expressions also support replacement operations. Given a focused expression (e_1, c), it is trivial to replace the focal point with a different expression e_2 of the appropriate type. To do so, one just has to pair the expression e_2 with the context c. The resulting focused expression is simply (e_2, c). This replacement operation is a crucial operation performed by the parsing algorithm presented in this section.

Obviously, not all replacements preserve the semantics and LL(1) property of the original expression. Leaving aside the semantics problem for a moment, consider the issue of preserving the LL(1) property. The following theorem identifies a condition under which the LL(1) property is preserved by focus replacement. The theorem states that the resulting expression will preserve the LL(1) property as long as the replacement expression is LL(1) in isolation, and that its should-not-follow set is a subset of the should-not-follow set of the expression it replaces.

Theorem 5.3 (LL(1) Preservation of Focus Replacement)**.** *For any environment Γ and LL(1) focused expression (e_1, c), and for any LL(1) expression e_2, if the should-not-follow set of e_2 is a subset of the should-not-follow set of e_1, then the focused expression (e_2, c) is also LL(1).*

Proof. Let Γ be an environment and c a context. The proof proceeds by structural induction on the context c. In both cases, let e_1 be a expression such that (e_1, c) is LL(1), and let e_2 be an LL(1) expression such that:

$$\text{SN-FOLLOW}_\Gamma(e_2) \subseteq \text{SN-FOLLOW}_\Gamma(e_1)$$

1. Consider the case when the context c is empty. Since $\mathtt{unfocus}((e_2, c)) = e_2$, which is LL(1) by assumption, the case is done.

2. Consider the case of a non-empty context c. In that case, $c = l :: c'$ for some layer l. The proof proceeds by case analysis on l.

(a) Consider the case where $l = \texttt{apply}(f)$ for some function f. In that case:

$$\texttt{unfocus}((e_1, c)) = \texttt{unfocus}((f \circledcirc e_1, c'))$$
$$\texttt{unfocus}((e_2, c)) = \texttt{unfocus}((f \circledcirc e_2, c'))$$

As $\texttt{unfocus}((e_1, c))$ is LL(1) by assumption, then so is $\texttt{unfocus}((f \circledcirc e_1, c'))$. Similarly, since e_2 is LL(1) by assumption, so is $f \circledcirc e_2$.

In addition, observe that the should-not-follow set of $f \circledcirc e_2$ is equal to the should-not-follow set of e_2, and therefore is also a subset of the should-not-follow set of e_1 and of $f \circledcirc e_1$.

Therefore the induction hypothesis, instantiated with $(f \circledcirc e_1, c')$ and $f \circledcirc e_2$, yields that:

$$\texttt{LL1}_\Gamma(\texttt{unfocus}((f \circledcirc e_2, c')))$$

And thus one can conclude that:

$$\texttt{LL1}_\Gamma(\texttt{unfocus}((e_2, c)))$$

(b) The case where $l = \texttt{prepend}(v)$ for some value v proceeds identically to the previous case.

(c) Consider the case where $l = \texttt{follow-by}(e')$ for some expression e'. In that case:

$$\texttt{unfocus}((e_1, c)) = \texttt{unfocus}((e_1 \cdot e', c'))$$
$$\texttt{unfocus}((e_2, c)) = \texttt{unfocus}((e_2 \cdot e', c'))$$

As $\texttt{unfocus}((e_1, c))$ is LL(1) by assumption, then so is $\texttt{unfocus}((e_1 \cdot e', c'))$.

It also must be the case that $e_2 \cdot e'$ is LL(1). The only possible conflicts are if:

- e_2 is not LL(1), which is prohibited by assumption.
- e' is not LL(1), which is also impossible as $(e_1 \cdot e', c')$ is LL(1), and therefore $e_1 \cdot e'$ as well as e' are also LL(1).
- The should-not-follow set of e_2 intersects with the first set of e'. This is also impossible. Indeed, by assumption the should-not-follow set of e_2 is a subset of the should-not-follow set of e_1, which is disjoint from the first set of e' by the fact that $e_1 \cdot e'$ is LL(1).

In addition, observe that the should-not-follow set of $e_2 \cdot e'$ is a subset of the should-not-follow set of $e_1 \cdot e'$.

Therefore the induction hypothesis, instantiated with $(e_1 \cdot e', c')$ and $e_2 \cdot e'$, yields that:

$$\texttt{LL1}_\Gamma(\texttt{unfocus}((e_2 \cdot e', c')))$$

And thus one can conclude that:

$$\texttt{LL1}_\Gamma(\texttt{unfocus}((e_2, c)))$$

□

Replacement Lemmas

Now that I have identified a condition under which the LL(1) property is preserved by replacement, let us focus back on the issue of preserving the semantics of the focused expression. The next lemma states that the focal point can be replaced by any equivalent expression without altering the semantics of the entire expression.

Lemma 5.3 (Equivalent Replacement)**.** *For any environment* Γ*, any context* c *and expressions* e_1*,* e_2*, when* $e_1 \equiv_\Gamma^{\langle\rangle} e_2$*, then:*

$$\mathit{unfocus}((e_1, c)) \equiv_\Gamma^{\langle\rangle} \mathit{unfocus}((e_2, c))$$

Proof. By structural induction on the context. □

The lemma that I have just shown has a very stringent condition: the new focal point must have the exact same meaning as the old focal point. In the case of variables and expressions in the environments, such a relation holds. The previous lemma implies that when focused on a var_x expression, it is possible to *unfold* that variable and replace it with the expression $\Gamma(x)$. The other direction also holds.

$$\begin{tikzcd}
u_0 \arrow[r, leftrightarrow, dashed, "\equiv_\Gamma^{\langle\rangle}"] & u_1 \\
(var_x, c) \arrow[u, "\mathtt{unfocus}(\cdot)"'] \arrow[r, leftrightarrow] & (\Gamma(x), c) \arrow[u, "\mathtt{unfocus}(\cdot)"']
\end{tikzcd}$$

Figure 5.4 – Replacement by unfolding.

Lemma 5.4 (Variable Elimination)**.** *For any environment* Γ*, any context* c *and identifier* x *in the domain of* Γ*:*

$$\mathit{unfocus}((var_x, c)) \equiv_\Gamma^{\langle\rangle} \mathit{unfocus}((\Gamma(x), c))$$

Proof. Immediate by Lemma 5.3. □

Unfortunately, most other replacements performed by the derivation function that I will present in this chapter do not satisfy the property that the replacement expression behaves exactly as the original focal point. Luckily, the result equivalence $\equiv_\Gamma^{\langle\rangle}$ of Lemma 5.3 is also often too strong for the purposes of the derivation function and parsing algorithm, and might be relaxed. A crucial observation is that, at the point when the next token t is known and fixed, the only interesting sequences in the semantics of an expression are those that start with the token t. All other sequences are irrelevant. Therefore, a weaker equivalence relation, such as $\equiv_\Gamma^{\langle t\rangle}$, is sufficient for the purposes of the derivation function. The following lemma offers more relaxed conditions in case of LL(1) expressions, with the caveat that the original and new expressions are only shown to be equivalent on token sequences starting with a given token t.

Given a token t, the following lemma states that one can replace the focal point of an LL(1) focused expression (e_1, c) by an other LL(1) expression e_2 that is equivalent for all token sequences starting with the token t and get a focused expression (e_2, c) that is also equivalent for all token sequences starting with t, provided that e_2 does not introduce new elements to the should-not-follow set, and that one of the following two conditions hold:

1. The kind k of the token t is part of the first set of e_1. Note that, since e_1 and e_2 behave equivalently on sequences starting by the token t, the kind k is also bound to be part of the first set of e_2.
2. The two expressions e_1 and e_2 behave the same on the empty sequence of values.

Lemma 5.5 (LL(1) Equivalent Replacement)**.** *For any environment* Γ*, any token* t *of kind* k*, any LL(1) focused expression* (e_1, c)*, and any LL(1) expression* e_2 *where* $e_1 \equiv_\Gamma^{\langle t\rangle} e_2$ *and* $\textsc{sn-follow}_\Gamma(e_2) \subseteq \textsc{sn-follow}_\Gamma(e_1)$*, if either*

$$k \in \textsc{first}_\Gamma(e_1) \qquad \textit{or} \qquad \forall v,\;\; e_1 \vdash_\Gamma \langle\rangle \rightsquigarrow v \iff e_2 \vdash_\Gamma \langle\rangle \rightsquigarrow v$$

Then:

$$\mathtt{unfocus}((e_1, c)) \equiv_\Gamma^{\langle t\rangle} \mathtt{unfocus}((e_2, c))$$

Proof. Let Γ be an environment and t a token of kind k. Let c be a context. The proof proceeds by structural induction on the context c. In both cases, let (e_1, c) be an LL(1) focused expression and e_2 be an LL(1) expression such that $e_1 \equiv_\Gamma^{\langle t\rangle} e_2$, $\textsc{sn-follow}_\Gamma(e_2) \subseteq \textsc{sn-follow}_\Gamma(e_1)$, and $k \in \textsc{first}_\Gamma(e_1) \vee \forall v, e_1 \vdash_\Gamma \langle\rangle \rightsquigarrow v \iff e_2 \vdash_\Gamma \langle\rangle \rightsquigarrow v$. Observe that, by Theorem 5.3, it is always the case that (e_2, c) is LL(1).

1. The case when c is empty trivially holds.
2. Consider the case when c is non-empty. In that case, $c = l :: c'$ for some layer l and context c'. The proof continues by case analysis on the layer l.

   (a) Consider the case when $l = \mathtt{apply}(f)$ for some function f. First, by definition of $\mathtt{unfocus}(\cdot)$:

   $$\mathtt{unfocus}((e_1, \mathtt{apply}(f) :: c')) \equiv_\Gamma^{\langle t\rangle} \mathtt{unfocus}((f \circledcirc e_1, c'))$$

   By induction hypothesis, given that the conditions are trivially satisfied:

   $$\mathtt{unfocus}((f \circledcirc e_1, c')) \equiv_\Gamma^{\langle t\rangle} \mathtt{unfocus}((f \circledcirc e_2, c'))$$

Finally, by definition of $\texttt{unfocus}(\cdot)$:

$$\texttt{unfocus}((f \circledcirc e_2, c')) \equiv_{\Gamma}^{\langle t \rangle} \texttt{unfocus}((e_2, \texttt{apply}(f) :: c'))$$

Transitivity of $\equiv_{\Gamma}^{\langle t \rangle}$ concludes this case.

(b) Consider the case when $l = \texttt{prepend}(v)$ for some value v. First, by definition of $\texttt{unfocus}(\cdot)$:

$$\texttt{unfocus}((e_1, \texttt{prepend}(v) :: c')) \equiv_{\Gamma}^{\langle t \rangle} \texttt{unfocus}((\varepsilon_v \cdot e_1, c'))$$

By induction hypothesis, given that the conditions are trivially satisfied:

$$\texttt{unfocus}((\varepsilon_v \cdot e_1, c')) \equiv_{\Gamma}^{\langle t \rangle} \texttt{unfocus}((\varepsilon_v \cdot e_2, c'))$$

Finally, by definition of $\texttt{unfocus}(\cdot)$:

$$\texttt{unfocus}((\varepsilon_v \cdot e_2, c')) \equiv_{\Gamma}^{\langle t \rangle} \texttt{unfocus}((e_2, \texttt{prepend}(v) :: c'))$$

Transitivity of $\equiv_{\Gamma}^{\langle t \rangle}$ concludes this case.

(c) Consider the case when $l = \texttt{follow-by}(e')$ for some expression e'. First, by definition of $\texttt{unfocus}(\cdot)$:

$$\texttt{unfocus}((e_1, \texttt{follow-by}(e') :: c')) \equiv_{\Gamma}^{\langle t \rangle} \texttt{unfocus}((e_1 \cdot e', c'))$$

In order to apply the induction hypothesis, I assert that $e_1 \cdot e' \equiv_{\Gamma}^{\langle t \rangle} e_2 \cdot e'$.

- For the first direction of the equivalence, assume there are a token sequence ts and a pair of values (v_1, v_2) such that $e_1 \cdot e' \vdash_{\Gamma} t :: ts \rightsquigarrow (v_1, v_2)$. Therefore, there must exist two token sequences ts_1 and ts_2 such that:

$$\begin{aligned} &e_1 \vdash_{\Gamma} ts_1 \rightsquigarrow v_1 \quad \wedge \\ &e' \vdash_{\Gamma} ts_2 \rightsquigarrow v_2 \end{aligned}$$

  When ts_1 is non-empty, it must start with token t, and therefore, by the assumption that $e_1 \equiv_{\Gamma}^{\langle t \rangle} e_2$:

$$e_2 \vdash_{\Gamma} ts_1 \rightsquigarrow v_1$$

  Therefore one gets that:

$$e_2 \cdot e' \vdash_{\Gamma} t :: ts \rightsquigarrow (v_1, v_2)$$

  Which concludes this direction, assuming ts_1 is non-empty.
  On the other hand when ts_1 is empty, we get, by the LL(1) property, that $k \not\in \textsc{first}_{\Gamma}(e_1)$. Therefore, by hypothesis it is the case that:

$$\forall v, e_1 \vdash_{\Gamma} \langle\rangle \rightsquigarrow v \iff e_2 \vdash_{\Gamma} \langle\rangle \rightsquigarrow v$$

And therefore that:

$$e_2 \cdot e' \vdash_\Gamma t :: ts \rightsquigarrow (v_1, v_2)$$

Which concludes this direction.

- For the second direction, assume there are a token sequence ts and a pair of values (v_1, v_2) such that $e_2 \cdot e' \vdash_\Gamma t :: ts \rightsquigarrow (v_1, v_2)$. Therefore, there must exist two token sequences ts_1 and ts_2 such that:

$$\begin{aligned} &e_2 \vdash_\Gamma ts_1 \rightsquigarrow v_1 \quad \wedge \\ &e' \vdash_\Gamma ts_2 \rightsquigarrow v_2 \end{aligned}$$

  When ts_1 is non-empty, it must start with token t, and therefore, by the hypothesis that $e_1 \equiv_\Gamma^{\langle t \rangle} e_2$:

$$e_1 \vdash_\Gamma ts_1 \rightsquigarrow v_1$$

  Thus, one gets that:

$$e_1 \cdot e' \vdash_\Gamma t :: ts \rightsquigarrow (v_1, v_2)$$

  Which concludes this direction, assuming ts_1 is non-empty.
  On the other hand when ts_1 is empty, we get, by the LL(1) property, that $k \notin \textsc{first}_\Gamma(e_2)$. Since $e_1 \equiv_\Gamma^{\langle t \rangle} e_2$, it must also be the case that $k \notin \textsc{first}_\Gamma(e_1)$. Therefore, it is the case that:

$$\forall v, e_1 \vdash_\Gamma \langle\rangle \rightsquigarrow v \iff e_2 \vdash_\Gamma \langle\rangle \rightsquigarrow v$$

  And therefore that:

$$e_2 \cdot e' \vdash_\Gamma t :: ts \rightsquigarrow (v_1, v_2)$$

  Which concludes this direction.

Having proven the non-trivial conditions, the proof then proceeds by invoking the induction hypothesis:

$$\texttt{unfocus}((e_1 \cdot e', c')) \equiv_\Gamma^{\langle t \rangle} \texttt{unfocus}((e_2 \cdot e', c'))$$

Finally, by definition of $\texttt{unfocus}(\cdot)$:

$$\texttt{unfocus}((e_2 \cdot e', c')) \equiv_\Gamma^{\langle t \rangle} \texttt{unfocus}((e_2, \texttt{follow-by}(e') :: c'))$$

Transitivity of $\equiv_\Gamma^{\langle t \rangle}$ concludes this case.

□

Elimination Lemmas

Although the previous lemma is very general, it is also arguably a bit abstract and convoluted. In this part, I introduce several useful and more concrete transformations whose correctness is a direct corollary of the previous lemma. An overview of the operations is presented in Figure 5.5, while the actual lemmas follow.

$$\begin{tikzcd} u_0 \arrow[r, leftrightarrow, dashed, "\equiv_\Gamma^{\langle t \rangle}"] & u_1 \\ (e, c) \arrow[u, "\texttt{unfocus}(\cdot)"] \arrow[r] & (\varepsilon_{\texttt{null}_\Gamma(e)}, c) \arrow[u, "\texttt{unfocus}(\cdot)"] \end{tikzcd}$$

$$\boxed{\textsc{nullable}_\Gamma(e) \text{ and } k \notin \textsc{first}_\Gamma(e)}$$

$$\begin{tikzcd} u_0 \arrow[r, leftrightarrow, dashed, "\equiv_\Gamma^{\langle t \rangle}"] & u_1 \\ (e_1 \cdot e_2, c) \arrow[u, "\texttt{unfocus}(\cdot)"] \arrow[r] & (\varepsilon_{\texttt{null}_\Gamma(e_1)} \cdot e_2, c) \arrow[u, "\texttt{unfocus}(\cdot)"] \end{tikzcd}$$

$$\boxed{\textsc{nullable}_\Gamma(e_1) \text{ and } k \notin \textsc{first}_\Gamma(e_1)}$$

$$\begin{tikzcd} u_0 \arrow[r, leftrightarrow, dashed, "\equiv_\Gamma^{\langle t \rangle}"] & u_1 \\ (e_1 \vee e_2, c) \arrow[u, "\texttt{unfocus}(\cdot)"] \arrow[r] & (e_1, c) \arrow[u, "\texttt{unfocus}(\cdot)"] \end{tikzcd}$$

$$\boxed{k \in \textsc{first}_\Gamma(e_1)}$$

$$\begin{tikzcd} u_0 \arrow[r, leftrightarrow, dashed, "\equiv_\Gamma^{\langle t \rangle}"] & u_1 \\ (e_1 \vee e_2, c) \arrow[u, "\texttt{unfocus}(\cdot)"] \arrow[r] & (e_2, c) \arrow[u, "\texttt{unfocus}(\cdot)"] \end{tikzcd}$$

$$\boxed{k \in \textsc{first}_\Gamma(e_2)}$$

Figure 5.5 – Elimination operations. In the figure, k represents the kind of the token t. All operations are subject to some preconditions, which have been highlighted. Additionally, those operations require that all involved expressions are LL(1).

Lemma 5.6 (Nullable Elimination). *For any environment* Γ, *LL(1) focused expression* (e, c), *and token* t *of kind* k, *when* $k \notin \textsc{first}_\Gamma(e)$ *and* $\textsc{nullable}_\Gamma(e)$, *then:*

$$unfocus((e, c)) \equiv_\Gamma^{\langle t \rangle} unfocus((\varepsilon_{null_\Gamma(e)}, c))$$

Proof. Immediate by Lemma 5.5. □

Lemma 5.7 (Left Disjunction Elimination). *For any environment* Γ, *LL(1) focused expression* $(e_1 \vee e_2, c)$, *and token* t *of kind* k, *when* $k \in \textsc{first}_\Gamma(e_1)$, *then:*

$$unfocus((e_1 \vee e_2, c)) \equiv_\Gamma^{\langle t \rangle} unfocus((e_1, c))$$

Proof. Immediate by Lemma 5.5. □

Lemma 5.8 (Right Disjunction Elimination)**.** *For any environment* Γ*, LL(1) focused expression* $(e_1 \vee e_2, c)$*, and token* t *of kind* k*, when* $k \in \textsc{first}_\Gamma(e_2)$*, then:*

$$\mathit{unfocus}((e_1 \vee e_2, c)) \equiv_\Gamma^{\langle t \rangle} \mathit{unfocus}((e_2, c))$$

Proof. Immediate by Lemma 5.5. □

Lemma 5.9 (Prefix Elimination)**.** *For any environment* Γ*, LL(1) focused expression* $(e_1 \cdot e_2, c)$*, and token* t *of kind* k*, when* $k \notin \textsc{first}_\Gamma(e_1)$ *and* $\textsc{nullable}_\Gamma(e_1)$*, then:*

$$\mathit{unfocus}((e_1 \cdot e_2, c)) \equiv_\Gamma^{\langle t \rangle} \mathit{unfocus}((\varepsilon_{\mathit{null}_\Gamma(e_1)} \cdot e_2, c))$$

Proof. Let Γ be an environment, t a token of kind k, and $(e_1 \cdot e_2, c)$ be an LL(1) focused expression. Assume $k \notin \textsc{first}_\Gamma(e_1)$ and $\textsc{nullable}_\Gamma(e_1)$.

By Lemma 5.2, the following holds:

$$\texttt{unfocus}((e_1 \cdot e_2, c)) \equiv_\Gamma^{\langle t \rangle} \texttt{unfocus}((e_1, \texttt{follow-by}(e_2) :: c))$$

Next, by Lemma 5.6, the following equivalence also holds:

$$\texttt{unfocus}((e_1, \texttt{follow-by}(e_2) :: c)) \equiv_\Gamma^{\langle t \rangle} \texttt{unfocus}((\varepsilon_{\texttt{null}_\Gamma(e_1)}, \texttt{follow-by}(e_2) :: c))$$

Finally, by Lemma 5.2 again:

$$\texttt{unfocus}((\varepsilon_{\texttt{null}_\Gamma(e_1)}, \texttt{follow-by}(e_2) :: c)) \equiv_\Gamma^{\langle t \rangle} \texttt{unfocus}((\varepsilon_{\texttt{null}_\Gamma(e_1)} \cdot e_2, c))$$

The proof concludes by transitivity of $\equiv_\Gamma^{\langle t \rangle}$. □

5.3 Zippy LL(1) Parsing with Derivatives

So far in this chapter, I have introduced the notion of LL(1) focused expressions, which are simply LL(1) expression with a focal point. I have also introduced several operations that move or replace the focal point while preserving the LL(1) property and the semantics (to some degree) of the original focused expression. Equipped with those operations, I will now be able to describe the LL(1) *parsing with derivatives and zippers* algorithm. I will start by presenting several higher-level operations that will play a crucial role in the zippy LL(1) derivation function $\zeta^k_\Gamma(\cdot)$ that is at the heart of the parsing algorithm.

5.3.1 Moving the Focus Downwards with `pierce`

The first high-level operation that I present is $\texttt{pierce}^k_\Gamma(\cdot,\cdot)$. The goal of the function is to move the focus all the way down the current focal point towards an $elem_k$ expression. Given a kind k and an LL(1) focused expression (e, c) where $k \in \textsc{first}_\Gamma(e)$, the function moves the focus towards the unique $elem_k$ node within e that caused k to be part of $\textsc{first}_\Gamma(e)$, unfolding variables, and eliminating prefixes and disjunctions along the way. Since the node in focus will always be an $elem_k$ node, the `pierce` function simply returns the context part of the resulting focused expression. The function is defined as follows:

$$\begin{aligned}
\texttt{pierce}^k_\Gamma(elem_k, c) &:= c \\
\texttt{pierce}^k_\Gamma(e_1 \vee e_2, c) &:= \begin{cases} \texttt{pierce}^k_\Gamma(e_1, c) & \text{if } k \in \textsc{first}_\Gamma(e_1) \\ \texttt{pierce}^k_\Gamma(e_2, c) & \text{otherwise} \end{cases} \\
\texttt{pierce}^k_\Gamma(e_1 \cdot e_2, c) &:= \begin{cases} \texttt{pierce}^k_\Gamma(e_1, \texttt{follow-by}(e_2) :: c) & \text{if } k \in \textsc{first}_\Gamma(e_1) \\ \texttt{pierce}^k_\Gamma(e_2, \texttt{prepend}(\texttt{null}_\Gamma(e_1)) :: c) & \text{otherwise} \end{cases} \\
\texttt{pierce}^k_\Gamma(f \circledcirc e, c) &:= \texttt{pierce}^k_\Gamma(e, \texttt{apply}(f) :: c) \\
\texttt{pierce}^k_\Gamma(var_x, c) &:= \texttt{pierce}^k_\Gamma(\Gamma(x), c)
\end{aligned}$$

Note that the definition assumes that the input focused expression (e, c) is LL(1) and that $k \in \textsc{first}_\Gamma(e)$. Since the definition is non-trivial, I will shortly prove that the function is well-defined, that is that:

1. the definition handles all constructs satisfying the preconditions,
2. that the preconditions of all functions invoked within the body of the definition are respected, and
3. that the computation terminates.

The three properties are proven in the following theorems.

Theorem 5.4 (Match-completeness)**.** *The definition of* $\mathtt{pierce}_{\Gamma}^{k}(\cdot,\cdot)$ *handles all constructs that can appear at the top of expressions* e *such that* $k \in \text{FIRST}_{\Gamma}(e)$.

Proof. The only two constructs which are not handled are ε_v and $\bot$, both of which have empty first sets. □

Theorem 5.5 (Preconditions)**.** *All arguments* e', c' *of recursive calls made in* $\mathtt{pierce}_{\Gamma}^{k}(e,c)$ *satisfy* $\text{LL1}_{\Gamma}((e',c'))$ *and* $k \in \text{FIRST}_{\Gamma}(e')$ *given that* $\text{LL1}_{\Gamma}((e,c))$ *and* $k \in \text{FIRST}_{\Gamma}(e)$. *Additionally, the preconditions of the calls to* $\mathtt{null}_{\Gamma}(\cdot)$, *that is that the argument expression is LL(1) and nullable, are also satisfied.*

Proof. By simple case analysis. □

Theorem 5.6 (Well-foundedness)**.** *The computation of* $\mathtt{pierce}_{\Gamma}^{k}(e,c)$ *terminates for any environment* Γ, *LL(1) focused expression* (e,c) *and kind* k *where* $k \in \text{FIRST}_{\Gamma}(e)$.

Proof. Follows directly by LL(1) induction (Lemma 4.1). □

The function can be seen as simply performing a series of focus movements and replacements, as defined in the previous section.

- When the focal point is already an $elem_k$ expression, $\mathtt{pierce}_{\Gamma}^{k}(\cdot,\cdot)$ simply returns the current context untouched.
- In case of a disjunction, $\mathtt{pierce}_{\Gamma}^{k}(\cdot,\cdot)$ simply eliminates the disjunction and only keeps the side which starts with k. The function proceeds recursively on that new focused expression.
- For sequences, either $\mathtt{pierce}_{\Gamma}^{k}(\cdot,\cdot)$ performs a prefix elimination followed by a move to focus e_2, or simply moves to focus e_1, depending on the side which contributed to $k \in \text{FIRST}_{\Gamma}(e_1 \cdot e_2)$. The function then proceeds recursively on that new focused expression.
- In case of $f \odot e'$, the focus is simply moved towards e', at which point the function proceeds recursively.
- Finally, in case of a variable var_x, the variable is unfolded and the function proceeds recursively on $\Gamma(x)$.

Since all those movements and replacements preserve the LL(1) property, the entire $\mathtt{pierce}_{\Gamma}^{k}(\cdot,\cdot)$ function also preserves the LL(1) property. Additionally, since all steps preserve $\equiv_{\Gamma}^{\langle t\rangle}$-equivalence for any token t of kind k, the focused expression $(elem_k, \mathtt{pierce}_{\Gamma}^{k}(e,c))$ will be equivalent to (e,c) for all tokens sequences starting with the token t. Those insights are formalised in the following two theorems.

Theorem 5.7 (LL(1) Preservation)**.** *For any environment* Γ*, kind* k*, LL(1) focused expression* (e, c) *with* $k \in \text{FIRST}_\Gamma(e)$*, the following holds:*

$$\text{LL1}_\Gamma((elem_k, \mathtt{pierce}^k_\Gamma(e, c)))$$

Proof. Let Γ be an environment and k a kind. The proof proceeds by Lemma 4.1. In each case, the original focused expression can be transformed into a focused expression where an induction hypothesis holds on the focal point, through a series of focus movements and focus replacement. Lemma 5.2 ensures that the LL(1) property is preserved by focus movements. For focus replacements, since the new focal point never introduces new elements to the should-not-follow set, Theorem 5.3 ensures that the LL(1) property is also preserved. □

Theorem 5.8 (Correctness)**.** *For any environment* Γ*, LL(1) focused expression* (e, c)*, token* t *of kind* $k \in \text{FIRST}_\Gamma(e)$*:*

$$\mathtt{unfocus}((e, c)) \equiv^{\langle t \rangle}_\Gamma \mathtt{unfocus}((elem_k, \mathtt{pierce}^k_\Gamma(e, c)))$$

Proof. Let Γ be an environment and t be a token of kind k. The proof proceeds by the LL(1) induction principle (Lemma 4.1) on LL(1) expressions e such that $k \in \text{FIRST}_\Gamma(e)$. In each case, let c be a context such that (e, c) is LL(1). Since $\mathtt{pierce}^k_\Gamma(\cdot, \cdot)$ only applies move operations (which preserve $\equiv^{\langle t \rangle}_\Gamma$-equivalence by Lemma 5.2) and various replacement operations (which preserve $\equiv^{\langle t \rangle}_\Gamma$-equivalence by Lemma 5.4, Lemma 5.7, Lemma 5.8, and Lemma 5.9), the expression (e, c) is always $\equiv^{\langle t \rangle}_\Gamma$-equivalent to the focused expression passed as argument to the recursive call to $\mathtt{pierce}^k_\Gamma(\cdot, \cdot)$, and on which an induction hypothesis is available.

□

Similarities with Simple (Non-Zippy) LL(1) Derivation

Notice that the definition of `pierce` is almost identical to the definition of (non-zippy) LL(1) derivation $\delta^t_\Gamma(\cdot)$. The recursive structure of the two definitions are identical. There are however a few key differences in the definitions:

- Instead of directly building the resulting expression as is the case for $\delta^t_\Gamma(\cdot)$, `pierce` accumulate layers on top of an existing context passed as argument. Contrary to $\delta^t_\Gamma(\cdot)$, the function `pierce` is *tail recursive*, meaning that recursive calls are always the last action performed within the body of the function. Such tail recursive functions can be implemented more efficiently depending on the implementation language.
- Although $\delta^t_\Gamma(\cdot)$ had to directly replace the leaf $elem_k$ node directly with an ε_t node in order to return from the last recursive call, the function `pierce` does not need to perform that replacement. The function `pierce` simply returns the context around that

> *elem*$_k$ node. With the focus on that node, it is trivial to replace it by an ε_t node, but it is also possible to easily refer to the expression with the original *elem*$_k$ node still in place.
>
> For this reason, `pierce` also does not need the actual token t as an argument, only its kind k. This fact will prove useful when I investigate caching opportunities later on.

5.3.2 Moving the Focus Upwards

In the previous few paragraphs, I have presented the $\texttt{pierce}^k_\Gamma(\cdot,\cdot)$ function, which, when applied on an LL(1) focused expression (e,c), where $k \in \textsc{first}_\Gamma(e)$, moves the focus down the expression towards the unique *elem*$_k$ node that caused $k \in \textsc{first}_\Gamma(e)$.

As discussed, $\texttt{pierce}^k_\Gamma(\cdot,\cdot)$ requires k to be in the first set of the focal point e. Unfortunately, it is not always the case that k is in the first set of the focal point when k is part of the first set of the expression.

$$k \in \textsc{first}_\Gamma(\texttt{unfocus}((e,c))) \;\not\Longrightarrow\; k \in \textsc{first}_\Gamma(e)$$

Indeed, when e is nullable, the context c may also contribute to the first set of $\texttt{unfocus}((e,c))$, and therefore the *elem*$_k$ to be focused might not be found within e, but within an expression somewhere in the context c. To resolve this issue, I will now introduce the function $\texttt{locate}^k_\Gamma(\cdot)$, as well as the helper function $\texttt{plug}(\cdot,\cdot)$.

The goal of $\texttt{locate}^k_\Gamma(\cdot)$ is to move the focus up the context towards an expression which starts with the desired kind k. As such an expression may not always exist, the function may return `none`. To perform this task, the $\texttt{locate}^k_\Gamma(\cdot)$ function makes use of the helper function $\texttt{plug}(\cdot,\cdot)$, which I shall introduce first.

5.3.3 The `plug` function

The function $\texttt{plug}(\cdot,\cdot)$ is parameterised by a value v and a context c. The goal of the function is to move the focus towards the next expression in the context c when the current focal point reduces down to the value v. The function is defined as follows:

$$\begin{aligned}
\texttt{plug}(v,\langle\rangle) &:= (\varepsilon_v,\langle\rangle)\\
\texttt{plug}(v,\texttt{apply}(f) :: c') &:= \texttt{plug}(f(v),c')\\
\texttt{plug}(v,\texttt{prepend}(v') :: c') &:= \texttt{plug}((v',v),c')\\
\texttt{plug}(v,\texttt{follow-by}(e) :: c') &:= (e,\texttt{prepend}(v) :: c')
\end{aligned}$$

When the context c is empty, $\texttt{plug}(v,c)$ simply returns the focused expression $(\varepsilon_v,\langle\rangle)$. In the case of an $\texttt{apply}(f)$ or a $\texttt{prepend}(v')$ layer at the top of the context, $\texttt{plug}(v,c)$ respectively applies f to v or pairs v' with v before proceeding recursively on the rest of the context. Finally, in case of a $\texttt{follow-by}(e)$ layer at the top of the context, $\texttt{plug}(v,c)$ stops and returns a focused expression where e is the focal point. In that case, the top `follow-by`(e) layer is replaced by a `prepend`(v) layer. That `prepend`(v) layer keeps track of the value that was plugged.

Theorem 5.9 (LL(1) Preservation). *When (ε_v, c) is LL(1), the focused expression returned by $plug(v, c)$ is also LL(1).*

Proof. By structural induction on the context c. □

Theorem 5.10 (Correctness). *The focused expression returned by $plug(v, c)$ is equivalent to (ε_v, c).*

$$\forall \Gamma, c, v. \quad unfocus(plug(v, c)) \equiv_\Gamma unfocus((\varepsilon_v, c))$$

Proof. By structural induction on the context c. □

Theorem 5.11 (Progress). *When c is non-empty, the context of the focused expression returned by $plug(v, c)$ has a strictly smaller weight than c.*

Proof. By structural induction on the context c. □

5.3.4 The `locate` function

The next function I introduce is $\texttt{locate}^k_\Gamma(\cdot)$. The goal of $\texttt{locate}^k_\Gamma(\cdot)$ is to move the focus up the context until the focal point starts with the desired kind k. To move the focus upwards, the function makes use of the $\texttt{plug}(\cdot,\cdot)$ that was just discussed. The function is defined as follows:

$$\texttt{locate}^k_\Gamma((e, c)) := \begin{cases} \texttt{some}((e, c)) & \text{if } k \in \textsc{first}_\Gamma(e) \\ \texttt{none} & \text{if } \neg\textsc{nullable}_\Gamma(e) \text{ or } c = \langle\rangle \\ \texttt{locate}^k_\Gamma(\texttt{plug}(\texttt{null}_\Gamma(e), c)) & \text{otherwise} \end{cases}$$

In case the focal point already contains k in the first set, $\texttt{locate}^k_\Gamma((e, c))$ returns the focused expression (e, c) untouched. Otherwise, when the focal point e is not nullable, or when the context c is empty, $\texttt{locate}^k_\Gamma((e, c))$ returns `none`. In that case, the expression $\texttt{unfocus}((e, c))$ is guaranteed not to have k in its first set. In the opposite case, when both e is nullable and the context is not empty, the unique value associated with the empty sequence of tokens by e is retrieved, and `plug` is used to move the focus to the next expression in the context.

Theorem 5.12 (Well-foundedness). *The computation of $locate^k_\Gamma((e, c))$ terminates for all environments Γ, kind k and LL(1) focused expression (e, c).*

Proof. For all recursive calls, the weight of the context of the focused expression argument is always strictly decreasing. Indeed, by Theorem 5.11, the weight of the context of $\texttt{plug}(v, c)$ is strictly smaller than the weight of c, and so for any v and c. □

Theorem 5.13 (First Location)**.** *When $\mathtt{locate}^k_\Gamma((e,c)) = \mathtt{some}((e',c'))$, the focal point e' contains the kind k in its first set.*

Proof. By strong induction on the weight of the context c. The recursive case makes use of Theorem 5.11 to argue that the recursive context argument is smaller. □

Theorem 5.14 (LL(1) Preservation)**.** *When (e,c) is LL(1) and $\mathtt{locate}^k_\Gamma((e,c)) = \mathtt{some}((e',c'))$, then (e',c') is also LL(1).*

Proof. Let Γ be an environment and k a kind. The proof proceeds by strong induction on the weight of the context c. In both cases, let (e,c) be an LL(1) focused expression and assume $\mathtt{locate}^k_\Gamma((e,c)) = \mathtt{some}((e',c'))$.

1. Consider the case when the weight of the context c is zero. In that case, the context c is bound to be empty. Thus, for $\mathtt{locate}^k_\Gamma((e,c))$ to not be $\mathtt{none}$, it must be the case that $k \in \textsc{first}_\Gamma(e)$ and thus that $\mathtt{locate}^k_\Gamma((e,c)) = \mathtt{some}((e,c))$. Therefore in this case $(e',c') = (e,c)$. Since by assumption (e,c) is LL(1), then so is (e',c').

2. Consider the case when the weight is non-zero. Since the weight is non-zero, the context c is non-empty. The only interesting case is when k is not part of the first set of e and e is nullable.

   In that case, the call to $\mathtt{locate}^k_\Gamma((e,c))$ reduces to $\mathtt{locate}^k_\Gamma(\mathtt{plug}(\mathtt{null}_\Gamma(e),c))$. By Theorem 5.3, the focused expression $(\varepsilon_{\mathtt{null}_\Gamma(e)},c)$ is LL(1). Additionally, by Theorem 5.9, the focused expression $\mathtt{plug}(\mathtt{null}_\Gamma(e),c)$ is also LL(1). Finally, by Theorem 5.11, the context of the focused expression $\mathtt{plug}(\mathtt{null}_\Gamma(e),c)$ has a strictly smaller weight than c. Therefore the induction hypothesis applies and the proof concludes.

□

Theorem 5.15 (Soundness)**.** *For any environment Γ, LL(1) focused expression (e,c) and token t of kind k, when $\mathtt{locate}^k_\Gamma((e,c)) = \mathtt{some}((e',c'))$, then:*

$$\mathtt{unfocus}((e,c)) \equiv^{\langle t\rangle}_\Gamma \mathtt{unfocus}((e',c'))$$

Proof. Let Γ be an environment, t a token of kind k and c a context. The proof proceeds by strong induction on the weight of the context c. In each case, let e by an expression such that (e,c) is LL(1) and $\mathtt{locate}^k_\Gamma((e,c)) = \mathtt{some}((e',c'))$ for some (e',c').

1. The case when the weight of c is zero is trivial.

2. Consider the case when the weight of c is non-zero, that is when the context c is non-empty. The only interesting case is when k is not part of the first set of e and e is nullable.

In that case, the call to $\texttt{locate}^k_\Gamma((e,c))$ reduces to $\texttt{locate}^k_\Gamma(\texttt{plug}(\texttt{null}_\Gamma(e), c))$.

Notice that, by Lemma 5.6, the following equivalence holds:

$$\texttt{unfocus}((e,c)) \equiv^{\langle t \rangle}_\Gamma \texttt{unfocus}((\varepsilon_{\texttt{null}_\Gamma(e)}, c))$$

Additionally, by Theorem 5.10:

$$\texttt{unfocus}((\varepsilon_{\texttt{null}_\Gamma(e)}, c)) \equiv^{\langle t \rangle}_\Gamma \texttt{unfocus}(\texttt{plug}(\texttt{null}_\Gamma(e), c))$$

Finally, by induction hypothesis, since the weight is decreasing by Theorem 5.11:

$$\texttt{unfocus}(\texttt{plug}(\texttt{null}_\Gamma(e), c)) \equiv^{\langle t \rangle}_\Gamma \texttt{unfocus}((e', c'))$$

Therefore, by transitivity:

$$\texttt{unfocus}((e,c)) \equiv^{\langle t \rangle}_\Gamma \texttt{unfocus}((e', c'))$$

□

Lemma 5.10 (Core Failure)**.** *For any environment Γ, any token t of kind k, and any focused expression (e,c), when $k \notin \text{FIRST}_\Gamma(e)$ and $\neg\text{NULLABLE}_\Gamma(e)$, then:*

$$k \notin \text{FIRST}_\Gamma(\textit{unfocus}((e,c)))$$

Proof. By structural induction on the context. □

Theorem 5.16 (Completeness)**.** *For any environment Γ, LL(1) focused expression (e,c) and token t of kind k, when $\textit{locate}^k_\Gamma((e,c)) = \textit{none}$, then $k \notin \text{FIRST}_\Gamma(\textit{unfocus}((e,c)))$.*

Proof. Let Γ be an environment, t a token of kind k, and c a context. The proof proceeds by strong induction on the weight of the context c. In each case, let e by an expression such that (e,c) is LL(1) and $\texttt{locate}^k_\Gamma((e,c)) = \texttt{none}$.

1. The case when the weight of c is zero, that is when the context is empty, is trivial.
2. Consider the case when the context c is non-empty. As $k \in \text{FIRST}_\Gamma(e)$ leads to a contradiction, it must be the case that $k \notin \text{FIRST}_\Gamma(e)$. Additionally, when e is not nullable, Lemma 5.10 applies and immediately concludes the case.

   The only remaining case is when k is not part of the first set of e and when e is nullable.

   In that case, the call to $\texttt{locate}^k_\Gamma((e,c))$ reduces to $\texttt{locate}^k_\Gamma(\texttt{plug}(\texttt{null}_\Gamma(e), c))$, which is therefore also bound to be $\texttt{none}$.

   By induction hypothesis, since the weight is decreasing by Theorem 5.11:

$$k \notin \text{FIRST}_\Gamma(\texttt{unfocus}(\texttt{plug}(\texttt{null}_\Gamma(e), c)))$$

Next, by Theorem 5.10, it must be the case that:

$$k \notin \text{FIRST}_\Gamma(\texttt{unfocus}((\varepsilon_{\texttt{null}_\Gamma(e)}, c)))$$

Finally, by Lemma 5.6:

$$k \notin \text{FIRST}_\Gamma(\texttt{unfocus}((e, c)))$$

□

5.3.5 Zippy LL(1) Derivation

Now that I have defined the $\texttt{locate}^k_\Gamma(\cdot)$ and $\texttt{pierce}^k_\Gamma(\cdot,\cdot)$ helper functions, I can finally define the zippy LL(1) derivation function $\zeta^t_\Gamma(\cdot)$. The function assumes that the argument focused expression (e, c) is LL(1). It is defined as follows:

$$\zeta^t_\Gamma((e,c)) := \begin{cases} \texttt{some}((\varepsilon_t, \texttt{pierce}^k_\Gamma(e', c')) & \text{if } \texttt{locate}^k_\Gamma((e,c)) = \texttt{some}((e', c')), k = \texttt{kind}(t) \\ \texttt{none} & \text{otherwise} \end{cases}$$

Theorem 5.17 (Preconditions)**.** *All preconditions of the call to* $\texttt{pierce}^k_\Gamma(e', c')$ *are satisfied.*

Proof. The preconditions are that (e', c') is LL(1) and $k \in \text{FIRST}_\Gamma(e')$. The first precondition is ensured by Theorem 5.14, the second by Theorem 5.13. □

The zippy LL(1) function makes use of the two helper functions $\texttt{locate}^k_\Gamma(\cdot)$ and $\texttt{pierce}^k_\Gamma(\cdot,\cdot)$. Figure 5.6 presents a visualisation of the algorithm, and showcases how the various intermediate expressions are related.

$$\begin{tikzcd}
u_0 \arrow[rr, leftrightarrow, dashed, "\equiv^{\langle t \rangle}_\Gamma"'] \arrow[rrrrrr, leftrightarrow, dashed, bend left, "\gg^{\langle t \rangle}_\Gamma"] & & u_1 \arrow[rr, leftrightarrow, dashed, "\equiv^{\langle t \rangle}_\Gamma"'] & & u_2 \arrow[rr, leftrightarrow, dashed, "\gg^{\langle t \rangle}_\Gamma"'] & & u_3 \\
(e,c) \arrow[u, "\texttt{unfocus}(\cdot)"'] \arrow[rr, "\texttt{locate}^k_\Gamma(\cdot)"] & {} \arrow[d] & (e',c') \arrow[u, "\texttt{unfocus}(\cdot)"'] \arrow[rr, "\texttt{pierce}^k_\Gamma(\cdot,\cdot)"] & & (elem_k, c_p) \arrow[u, "\texttt{unfocus}(\cdot)"'] \arrow[rr, "\text{replacement}"] & & (\varepsilon_t, c_p) \arrow[u, "\texttt{unfocus}(\cdot)"'] \\
& \texttt{none} & & & c_p = \texttt{pierce}^k_\Gamma(e_1, c_1) & &
\end{tikzcd}$$

Figure 5.6 – Visualisation of the zippy LL(1) derivation function $\zeta^t_\Gamma(\cdot)$.

It is worth thinking of the zippy LL(1) derivation function as working in three distinct phases, represented by bold arrows in Figure 5.6:

1. In the first phase, $\texttt{locate}^k_\Gamma((e,c))$ is used to move the focus up towards a point (e', c') such that $k \in \textsc{first}_\Gamma(e')$. Note that such a focal point may not always exist, and for this reason $\texttt{locate}^k_\Gamma((e,c))$ may also return `none`. If it is the case, then the zippy LL(1) derivation function also returns `none`. When it actually leads to a new focused expression, the phase is guaranteed to preserve the LL(1) property and $\equiv^{\langle t\rangle}_\Gamma$-equivalence.
2. In the second phase, the focus is moved down towards the unique $elem_k$ node that caused $k \in \textsc{first}_\Gamma(e')$. The context around that $elem_k$ is returned by $\texttt{pierce}^k_\Gamma(e', c')$. Again, the phase preserves the LL(1) property and $\equiv^{\langle t\rangle}_\Gamma$-equivalence.
3. In the last phase, the focal point $elem_k$ is replaced by an ε_t expression. That phase actually results in a focused expression that is derivative of the initial focused expression by the token t. The LL(1) property is still maintained.

Note that although logically distinct, the last two phases are merged in the actual definition. Indeed, with the present definition of $\zeta^t_\Gamma(\cdot)$, the $(elem_k, \texttt{pierce}^k_\Gamma(e', c'))$ focused expression is not materialised. However, keeping the two phases logically distinct simplifies reasoning.

The following theorems state the LL(1) preservation property of zippy LL(1) derivation, as well as its correctness with respect to the semantic relation $\gg^{\langle t\rangle}_\Gamma$.

Theorem 5.18 (LL(1) Preservation)**.** *For any environment* Γ*, token* t *and LL(1) focused expression* (e,c)*, if it is the case that* $\zeta^t_\Gamma((e,c)) = \mathit{some}((e',c'))$*, then* (e',c') *is also LL(1).*

Proof. Straightforward by Theorem 5.7 and Theorem 5.3. □

Theorem 5.19 (Soundness)**.** *For any environment* Γ*, token* t *and LL(1) focused expression* (e,c)*, if it is the case that* $\zeta^t_\Gamma((e,c)) = \mathit{some}((e',c'))$*, then:*

$$\mathit{unfocus}((e,c)) \gg^{\langle t\rangle}_\Gamma \mathit{unfocus}((e',c'))$$

Proof. Immediate by Theorem 5.15, Theorem 5.8, Lemma 5.1, as well as Theorem 3.8. □

Theorem 5.20 (Completeness)**.** *For any environment* Γ*, token* t *and LL(1) focused expression* (e,c)*, if it is the case that* $\zeta^t_\Gamma((e,c)) = \mathit{none}$*, then:*

$$k \notin \textsc{first}_\Gamma((e,c))$$

Proof. Immediate by Theorem 5.16. □

5.3.6 The `result` function

The last function that I need to introduce before the zippy LL(1) parsing algorithm is the $\texttt{result}_\Gamma(\cdot)$ function. Given an LL(1) focused expression (e, c), the goal of this function is to return the unique value associated with the empty string by the expression (e, c), if any.

$$\texttt{result}_\Gamma((e,c)) := \begin{cases} \texttt{none} & \text{if } \neg\text{NULLABLE}_\Gamma(e) \\ \texttt{some}(\texttt{null}_\Gamma(e)) & \text{if } c = \langle\rangle \\ \texttt{result}_\Gamma(\texttt{plug}(\texttt{null}_\Gamma(e), c)) & \text{otherwise} \end{cases}$$

Theorem 5.21 (Well-foundedness)**.** *The computation of $\texttt{result}_\Gamma((e,c))$ terminates for all environments Γ and LL(1) focused expression (e,c).*

Proof. By Theorem 5.11, the weight of the context of $\texttt{plug}(\texttt{null}_\Gamma(e), c)$ is strictly smaller than the weight of c, thereby showing termination. □

Theorem 5.22 (Soundness)**.** *For any environment Γ, and LL(1) focused expression (e,c), if $\texttt{result}_\Gamma((e,c)) = \texttt{some}(v)$, then:*

$$\texttt{unfocus}((e,c)) \vdash_\Gamma \langle\rangle \rightsquigarrow v$$

Proof. By induction on the context c. □

Theorem 5.23 (Completeness)**.** *For any environment Γ, and LL(1) focused expression (e,c), if $\texttt{result}_\Gamma((e,c)) = \texttt{none}$, then:*

$$\neg\text{NULLABLE}_\Gamma(\texttt{unfocus}((e,c)))$$

Proof. By induction on the context c. □

5.3.7 Zippy LL(1) Parsing with Derivatives Algorithm

Now is finally the time to introduce the *zippy LL(1) parsing with derivatives* algorithm. Given an LL(1) focused expression and a sequence of tokens as inputs, the algorithm returns the unique value associated with the sequence by the expression, if any.

$$\texttt{zippy-ll1-parse}_\Gamma((e,c), \langle\rangle) := \texttt{result}_\Gamma((e,c))$$

$$\texttt{zippy-ll1-parse}_\Gamma((e,c), t :: ts) := \begin{cases} \texttt{zippy-ll1-parse}_\Gamma((e',c'), ts) & \text{if } \zeta^t_\Gamma((e,c)) = \texttt{some}((e',c')) \\ \texttt{none} & \text{otherwise} \end{cases}$$

The algorithm repeatedly applies zippy LL(1) derivation $\zeta^{t}_{\Gamma}(\cdot)$ for each input token t, and computes the final result using the $\texttt{result}_{\Gamma}(\cdot)$ function. If at any point derivation fails and returns `none`, the entire algorithm stops and returns `none`.

Theorem 5.24 (Preconditions)**.** *All preconditions to the recursive call to* $\textit{zippy-ll1-parse}_{\Gamma}((e', c'), ts)$ *are satisfied.*

Proof. The only precondition is that (e', c') is LL(1). The precondition is verified by Theorem 5.18. □

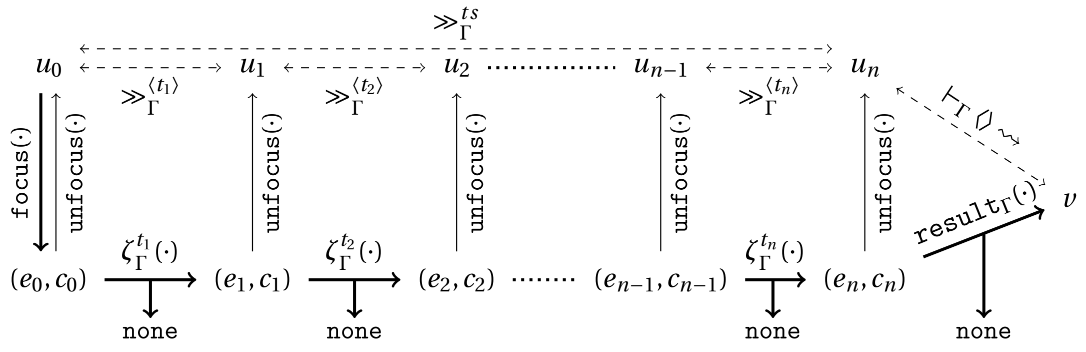
5.3.8 On the Correctness of Zippy LL(1) Parsing with Derivatives

In this section, I will investigate the correctness of the *zippy LL(1) parsing with derivatives* algorithm. The correctness statement of the algorithm is expressed in terms of the semantic relation $e \vdash_{\Gamma} ts \rightsquigarrow v$:

$$\forall \Gamma, e. \quad \texttt{LL1}_{\Gamma}(e) \implies \forall ts, v. \quad \texttt{zippy-ll1-parse}_{\Gamma}(\texttt{focus}(e), ts) = \texttt{some}(v) \iff e \vdash_{\Gamma} ts \rightsquigarrow v$$

For it to be correct, given an environment Γ, an LL(1) expression e and a sequence of tokens ts, the *zippy LL(1) parsing with derivatives* algorithm must successfully return a value v if and only if the expression associates the token sequence ts with the value v according to the semantics relation.

The proof of correctness of the *zippy LL(1) parsing with derivatives* algorithm is composed of several parts, each of which reasoning about a specific part of the algorithm. In order to facilitate the understanding of the argument and its various parts, Figure 5.7 presents an overview of the correctness argument and how the various theorems presented in this section are used. An explanation of the figure follows.



$\gg^{ts}_{\Gamma}$
u_0
u_1
u_2
u_{n-1}
u_n
$\gg^{\langle t_1\rangle}_{\Gamma}$
$\gg^{\langle t_2\rangle}_{\Gamma}$
$\gg^{\langle t_n\rangle}_{\Gamma}$
$\vdash_{\Gamma} \emptyset \rightsquigarrow$
focus(·)
unfocus(·)
$\zeta^{t_1}_{\Gamma}(\cdot)$
$\zeta^{t_2}_{\Gamma}(\cdot)$
$\zeta^{t_n}_{\Gamma}(\cdot)$
$\texttt{result}_{\Gamma}(\cdot)$
v
(e_0, c_0)
(e_1, c_1)
(e_2, c_2)
(e_{n-1}, c_{n-1})
(e_n, c_n)
none

Figure 5.7 – Visualisation of the *zippy LL(1) parsing with derivatives algorithm.*

The bolder arrows show the path taken by the algorithm: First, the starting LL(1) expression u_0 is turned into a focused expression (e_0, c_0) using $\texttt{focus}(\cdot)$. Then, zippy LL(1) derivation $\zeta^{t}_{\Gamma}(\cdot)$

is repeatedly applied, once for each input token. Finally, the $\texttt{result}_\Gamma(\cdot)$ function is used to return the value associated with the empty string. At every derivation step, as well as in the $\texttt{result}_\Gamma(\cdot)$ step, the process can stop and return `none`.

The solid arrows (labeled by $\texttt{unfocus}(\cdot)$) link each focused expression (e_i, c_i) to its unfocused counterpart u_i. The unfocused counterparts are not actually computed by the algorithm, but are useful to reason about the meaning of focused expressions.

Dashed arrows express relations. The dashed arrow between an unfocused expression u_i and its successor u_{i+1} states that $u_i \gg_\Gamma^{\langle t \rangle} u_{i+1}$, that is that u_{i+1} is a derivative of u_i by the token t_{i+1}. The arrow that relates u_0 to u_n states that $u_0 \gg_\Gamma^{ts} u_n$, that is that u_n is a derivative of u_0 by the sequence of tokens ts. This relation follows directly by transitivity of $\gg_\Gamma^{\cdot}$.Finally, the dashed arrow between u_n and v indicates that $u_n \vdash_\Gamma \langle\rangle \rightsquigarrow v$. Together, those relations state the soundness of the algorithm.

In order to argue the completeness of the algorithm, I will show that whenever `none` is returned by derivation of u_i by t_{i+1}, then there are no *productive* expressions u_{i+1} such that $u_i \gg_\Gamma^{\langle t \rangle} u_{i+1}$. I will also need to show that whenever $\texttt{result}_\Gamma((e_n, c_n))$ return `none`, then $\neg\textsc{nullable}_\Gamma(u_n)$.

Theorem 5.25 (Soundness)**.** *For any environment* Γ*, LL(1) focused expression* (e, c) *and token sequence* ts*, whenever:*

$$\textit{zippy-ll1-parse}_\Gamma((e,c), ts) = \textit{some}(v)$$

Then:

$$\textit{unfocus}((e,c)) \vdash_\Gamma ts \rightsquigarrow v$$

Proof. Let Γ be an environment, (e, c) be an LL(1) focused expression, ts a sequence of tokens. The proof proceeds by structural induction on the list of tokens.

1. In the case the list of tokens is empty, $\texttt{zippy-ll1-parse}_\Gamma((e,c), \langle\rangle)$ reduces to $\texttt{result}_\Gamma((e,c))$. By assumption, $\texttt{result}_\Gamma((e,c)) = \texttt{some}(v)$. Finally, Theorem 5.22 ensures that $\texttt{unfocus}((e,c) \vdash_\Gamma \langle\rangle \rightsquigarrow v$, which concludes the case.

2. Consider the case when the list of tokens is non-empty. Let t denote the first token of the list and ts' the rest.

   In this case, $\texttt{zippy-ll1-parse}_\Gamma((e,c), t :: ts')$, by assumption, reduces to $\texttt{zippy-ll1-parse}_\Gamma((e',c'), ts)$ from some focused expression (e', c') where $\zeta_\Gamma^t((e,c)) = \texttt{some}((e',c'))$.

   Again by assumption, it is the case that:

$$\texttt{zippy-ll1-parse}_\Gamma((e',c'), ts) = \texttt{some}(v)$$

Therefore, by applying the induction hypothesis, one gets that:

$$\texttt{unfocus}((e', c') \vdash_\Gamma ts' \rightsquigarrow v$$

Additionally, from Theorem 5.19:

$$(e, c) \gg_\Gamma^{\langle t \rangle} (e', c')$$

Therefore, by composition:

$$\texttt{unfocus}((e, c) \vdash_\Gamma t :: ts' \rightsquigarrow v$$

Which concludes the proof.

□

Theorem 5.26 (Completeness)**.** *For any environment* Γ*, LL(1) focused expression* (e, c) *and token sequence* ts*, whenever:*

$$\textit{zippy-ll1-parse}_\Gamma((e, c), ts) = \textit{none}$$

Then there does not *exist any value* v *such that:*

$$\textit{unfocus}((e, c)) \vdash_\Gamma ts \rightsquigarrow v$$

Proof. Let Γ be an environment, (e, c) be an LL(1) focused expression, ts a sequence of tokens. The proof proceeds by structural induction on the list of tokens.

1. In the case the list of tokens is empty, $\texttt{zippy-ll1-parse}_\Gamma((e, c), \langle\rangle)$ reduces to $\texttt{result}_\Gamma((e, c))$. By assumption, $\texttt{result}_\Gamma((e, c)) = \texttt{none}$.

   Therefore, by Theorem 5.23, we get that $\neg\text{NULLABLE}_\Gamma(\texttt{unfocus}((e, c)))$, ensures that there does not exist any value v such that $\texttt{unfocus}((e, c) \vdash_\Gamma \langle\rangle \rightsquigarrow v$.

2. Consider the case when the list of tokens is non-empty. Let t denote the first token of the list and ts' the rest. Let k be the kind of t.

   In this case, $\texttt{zippy-ll1-parse}_\Gamma((e, c), t :: ts') = \texttt{none}$ may be a result of $\zeta_\Gamma^t((e, c)) = \texttt{none}$ or may come from the recursive call.

   (a) Consider the case when $\zeta_\Gamma^t((e, c)) = \texttt{none}$. In this case, by Theorem 5.20, one gets that:

   $$k \notin \text{FIRST}_\Gamma(\texttt{unfocus}((e, c)))$$

   And therefore:

   $$\neg\exists v, \quad \texttt{unfocus}((e, c) \vdash_\Gamma t :: ts' \rightsquigarrow v$$

(b) Consider the case when $\zeta^t_\Gamma((e,c)) = \texttt{some}((e',c'))$ for some (e',c').
By assumption, one gets that $\texttt{zippy-ll1-parse}_\Gamma((e',c'), ts') = \texttt{none}$.
Therefore, by induction hypothesis:

$$\neg\exists v, \quad \texttt{unfocus}((e',c')) \vdash_\Gamma ts' \rightsquigarrow v$$

Additionally, by Theorem 5.19 one has that:

$$\texttt{unfocus}((e,c)) \gg^{\langle t\rangle}_\Gamma \texttt{unfocus}((e',c'))$$

Therefore, one can conclude that:

$$\neg\exists v, \quad \texttt{unfocus}((e,c)) \vdash_\Gamma t :: ts' \rightsquigarrow v$$

Which concludes the proof.

□

Theorem 5.27 (Correctness)**.** *For any environment Γ, LL(1) expression e, token sequence ts and value v:*

$$\textit{zippy-ll1-parse}_\Gamma(\textit{focus}(e), ts) = \textit{some}(v)$$
$$\iff$$
$$e \vdash_\Gamma ts \rightsquigarrow v$$

Proof. Immediate by definition of $\texttt{focus}(\cdot)$, Theorem 5.25, and Theorem 5.26. □

5.4 Example Execution

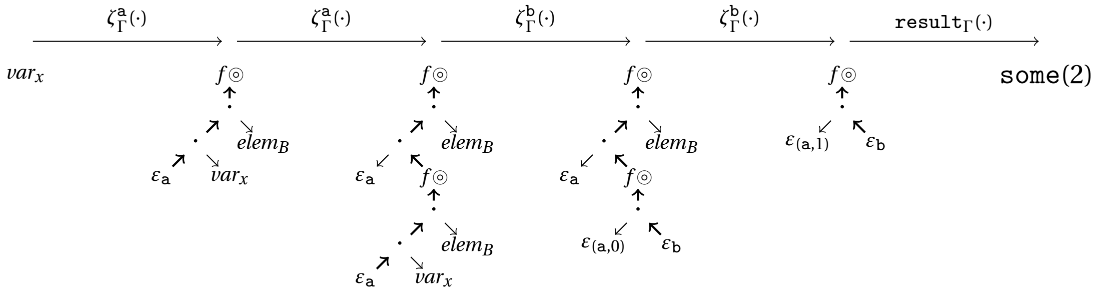
As an example of the execution of the algorithm, let us go back to our running example. Recall that, in that example, the environment Γ assigns the identifier x to the expression:

$$(f \circledcirc ((elem_A \cdot var_x) \cdot elem_B)) \vee \varepsilon_0$$

Where the function f accepts as input nested pairs of the form $((t_1, n), t_2)$ and returns the value n.

Figure 5.8 presents an overview of the execution of the algorithm on the focused expression $\texttt{focus}(var_x) = (var_x, \langle\rangle)$ and the input token sequence $\langle \mathtt{a}, \mathtt{a}, \mathtt{b}, \mathtt{b}\rangle$. At a high level, the execution consists of four calls to $\zeta_\Gamma^{\cdot}(\cdot)$, one for each input token, followed by a call to the $\texttt{result}_\Gamma(\cdot)$ function.

After each derivation step, the resulting data structure is shown. Notice that some arrows are reversed compared to the canonical tree representation of the expression. Those arrows are represented in bolder font in the diagram. At the end of those arrows are not typical expression nodes, but layers. Those reversed arrows always form a path from an expression to the root of the canonical expression tree. The expression e at the start of this path is the focal point and the stack of layers on this path is the context c. Together they form a focused expression (e, c).



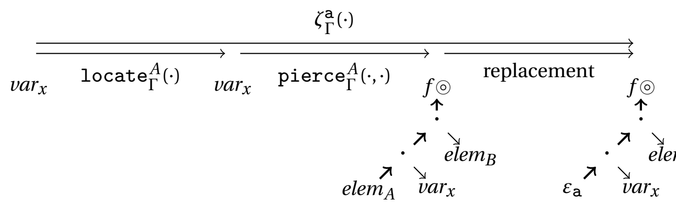
$\zeta_\Gamma^{\mathtt{a}}(\cdot)$
$\zeta_\Gamma^{\mathtt{a}}(\cdot)$
$\zeta_\Gamma^{\mathtt{b}}(\cdot)$
$\zeta_\Gamma^{\mathtt{b}}(\cdot)$
$\texttt{result}_\Gamma(\cdot)$
var_x
$f\circledcirc$
$elem_B$
$\varepsilon_{\mathtt{a}}$
var_x
$\varepsilon_{(\mathtt{a},0)}$
$\varepsilon_{\mathtt{b}}$
$\varepsilon_{(\mathtt{a},1)}$
some(2)

Figure 5.8 – Example execution of the *zippy LL(1) parsing with derivatives* algorithm. The execution consists of four derivation calls and a final call to $\texttt{result}_\Gamma(\cdot)$.

Each derivation call can be further decomposed into three phases:

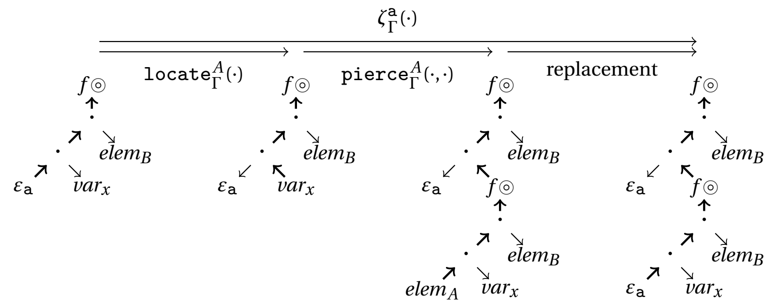
1. An upwards `locate` phase, in which the focus is moved towards the next sequent expression in the context that starts with the kind k of the derived token t, if any.
2. A downwards `pierce` phase, in which the focus is moved towards the unique leaf $elem_k$ node in the tree, eliminating disjunctions, nullifying prefixes and unrolling definitions along the way as needed.
3. A replacement phase, in which the focal $elem_k$ node is replaced by an ε_t node.

Figure 5.9 presents, for each derivation call in the current example, an overview of those three phases. Before and after each phase is a representation of the focused expression.



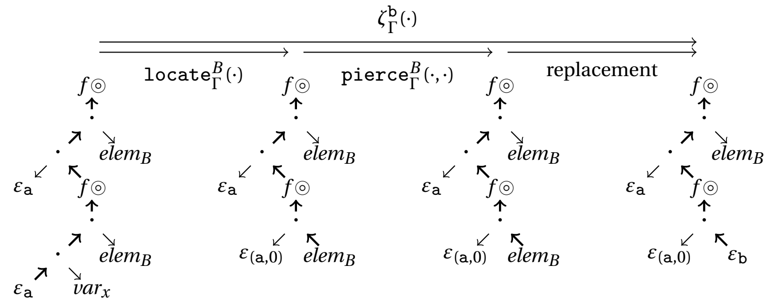
$\zeta^{\texttt{a}}_{\Gamma}(\cdot)$
$\texttt{locate}^{A}_{\Gamma}(\cdot)$
$\texttt{pierce}^{A}_{\Gamma}(\cdot,\cdot)$
replacement
var_x
$f\circledcirc$
$elem_B$
$elem_A$
$\varepsilon_{\texttt{a}}$

(a) First zippy derivation by token a.



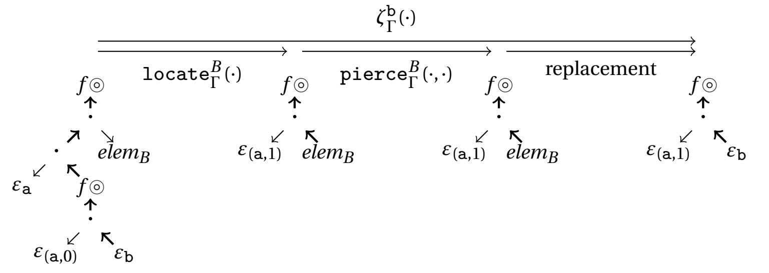
$\zeta^{\texttt{a}}_{\Gamma}(\cdot)$
$\texttt{locate}^{A}_{\Gamma}(\cdot)$
$\texttt{pierce}^{A}_{\Gamma}(\cdot,\cdot)$
replacement
$f\circledcirc$
$elem_B$
$elem_A$
$\varepsilon_{\texttt{a}}$
var_x

(b) Second zippy derivation by token a.



$\zeta^{\texttt{b}}_{\Gamma}(\cdot)$
$\texttt{locate}^{B}_{\Gamma}(\cdot)$
$\texttt{pierce}^{B}_{\Gamma}(\cdot,\cdot)$
replacement
$f\circledcirc$
$elem_B$
$\varepsilon_{\texttt{a}}$
var_x
$\varepsilon_{(\texttt{a},0)}$
$\varepsilon_{\texttt{b}}$

(c) First zippy derivation by token b.

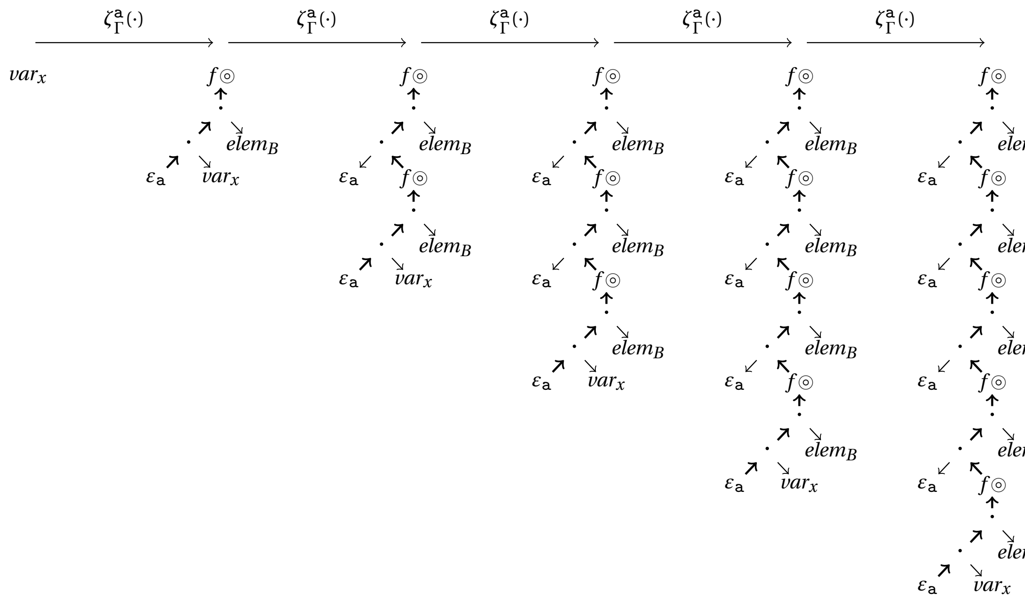


$\zeta^{\texttt{b}}_{\Gamma}(\cdot)$
$\texttt{locate}^{B}_{\Gamma}(\cdot)$
$\texttt{pierce}^{B}_{\Gamma}(\cdot,\cdot)$
replacement
$f\circledcirc$
$elem_B$
$\varepsilon_{\texttt{a}}$
$\varepsilon_{(\texttt{a},0)}$
$\varepsilon_{\texttt{b}}$
$\varepsilon_{(\texttt{a},1)}$

(d) Second zippy derivation by token b.

Figure 5.9 – Example zippy derivations. Each derivation consists of three distinct phases: An upwards `locate` phase, a downwards `pierce` phase and a replacement phase. After each phase the resulting focused expression is displayed.

To illustrate the effect of the zipper data structure on the runtime complexity of the algorithm, let us examine successive derivatives by tokens of kind A of the example context-free expression var_x. In Section 4.4.8, this specific situation exposed a problematic quadratic behaviour of the non-zippy LL(1) *parsing with derivatives* algorithm. In that earlier section, I demonstrated that the quadratic behaviour arose from the repeated traversals of the increasingly larger derivative expressions. In the present section, I will show that using a zipper data structure avoids this inefficient behaviour. In the next section, I will present a formal argument that the runtime complexity is indeed worst-case linear in the number of input tokens.



$\zeta^{\mathrm{a}}_{\Gamma}(\cdot)$
var_x
$f\circledcirc$
$elem_B$
ε_{a}

Figure 5.10 – Successive states of the parser after processing tokens of kind A.

Figure 5.10 shows the resulting data structure before and after each successive derivation by tokens of kind A. At a first glance, the situation does not seem particularly better than the one in Section 4.4.8. Indeed, the size of the data structures are similar. In this specific example, the data structures representing the derivative expressions grow with each successive derivation. There is however one significant change compared to the non-zippy version seen earlier in this thesis: Instead of pointers flowing from the expression root, pointers always flow from the point where the last derivation ended. In the case where the pointers flowed from the root, derivation needed to start back at the root and had to traverse and rebuild the accumulated layers in order to compute the next derivative. In the present case, thanks to the zipper structure, derivation can start at the point where the previous derivation ended. All nodes above a certain *horizon* point are not visited by derivation, and, thanks to immutability, can be directly referenced without requiring any extra work. Figure 5.11 illustrates this notion of horizon for one of the showcased derivations.

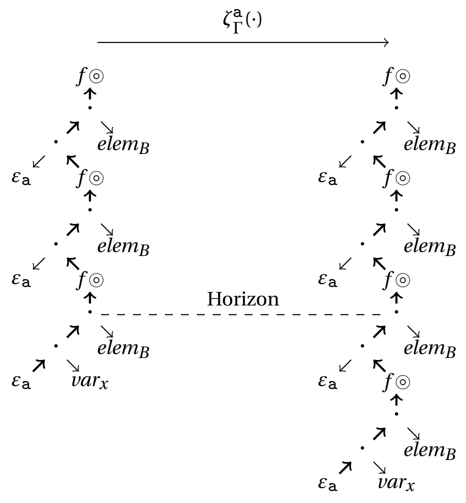

$\zeta^{\mathtt{a}}_{\Gamma}(\cdot)$
$f\circledcirc$
$elem_B$
$\varepsilon_{\mathtt{a}}$
Horizon
var_x

Figure 5.11 – Visited nodes in a derivation by a. The dashed line denotes a horizon, above which nodes are not visited during the derivation. Thanks to immutability, layer nodes above the horizon can be referenced directly in the derivative expressions.

5.5 Complexity Analysis

Now that we have an intuitive understanding of how the zipper structure can help with the runtime complexity, I present a formal argument about the worst-case time complexity of the algorithm. In order to simplify the argument, I will assume that user-defined functions that appear in *map* nodes have constant time complexity with respect to the number of input tokens consumed at the point they are applied. Since such functions generally apply constant time data constructors, such an assumption is not unreasonable. It is however theoretically possible to arbitrarily worsen the complexity result presented in this section by crafting malicious user-defined functions.

To begin the complexity analysis of the parsing algorithm, a first crucial observation is to be made: throughout its entire execution, the algorithm never instantiates new expressions, apart from trivial ε_v expressions. Instead of instantiating new expressions nodes, the algorithm instantiate *layer* nodes. This has several important consequences:

1. Calls to $\mathtt{pierce}^{\cdot}_{\Gamma}(\cdot,\cdot)$, as they act on normal expressions, can only be made on a fixed number of preexisting expressions. Those expressions are all subexpressions of the original expression and expressions in the environment, and as such have a size bounded by the initial number of nodes. Therefore, calls to $\mathtt{pierce}^{\cdot}_{\Gamma}(\cdot,\cdot)$ are bound to run in time constant with respect to the number of input tokens.

2. For the same reason, the number of layers added to the context by $\mathtt{pierce}^{\cdot}_{\Gamma}(\cdot,\cdot)$ is constant with respect to the number of tokens processed. The number of layers added is bounded by the number of nodes in the original expression and in the environment,

and is therefore also constant with respect to the number of input tokens.

3. Furthermore, since there is only a finite collection of expressions, as the input sequence of tokens grows larger, property checks performed by the parsing algorithm are bound to be repeatedly called on the same collection of expressions. It therefore makes sense to precompute properties such as first sets, nullability, as well as the value returned by $\texttt{null}_{\Gamma}(\cdot)$, for all subexpressions in the initial expression and environment. This can be performed efficiently with the help of propagation networks, as discussed earlier in Section 3.8.1. The cost of this computation, as it happens entirely before processing any token, is constant with respect to the number of tokens processed. I discuss this in more details in Section 5.6.

With this observation and its consequences in mind, let us examine the worst-case runtime complexity of the *zippy LL(1) parsing with derivatives* algorithm. Given an input of n tokens, the execution of the parsing algorithm can be decomposed into:

- At most n derivations, once per input token. Each derivation can be further decomposed into:
  - A single call to $\texttt{locate}^{\cdot}_{\Gamma}(\cdot)$,
  - At most a single call to $\texttt{pierce}^{\cdot}_{\Gamma}(\cdot,\cdot)$, which takes constant time with respect to the number of input tokens n, as argued earlier,
  - At most one constant time replacement operation.
- At most a single call to the $\texttt{result}_{\Gamma}(\cdot)$ method.

Together, the n calls to $\texttt{pierce}^{\cdot}_{\Gamma}(\cdot,\cdot)$ and the n replacement operations amount to work that is linear in the number of input tokens n. The only operations unaccounted for are the $\texttt{locate}^{\cdot}_{\Gamma}(\cdot)$ calls and the single $\texttt{result}_{\Gamma}(\cdot)$ call. I argue that those calls also amount to linear work.

Calls to $\texttt{locate}^{\cdot}_{\Gamma}(\cdot)$ and $\texttt{result}_{\Gamma}(\cdot)$ share almost the same recursive structure. Such methods repeatedly call $\texttt{plug}(\cdot,\cdot)$ to move up the layers of context until some stop condition is satisfied. In between each layer, only a constant amount of work is performed. I therefore argue that the cumulative running time of those two operations is linear in the number of layers traversed by $\texttt{plug}(\cdot,\cdot)$.

Then, a second crucial operation comes into play: Each layer is only visited at most once by $\texttt{plug}(\cdot,\cdot)$. Indeed, after a layer is visited by $\texttt{plug}(\cdot,\cdot)$, it is immediately discarded and will not appear in any subsequent derivative. The total number of layers visited by $\texttt{plug}(\cdot,\cdot)$ is therefore bounded by the total number of layer nodes *created* during the execution of the algorithm.

Layer nodes are only created in two places:

- By calls to $\mathtt{pierce}^{\cdot}_{\Gamma}(\cdot,\cdot)$. As argued before, each call to $\mathtt{pierce}^{\cdot}_{\Gamma}(\cdot,\cdot)$ may only create a constant number of layers. Since there are at most n such calls, the total number of layers created this way is linear in n.

- By calls to $\mathtt{plug}(\cdot,\cdot)$. In this case, only $\mathtt{prepend}(\cdot)$ layers are ever created, and only in the case a $\mathtt{follow\text{-}by}(\cdot)$ layer had just been visited and discarded. Therefore, the total number of layers created by calls to $\mathtt{plug}(\cdot,\cdot)$ is bounded by the total number of $\mathtt{follow\text{-}by}(\cdot)$ layers ever created. As such layers can only ever be created by calls to $\mathtt{pierce}^{\cdot}_{\Gamma}(\cdot,\cdot)$, the total number of such layers is also bounded by n.

Since only a number of layers linear in n can ever be created by the parsing algorithm, and such layers are only visited at most once, the number of layers visited is also linear in n. This concludes the proof that the *zippy LL(1) parsing with derivatives* algorithm has a worst-case time complexity that is linear the number of input tokens.

5.6 Memoisation

As explained in the previous section, a quick inspection of the *zippy LL(1) parsing with derivatives* algorithm shows that the algorithm never instantiates any new expression nodes, apart from trivial ε_v nodes. The algorithm only ever instantiates *layers*. Since no non-trivial expression nodes are ever created during the runtime of the algorithm, and given the finiteness of the original expression data structure, methods operating on expression nodes will thus be applied only on a finite number of them, and so repeatedly. This offers an opportunity to efficiently precompute or memoise those methods. Among those methods are all property checks that are used by the algorithm, such as nullability and first sets. Additionally, as I will explain, the results of the $\mathtt{pierce}^{k}_{\Gamma}(\cdot,\cdot)$ method can also be memoised.

5.6.1 Properties

The parsing algorithm is guided by two key properties of expressions: nullability and first sets. Those properties are queried during during derivation, specifically during the upwards $\mathtt{locate}^{k}_{\Gamma}(\cdot)$ phase as well as within the downwards $\mathtt{pierce}^{k}_{\Gamma}(\cdot,\cdot)$ phase. They are also used by the $\mathtt{result}_{\Gamma}(\cdot)$ function called at the end of the execution of the parsing algorithm. As discussed in Section 3.8.1, such properties can be efficiently computed using propagation networks and then cached. Checking the LL(1) property before running the parsing algorithm would also require such properties to be computed. Precomputing such properties allows for the LL(1) checking procedure to be run almost for free before parsing.

5.6.2 Calls to $\texttt{pierce}^{k}_{\Gamma}(\cdot,\cdot)$

Interestingly, calls to $\texttt{pierce}^{k}_{\Gamma}(\cdot,\cdot)$ can also be memoised. Given a kind k, an expression e, and a context c, calls to $\texttt{pierce}^{k}_{\Gamma}(e,c)$ append a sequence of layers on top of the stack c. Importantly, the sequence of layers added to the context c is independent of the parameter context c, it only depends on k and e. Since both e and k range over finite domains, it is interesting to memoise calls to $\texttt{pierce}^{k}_{\Gamma}(e,c)$ by caching the sequence of stacks to be added to the context for every pair of expression e and kind k.

Since the sequence of layers is to be appended on top of the current context c, I suggest storing this sequence of layers in reverse order, so that it is possible to add all layers on top of the stack in an efficient tail-recursive fashion.

5.7 Comparison with Traditional LL(1) Parsing

In this chapter, I have presented a parsing algorithm for LL(1) expressions that I derived rather naturally by combining Brzozowski's derivatives (Brzozowski, 1964; Might et al., 2011) with Huet's zippers (Huet, 1997). Interestingly, this algorithm, as I will shortly explain, has striking similarities with the traditional LL(1) parsing algorithm (Stearns and Lewis, 1969; Aho et al., 2006).

5.7.1 Presentation of Traditional LL(1) Parsing

The traditional LL(1) parsing algorithm operates on two data structures: a stack and a table. The stack is a sequence of terminal and non-terminal symbols, which is initially set to only contain the start symbol of the LL(1) grammar. At each iteration, the top symbol of the stack is removed and replaced by zero or more symbols as follows:

- In case the symbol removed from the stack is a terminal symbol, it is matched against the next token of input. If the two agree, the token is consumed and no extra symbols are added on top of the stack. If the two disagree, a parsing error is reported.
- In case of a non-terminal symbol, the combination of the non-terminal and the kind of the next token, if any, is looked up in the parsing table. In case all input tokens have already been processed, a distinct placeholder kind `EOF` is used instead of the non-existent token kind. For each such pair of non-terminal and kind, the parsing table indicates which grammar rule is to be applied, if any. The top of the stack is replaced with the right-end side of the rule, with the first symbol of the production ending up on top of the stack. If the entry in the table is empty, the algorithm reports a parsing error.

The execution ends successfully when all input tokens have been processed and the stack is empty. Handling the elaboration of the resulting parse tree requires extra bookkeeping: The algorithm must keep track of what rules were applied in order to progressively build the parse tree. This aspect of the algorithm is often brushed over in explanations of LL(1) parsing.

As just explained, the traditional LL(1) parsing algorithm relies heavily on the parsing table. This parsing table is initialised before processing input tokens according to the two following rules:

- For each grammar rule r of the form $L \mapsto \langle R_1, \dots, R_n \rangle$, the rule r is added to the table at the index (L, k) for every k in the `FIRST` set of $\langle R_1, \dots, R_n \rangle$. The `FIRST` set of a (sequence of) symbols is the sets of kinds that may start described sequences. It corresponds exactly to the notion of $\texttt{FIRST}_\Gamma(\cdot)$ sets presented in this thesis.
- For each nullable non-terminal L, the rule $L \mapsto \langle\rangle$ is added to the table at the index (L, k) for every k in the `FOLLOW` set of L. The `FOLLOW` set of a non-terminal contains all kinds that may directly follow in sequence after the non-terminal. The `FOLLOW` set of a non-terminal is set to contain the artificial kind `EOF` in case the non-terminal appears in trailing position in the grammar.

Thanks to the LL(1) property, at most a single entry is associated with each pair of non-terminal and token kind.

5.7.2 Similarities and Differences between the two Approaches

The LL(1) *parsing with derivatives and zippers* algorithm presented in this thesis is in essence very similar to the traditional LL(1) parsing algorithm. The two algorithms operate on functionally equivalent stacks. In the case of the *zippy LL(1) parsing with derivatives* algorithm, this stack arose naturally from the use of a zipper. In addition, it not only contains subsequent symbols (stored in $\texttt{follow-by}(\cdot)$ nodes) but also partial values (stored in $\texttt{prepend}(\cdot)$ layers) and references to user-defined functions to be applied (stored in $\texttt{apply}(\cdot)$ layers). The parsing table of the traditional algorithm can also be found in the *zippy LL(1) parsing with derivatives* algorithm in the form of the (memoised) $\texttt{pierce}^{\cdot}_\Gamma(\cdot, \cdot)$ method.

The main functional difference between the two algorithms is in the handling of nullable symbols. In the case of the traditional LL(1) algorithm, a rule of the form $L \mapsto \langle\rangle$ is to be applied only in case the non-terminal L is nullable and the kind of the next token is part of the `FOLLOW` set of L. This rule, as any other rule, is found in the parsing table. In the case of the *zippy LL(1) parsing with derivatives algorithm*, there is no notion of `FOLLOW` sets. The dual notion of should-not-follow sets is only used for LL(1) checking, not at parse time. Nullifying the focal point is the operation that corresponds to applying a rule of the form $L \mapsto \langle\rangle$. This operation is applied by two methods: $\texttt{locate}^{\cdot}_\Gamma(\cdot)$ and $\texttt{result}_\Gamma(\cdot)$. The $\texttt{locate}^{\cdot}_\Gamma(\cdot)$ and $\texttt{result}_\Gamma(\cdot)$ methods will nullify the expression in focus in case it is nullable and, in case of $\texttt{locate}^{\cdot}_\Gamma(\cdot)$, when it does not contain the desired kind in its first set. This condition is slightly weaker than the traditional condition expressed using `FOLLOW` sets, meaning that the traditional LL(1) parsing algorithm could potentially stop and report an error slightly earlier.

Note that the `FOLLOW` set is not precise enough to entirely avoid the application of nullifying rules, as showed by the following example. Consider an LL(1) grammar with non-terminals S, A, B, E and terminals a, b, c, with starting symbol S. The grammar admits the following rules:

$$
\begin{aligned}
S &\mapsto \langle A \rangle \\
S &\mapsto \langle B \rangle \\
A &\mapsto \langle a, E, a \rangle \\
B &\mapsto \langle b, E, b \rangle \\
E &\mapsto \langle \rangle \\
E &\mapsto \langle c \rangle
\end{aligned}
$$

Consider the input sequence $\langle a, b \rangle$, which is *not* accepted by the grammar. Given this input, and starting with an initial stack containing only S, the traditional LL(1) parsing algorithm executes the following operations:

1. Apply the rule $S \mapsto \langle A \rangle$ on the first element of the stack, resulting in the stack $\langle A \rangle$.
2. Apply the rule $A \mapsto \langle a, E, a \rangle$ on the first element of the stack, resulting in $\langle a, E, a \rangle$.
3. Remove the non-terminal a on top of the stack. Since it matches the next token of input, that token is consumed. At this point, the stack is reduced to $\langle E, a \rangle$ and the remaining input is reduced to $\langle b \rangle$.
4. Apply the rule $E \mapsto \langle \rangle$, resulting in the stack $\langle a \rangle$. This rule is applicable as E is nullable and as the next token, `b`, is part of the `FOLLOW` set of E.
5. Remove the non-terminal a on top of the stack. Since it does not match with the next token of input b, an error is reported.

In the fourth step, the non-terminal E was nullified even though the next token b could not be accepted at this point. The error is detected only after E is nullified. At the point of error, the stack will already have been altered so that it would not be possible to directly resume execution at that point with different remaining tokens.

The `FOLLOW` set thus is not sufficiently precise to avoid this type of applications. At this point, one wonders why bother with `FOLLOW` sets at all? Indeed, a modified version of the parsing table where the entry $L \mapsto \langle \rangle$ is added for all pairs of nullable non-terminals L and kinds k that are not part of the first set of L would also be entirely valid. Remains the question of LL(1) checking. As showcased in this thesis, the notion of should-not-follow sets, which is more compositional, is sufficient for LL(1) checking.

5.7.3 Advantages over the Traditional LL(1) Parsing Approach

Even though the two algorithms are in the end very similar, I argue that the *zippy LL(1) parsing with derivatives* approach has several advantages over the traditional approach:

- The zippy LL(1) parsing with derivatives algorithm deeply incorporates value elaboration. It supports the application of user-defined functions to build the resulting parse value. In the traditional LL(1) parsing approach, parse tree generation is often an afterthought. Furthermore, such approaches generally offer no support for the application of user-defined functions.
- The state of the zippy LL(1) parsing with derivative algorithm is represented as an expression with clearly defined semantics. As I demonstrate in this chapter, this representation makes it easy to reason about states and operations on states. This is further demonstrated by a Coq formalisation of the approach by Jad Hamza and myself (Hamza and Edelmann, 2019; Edelmann et al., 2020).
- The zippy LL(1) parsing with derivatives operates on immutable, persistent data structures. This makes it trivial to revert back to previous states and to share structures amongst many such states.
- Finally, the algorithm arises very naturally from the combination of Brzozowski's derivatives and Huet's zipper. This contrasts with the seemingly more *ad hoc* nature of the traditional LL(1) parsing algorithm. I argue that this has potential to makes the algorithm more teachable.

Additionally, as I will show shortly in Chapter 6, the approach naturally supports of parser combinator interface. This makes the approach easy to embed in functional programming languages.

6 SCALL1ON: A Scala Parser Combinator Library for LL(1) Languages

In this chapter, I present SCALL1ON, or simply *scallion*, a Scala parsing combinators library for LL(1) languages. The library features an applicative combinators-based interface that should be familiar to programmers experienced with parser combinators libraries. A collection of high-level combinators are available out of the box. The library is freely available online at https://github.com/epfl-lara/scallion.

Contrary to most parsing combinator libraries, SCALL1ON does not employ a shallow embedding of combinators. In traditional parsing combinators libraries, parsers and combinators are usually directly represented as host-language functions. Instead, SCALL1ON features a deep embedding of context-free expressions in Scala, in which expressions are represented as a reified datatype. This deep embedding enables many interesting features that are not traditionally found in parser combinators libraries, such as analysis of parsers, precise error description, parser resumption, enumeration, and pretty printing.

The theoretical foundations of the library are those presented in earlier chapters of this thesis. Context-free expressions implemented by the library correspond to those presented in Chapter 3. Properties of expressions, such as nullability and first set, follow the definitions of that chapter. The library implements the LL(1) checking procedure described in Chapter 4, and the *zippy LL(1) parsing with derivatives* algorithm presented in Chapter 5.

6.1 Overview

SCALL1ON is a parser combinators library for Scala. To illustrate how programmers may use the library, Figure 6.1 presents a JSON parser written using SCALL1ON. The code presents a parser operating on JSON tokens, and therefore relies on the existence of a JSON lexer to produce such tokens. The straightforward definition of JSON tokens, kinds, and values have been omitted in the interest of space. Such definitions can be found in Appendix C.

```scala
import scallion._
import JSON._

object JSONParser extends Parsers {
  override type Token = JSON.Token
  override type Kind = JSON.Kind
  override def getKind(token: Token): Kind = Kind.of(token)
  import Implicits._

  val booleanSyntax: Syntax[Value] = accept(BooleanKind) {
    case BooleanToken(value) => BooleanValue(value)
  }

  val numberSyntax: Syntax[Value] = accept(NumberKind) {
    case NumberToken(value) => NumberValue(value)
  }

  val stringSyntax: Syntax[StringValue] = accept(StringKind) {
    case StringToken(value) => StringValue(value)
  }

  val nullSyntax: Syntax[Value] = accept(NullKind) {
    case NullToken() => NullValue()
  }

  implicit def separatorSyntax(char: Char): Syntax[Token] = elem(SeparatorKind(char))

  lazy val arraySyntax: Syntax[Value] =
    ('[' ~ repsep(jsonSyntax, ',') ~ ']').map {
      case _ ~ vs ~ _ => ArrayValue(vs)
    }

  lazy val bindingSyntax: Syntax[(StringValue, Value)] =
    (stringSyntax ~ ':' ~ jsonSyntax).map {
      case key ~ _ ~ value => (key, value)
    }

  lazy val objectSyntax: Syntax[Value] =
    ('{' ~ repsep(bindingSyntax, ',') ~ '}').map {
      case _ ~ bs ~ _ => ObjectValue(bs)
    }

  lazy val jsonSyntax: Syntax[Value] = recursive {
    arraySyntax | objectSyntax | booleanSyntax |
      numberSyntax | stringSyntax.up[Value] | nullSyntax
  }

  val jsonParser: Parser[Value] = Parser(jsonSyntax)

  def apply(it: Iterator[Token]): ParseResult[Value] = jsonParser(it)
}
```

Figure 6.1 – JSON parser in SCALL1ON.

On line 4, a JSONParser object is defined. The object extends Parsers, a trait provided by the library. Extending the Parsers trait enriches the scope of the object body with parser combinators and related types, classes and functions. The trait requires of the object to implement the Token and Kind types, as well as the getKind method. In the present example, this is done on lines 5, 6 and 7. On line 8, implicit values are added to the scope. Such implicit values are required by a few of the combinators, such as repsep, for reasons linked to pretty printing that I will explain in Section 6.2.3.

On lines 10 - 24, four different Syntaxes are defined. Such syntaxes respectively correspond to the syntactic constructs for JSON booleans, numbers, strings and null values. In each case, the accept function is used to define the behaviour of the syntax. The function takes as parameter a Kind and a function from a single Token of that kind to some value of a chosen type. For each of the four present syntaxes, the parameter function of accept is used to map tokens of the accepted kind into actual JSON values of type Value, or the more specific StringValue in case of stringSyntax.

On line 26, a function is defined to convert single characters, such as '[' or ',', into a syntax for the separator represented by that character. The function offer an implicit conversion from such characters to Syntax objects, meaning that literal characters can be used as Syntax objects in the body of JSONParser.

On lines 28 - 31, syntax for JSON arrays is defined. The syntax of arrays consists of a sequence of three syntaxes:

1. A syntax for a single '[' separator to mark the start of the array,
2. A syntax for JSON values separated by ',' separators as the body of the array,
3. A syntax for a single ']' separator to mark the end of the array.

The sequence of the three syntaxes is built using the ~ operator of Syntax. Repetition with separation is implemented by the library combinator repsep. Finally, a call to the map combinator is used to specify how to merge the values from the three underlying syntax in the sequence into a single JSON Value representing the array.

Note that the syntax defined on lines 28 - 31 refers to the global syntax for JSON values (jsonSyntax), whose definition has not yet appeared. The lazy modifier is added to the definition so that the JSONParser object correctly initialises.

On lines 33 - 36, the syntax for key-value bindings is defined. Right after, on lines 38 - 41, the syntax for JSON objects is defined. The syntax definition is similar to that of arrays. However, the start and end marks differ. Furthermore, in the case of objects, the syntax bindingSyntax is repeated, while jsonSyntax is repeated in the case of arrays. Finally, a different constructor (ObjectValue) is applied to represent the resulting JSON Value.

On lines 43 - 46, the global syntax for JSON values is defined. It consists of the disjunction of all other JSON value syntaxes that were previously defined. Disjunctions are built using the `|` method of `Syntax`. The combinator `recursive` (line 43) is used to introduce further laziness and defer the computation of its parameter syntax. At a high level, this combinator makes it possible to define recursive `Syntax` objects such as `jsonSyntax`: `jsonSyntax` appears in `arraySyntax` and in `objectSyntax` via `bindingSyntax`, both of which appear in `jsonSyntax`.

Note that, on line 45, the method `up` is used to upcast a `Syntax[StringValue]` into a `Syntax[Value]`. This method invocation is necessary as, in SCALL1ON, the type `Syntax` is invariant in its type parameter: `Syntax[StringValue]` is not a subtype of `Syntax[Value]`, even though `StringValue` is a subtype of `Value`. This invariance stems from the ambivalent nature of SCALL1ON's syntaxes: Syntaxes aim to describe both parsers and pretty-printers, each of which constrain variance in opposite ways, as I will later explain.

Next, on line 48, the syntax is converted into an actual `Parser` object. At this point, properties of the syntax, such as its first set and its nullability, are computed. The LL(1) property of the syntax is also checked. In case of LL(1) conflicts, an exception is thrown at this point. Later, in Section 6.2.6, I will explain in details how SCALL1ON can help programmers debug such conflicts.

Finally, on line 50, the `apply` method of `JSONParser` is defined. Given an input iterator of tokens, the method simply calls the `Parser[Value]` object created on line 48 with the parameter tokens, which results in a `ParseResult` value. The `ParseResult` value either indicates a successful parse, in which case a parsed value is also returned, or indicates a parse error.

The code presented so far, apart from the LL(1) aspect and a few other idiosyncrasies, is very similar to code that one would write in other parsing combinator libraries. SCALL1ON's LL(1) checking phase, which takes place parser creation time, eliminates design errors that could potentially go unnoticed in some other approaches. Thanks to the *zippy LL(1) parsing with derivatives* algorithm, the runtime complexity of parsing is worst-case linear in the number of input tokens. Additionally, as I will shortly demonstrate, SCALL1ON provides features that go above and beyond what is generally possible in other parsing combinator libraries. I will demonstrate some of those features in a series of Scala console interactions with the example parser.

Property checking As a first series of interactions with the Scala console, I show that properties of the various syntaxes can be checked by simple method calls.

```
// Import members of the JSONParser object.
scala> import JSONParser._
import JSONParser._
```

```
// Check the LL(1) property of the top-level JSON syntax.
scala> jsonSyntax.isLL1
res1: Boolean = true

// Check if the syntax for arrays is productive.
scala> arraySyntax.isProductive
res2: Boolean = true

// Return the first set of the arrays syntax.
scala> arraySyntax.first
res3: Set[Kind] = Set(SeparatorKind('['))
```

The possibility to explicitly compute properties of syntaxes is a feature that distinguishes SCALL1ON from usual parser combinators libraries.

Enumeration Furthermore, SCALL1ON also implements enumeration of recognised sequences. In the code snippet below, an iterator over all sequences of token kinds describing JSON arrays is created. The iterator produces sequences ordered by increasing length. Afterwards, the first three such sequences are displayed.

```
// Create of the iterator.
scala> val it = Enumerator(arrayValue)
it: Iterator[Iterator[Kind]] = <iterator>

// Access the first element.
scala> it.next().toList
res4: List[Kind] = List(SeparatorKind('['), SeparatorKind(']'))

// Access the second element.
scala> it.next().toList
res5: List[Kind] = List(SeparatorKind('['), BooleanKind, SeparatorKind(']'))

// Access the third element.
scala> it.next().toList
res6: List[Kind] = List(SeparatorKind('['), NumberKind, SeparatorKind(']'))
```

Parsing Expectedly, once converted to `Parser` objects, syntaxes can be used for parsing. In the code snippet below, the parser is queried with the token sequences corresponding to the strings `"[1]"` and `"[1}"`. The parsed value is then tentatively extracted from the parse result.

In the first case, a JSON `ArrayValue` object is returned. Conversely, no parse value is available in the second case, indicating a parse error.

```
// Run the parser on "[1]".
scala> parser(JSONLexer("[1]")).getValue
res7: Option[Value] = Some(ArrayValue(Vector(NumberValue(1.0))))

// Run the parser on "[1}".
scala> parser(JSONLexer("[1}")).getValue
res8: Option[Value] = None
```

Parse errors The `ParseResult` object returned by the parser carries around more information than simply an `Option[Value]`. In case of errors, the `ParseResult` object indicates the type of error. There are two different types of errors:

1. An `UnexpectedToken` error, which indicates that an unexpected token was encountered. The actual token that caused the error is provided. The parameter token iterator is not consumed further than that token.
2. An `UnexpectedEnd` error, which indicates that the end of the input was reached, but no value is available yet. This error indicates an incomplete input.

In the code snippet below, the first invocation of the parser results in an `UnexpectedToken` error, while the second results in an `UnexpectedEnd` error.

```
// Run the parser on "[1}".
scala> parser(JSONLexer("[1}"))
res9: ParseResult = UnexpectedToken(SeparatorToken('}'), Focused(...))

// Run the parser on "[1".
scala> parser(JSONLexer("[1"))
res10: ParseResult = UnexpectedEnd(Focused(...))
```

Residual parsers All `ParseResult` objects also contain a *residual parser*. In the above code snippet, the `Focused(...)` fragments in the string representation of the results hinted at the presence of such residual parsers arguments. Thanks to the derivatives-based parsing algorithm, this residual parser is available for free, as it is built and maintained by the algorithm. This residual parser can be accessed through the `rest` field of any `ParseResult` object.

Interestingly, this residual parser is just like any regular parser users would write, and can be queried as such. In the code snippet below, the residual parser obtained after processing `"[1"` is stored under the name `residual`. Afterwards, the first set of the residual parser is queried, and then the residual parser is used to process the tokens corresponding to `", 2]"`, resulting in a parse value representing the entire JSON array `[1, 2]`.

```
scala> val residual = parser(JSONLexer("[1")).rest
residual: Parser[Value] = Focused(...)

scala> residual.first
res11: Set[Kind] = Set(SeparatorKind(']'), SeparatorKind(','))

scala> residual(JSONLexer(", 2]")).getValue
res12: Option[Value] = Some(ArrayValue(
    Vector(NumberValue(1.0), NumberValue(2.0))))
```

Having access to reified residual parsers is tremendously helpful to implement features such as informative error messages, error recovering, and even code completion, and so with little effort.

Pretty printing Finally, SCALL1ON syntaxes can also be converted to pretty printers in addition to parsers. This however requires slight modification to the code: Programmers must supply inverses of `map` argument functions. The code with local inverses is provided in Appendix D. With this change in place, creating a pretty printer and querying it is straightforward. In the code fragment below, a pretty printer is created for the global JSON syntax. Next, the sequence of tokens corresponding to the JSON value for the array `[null, 3.0]` is computed.

```
scala> val printer = PrettyPrinter(jsonSyntax)
printer: PrettyPrinter[Value] = PrettyPrinter(...)

scala> printer(ArrayValue(Seq(NullValue, NumberValue(3.0)))).get.toList
res13: List[Token] = List(SeparatorToken('['), NullToken(),
    SeparatorToken(','), NumberToken(3.0), SeparatorToken(']'))
```

Given correct inverses for user-defined functions, the pretty printer is guaranteed to return a sequence of tokens that can be parsed back to the same value. Additionally, under the same assumption, SCALL1ON will always return a sequence of tokens of minimal length. In this particular exemple, there is exactly one sequence of tokens corresponding to each JSON value.

6.2 Programming Interface

6.2.1 The Parsers trait

In order to support arbitrary tokens and kinds, SCALL1ON provides all its functionalities as part of a trait which leaves the type of tokens Token and the type of token kinds Kind abstract. The trait is called Parsers, and users of the library are expected to extend it. In addition to specifying the type of tokens and kinds, users are also required to implement the getKind function, which given a Token, returns its unique Kind.

Instead of extending the trait Parsers in the appropriate object, as done in the example JSON parser presented in overview, users may also create a parser object and import definitions from it in the appropriate scope.

```scala
val parsers = new Parsers {
  type Token = ...
  type Kind = ...
  def getKind(token: Token): Kind = ...
}
import parsers._
```

6.2.2 The Syntax Datatype

The base type provided by SCALL1ON is Syntax[A]. Syntax[A] objects are deeply embedded context-free expressions. A simplified view of the algebraic datatype is presented below. Most constructors correspond straightforwardly to that of context-free expressions as defined in Chapter 3. The few differences are highlighted afterwards.

```scala
sealed trait Syntax[A]
case class Success[A](value: A) extends Syntax[A]
case class Failure[A]() extends Syntax[A]
case class Elem(kind: Kind) extends Syntax[Token]
case class Transform[A, B](
  function: A => B,
  inverse: B => Seq[A],
  inner: Syntax[A]) extends Syntax[B]
case class Marked[A](mark: Mark, inner: Syntax[A]) extends Syntax[A]
case class Disjunction[A](left: Syntax[A], right: Syntax[A]) extends Syntax[A]
case class Sequence[A, B](left: Syntax[A], right: Syntax[B]) extends Syntax[A ~ B]
sealed abstract class Recursive[A] extends Syntax[A] {
  def inner: Syntax[A]
}
```

```
object Recursive {
  def apply[A](syntax: => Syntax[A]): Syntax[A] =
    new Recursive[A] {
      override lazy val inner: Syntax[A] = syntax
    }
}
```

The first change compared to the formalism of Chapter 3 is that the `Transform` constructor accepts an additional parameter for an inverse of the parameter function. Next is the presence of the `Marked` constructor, which has no influence on parsing but may play a role for enumeration purposes. Finally, the `Recursive` constructor encodes both variables and the environment of the formalism under the same construct: `Recursive` instances act as variables, and the associated expression in the environment is stored in the `inner` field of the variable. Thanks to the use of laziness, the `Recursive` construct allows for mutually recursive syntaxes.

Note that `Syntax` is invariant in its type parameter `A`. The reason is that syntaxes in SCALL1ON represent both parsers, in which `A` appears in covariant position, and pretty printers, in which `A` appears in contravariant position. The method up is however provided to upcast syntaxes.

6.2.3 Combinators

Although the `Syntax` case class constructors are available to users of the library, the preferred way to build syntaxes is through a collection of smart constructors and combinator functions. Note that, while some such functions directly correspond to constructors of the `Syntax` class, some of them encode higher-level patterns.

Such combinators are typical of parser combinator libraries (Leijen and Meijer, 2001; LAMP EPFL and Lightbend, Inc, 2019). The presence of certain methods obeying a set of associated laws makes SCALL1ON's `Syntax` a de facto instance of many well-known type classes (Wadler and Blott, 1989; Yorgey, 2009) such as the `Functor`, `Applicative` (McBride and Paterson, 2008), and `Alternative` type classes. Membership in such type classes, while not explicitly reflected in the implementation, grants programmers familiar with such concepts insights on how `Syntax` objects behave and can be composed.

Tables 6.1 and 6.2 list a selection of the smart constructors and basic combinators available in SCALL1ON. Table 6.3 presents various combinators that are implemented as methods of the `Syntax` trait. Finally, Table 6.4 presents a collection of combinators for defining operators of programming languages, notably infix binary combinators with various priority levels and directions of associativity.

Smart constructors
`def epsilon[A](value: A): Syntax[A]` Smart constructor for `Success`.
`def failure[A]: Syntax[A]` Smart constructor for `Failure`.
`def elem(kind: Kind): Syntax[Token]` Smart constructor for `Elem`.

`def accept[A](kind: Kind)(`
`  function: PartialFunction[Token, A],`
`  inverse: (A) => Seq[Token] =`
`    (x: A) => Seq()): Syntax[A]`
 	
Smart constructor for `Elem`, with a `function` directly applied on the matched `Token`.
An optional `inverse` can be provided for pretty printing purposes.

`def recursive[A](inner: => Syntax[A]): Syntax[A]` Smart constructor for `Recursive`. Note that the `inner` arguments are passed by name, which makes it possible to build mutually recursive definitions.

Table 6.1 – `Syntax` smart constructors.

Combinators
`def many[A](inner: Syntax[A]): Syntax[Seq[A]]` Zero or more repetitions.
`def many1[A](inner: Syntax[A]): Syntax[Seq[A]]` One or more repetitions.
`def opt[A](inner: Syntax[A]): Syntax[Option[A]]` Zero or one repetition.
`def repsep[A, B: Uninteresting](rep: Syntax[A], sep: Syntax[B]): Syntax[Seq[A]]` Zero or more repetitions, separated by a separator syntax. The `Uninteresting` constraint ensures that values of type `B` hold not interesting values and can safely be discarded. When pretty printing, such values of type `B` need be synthesised even though they do not contribute to the value of type `Seq[A]` to be pretty printed. I will discuss this in more depth in Section 6.2.3.
`def rep1sep[A, B: Uninteresting](rep: Syntax[A], sep: Syntax[B]): Syntax[Seq[A]]` One or more repetitions, separated by a separator syntax.

Table 6.2 – Basic combinators for describing syntaxes.

Combinator methods of `Syntax[A]`

```
def ~[B](that: Syntax[B]): Syntax[A ~ B]
```

Sequencing of `this` and `that` syntaxes. The result is wrapped in a pair of type `A ~ B` for easier pattern matching.

```
def ~<~[B](that: Syntax[B])(implicit ev: Uninteresting[B]): Syntax[A]
```

Sequencing of `this` and `that` syntaxes, with the right value discarded.

```
def ~>~[B](that: Syntax[B])(implicit ev: Uninteresting[A]): Syntax[B]
```

Sequencing of `this` and `that` syntaxes, with the left value discarded.

```
def |(that: Syntax[A]): Syntax[A]
```

Disjunction of `this` and `that` syntaxes.

```
def ||[B](that: Syntax[B]): Syntax[Either[A, B]]
```

Tagged disjunction of `this` and `that` syntaxes.

```
def map[B](function: A => B,
  inverse: (B) => Seq[A] = (b: B) => Seq()): Syntax[B]
```

Application of a function onto the parsed values. An optional `inverse` argument can be provided for pretty printing purposes.

```
def up[B >: A](implicit ev: Manifest[A]): Syntax[B]
```

Upcasting of `this` syntax. The implicit `Manifest` argument ensures that values can be checked to be of type `A` when pretty printing. Pretty printed values (of type `B`) that are not of the more specific type `A` can be discarded while pretty printing.

Table 6.3 – Methods of `Syntax[A]`.

Operators-related combinators

```
def infixLeft[Op, A](elem: Syntax[A], op: Syntax[Op])(
  function: (A, Op, A) => A,
  inverse: PartialFunction[A, (A, Op, A)] = PartialFunction.empty): Syntax[A]
```

Repetition of `elem` syntaxes separated by `op` syntaxes. Parsed values are the result of applying the provided `function` onto the various underlying values in a left-associated manner. The optional `inverse` function provides a way to support pretty printing.

```
def infixRight[Op, A](elem: Syntax[A], op: Syntax[Op])(
  function: (A, Op, A) => A,
  inverse: PartialFunction[A, (A, Op, A)] = PartialFunction.empty): Syntax[A]
```

Repetition of `elem` syntaxes separated by `op` syntaxes. Parsed values are the result of applying the provided `function` onto the various underlying values in a right-associated manner. The optional `inverse` function provides a way to support pretty printing.

```
def prefixes[Op, A](op: Syntax[Op], elem: Syntax[A])(
  function: (Op, A) => A,
  inverse: PartialFunction[A, (Op, A)] = PartialFunction.empty): Syntax[A]
```

Repetition of `op` syntaxes followed by a single instance of the `elem` syntax. Parsed values are the result of applying the provided `function` onto the various underlying values in a right-associated manner. The optional `inverse` function provides a way to support pretty printing.

```
def postfixes[Op, A](elem: Syntax[A], op: Syntax[Op])(
  function: (A, Op) => A,
  inverse: PartialFunction[A, (A, Op)] = PartialFunction.empty): Syntax[A]
```

Single instance of `elem` syntax followed by a repetition of `op` syntaxes. Parsed values are the result of applying the provided `function` onto the various underlying values in a left-associated manner. The optional `inverse` function provides a way to support pretty printing.

Table 6.4 – Combinators for describing postfix, prefix and infix operators.

As seen in Tables 6.1 to 6.4, SCALL1ON provides a sizeable collection of combinators, most of which encode patterns that are typically encountered in the syntax of formal languages, notably various forms of repetitions. While users are certainly free to encode such patterns themselves through mutually recursive syntax definitions, using such predefined combinators can save time and effort. The argument is the same as for using *folds* instead of recursive functions in the more general setting of functional programming languages (Gibbons, 2003). In addition, the combinators of SCALL1ON encode recursive patterns that are compatible with LL(1) constraints. In particular, no combinator makes use of left-recursion.

Skipping Values

Often, when building a parser, some syntaxes will not produce any semantically interesting values. Delimiters and separators often do not contribute directly to the produced value, but also serve as syntactic markers. When such syntaxes appear in sequences of syntaxes, the uninteresting value produced by such syntaxes however appear in the result. Such values can end up cluttering the parsed value. For instance, consider the syntax for JSON array of Listing 22. Because the values produced by the syntaxes `'['` and `']'` appear as part of the parse value of the sequence, such values must be handled by the call to the `map` combinator.

```
lazy val arraySyntax: Syntax[Value] = ('[' ~ repsep(jsonSyntax, ',') ~ ']').map {
  case _ ~ elems ~ _ => ArrayValue(elems)
}
```

Listing 22 – JSON array syntax. Features delimiter syntaxes with uninteresting values.

To offer a clutter-free way of handling such syntaxes, SCALL1ON offers a `skip` combinator which can be used within sequences. The JSON array syntax previously shown can make use of such a combinator to indicate that the delimiter syntaxes `'['` and `']'` produce uninteresting values that should be skipped when building up the resulting value (see Listing 23).

```
lazy val arraySyntax: Syntax[Value] = ('['.skip ~ repsep(jsonSyntax, ',') ~ ']'.skip).map(
  elems => ArrayValue(elems))
```

Listing 23 – JSON array syntax. The combinator `skip` is called on syntaxes with uninteresting values.

The `skip` method of `Syntax[A]` has the following signature:

```
def skip(implicit ev: Uninteresting[A]): Skip
```

The returned `Skip` class provides overloaded definitions of the sequencing combinator ~. In addition, `Syntax` also provide an overloaded definition of ~ which accepts values of type `Skip` as argument. Those overloaded definitions ensure that `skip` can be called anywhere in a sequence of syntaxes written using ~.

Uninteresting Values

As just discussed, some of the combinators in SCALL1ON accept argument syntaxes that do not contribute to the resulting value. As an example, consider the separator parameter of `repsep` and the `skip`'d separator syntaxes of Listing 23. The values produced by such syntaxes are *uninteresting*: they do not contribute to the parse values produced by larger syntaxes. When pretty printing, it is therefore impossible to decide what value to pass down to such syntaxes based on the value to be pretty printed.

The constraint `Uninteresting` ensures that the type of values produced by a syntax can be safely discarded at parse time, and potentially synthesised at pretty printing time. Users of SCALL1ON can choose between two modes depending on what instances of `Uninteresting` they provide:

- By using the instance from `SafeImplicits`, only `Unit` is considered uninteresting. In this case, it is possible to synthesis `()` as the value given to the uninteresting syntaxes during pretty printing, ensuring that pretty printing works correctly.

- In contrast, when users are not interested in the pretty printing capabilities offered by SCALL1ON, they can instead use instances provided by `Implicits`. In this case, all types are considered `Uninteresting`, and any uninteresting syntax will not be queried during pretty printing, which often makes the pretty printer incomplete.

This `Uninteresting` mechanism allows users to be warned when using constructs that would otherwise discard values and break pretty printing, while still allowing such constructs.

6.2.4 Parser Construction

In SCALL1ON, Syntax objects are mere description of syntaxes. In order to obtain an actual parser out of them, users must convert them into a proper Parser instance beforehand. At the user level, doing so only takes a single call to the Parser factory object. Objects that implement the Parser trait offer methods to parse sequences of tokens, return properties of the Parser, or even convert the Parser back to a Syntax object.

```
sealed trait Parser[A] {
  def apply(tokens: Iterator[Token]): ParseResult[A]

  def nullable: Option[A]
  def isNullable: Boolean = nullable.nonEmpty
  def isProductive: Boolean = isNullable || first.nonEmpty
  def first: Set[Kind]

  def syntax: Syntax[A]
}

object Parser {
  def apply[A](syntax: Syntax[A], enforceLL1: Boolean = true): Parser[A] = // Omitted.
}
```

During this conversion process into a Parser object, properties of the Syntax expression and of all its subexpressions are computed, and the LL(1) property of the Syntax is checked. In case the syntax is not LL(1), by default an exception is thrown. By setting enforceLL1 to false, the factory object will not throw any exception, but the resulting Parser will behave as an under-approximation of the argument Syntax.

6.2.5 Properties

Properties of a Syntax, such as nullability and first set, are computed during the Parser creation process using propagation networks and are stored as part a distinct datatype which mirrors the given Syntax datatype. This mirrored datatype is not directly visible to users of the library, but is used extensively by the parsing algorithm.

For convenience, the properties associated with a Syntax are also made available to users of the library through a Properties object.

```
case class Properties[A](
    nullable: Option[A],
    first: Set[Kind],
    shouldNotFollow: Set[Kind],
    conflicts: Set[Conflict]) {

  def isNullable: Boolean = nullable.nonEmpty
  def isProductive: Boolean = isNullable || first.nonEmpty
  def isLL1: Boolean = conflicts.isEmpty
}
```

Thanks to an implicit conversion from `Syntax[A]` to `Properties[A]`, users of the library can call methods of the `Properties` class directly on `Syntax` objects.

```
implicit def syntaxToLL1Properties[A](syntax: Syntax[A]): Properties[A] = {
  if (!syntaxToPropertiesCache.containsKey(syntax)) {
    Parser(syntax, enforceLL1=false)  // Triggers the computation of properties.
  }
  syntaxToPropertiesCache.get(syntax).asInstanceOf[Properties[A]]
}
```

For efficiency, the properties associated with a syntax are stored in a reference-associated cache.

6.2.6 LL(1) Conflicts

As stated earlier, SCALL1ON performs LL(1) checks during the `Parser` creation phase, right after the computation of properties. In case of LL(1) conflicts, a `ConflictException` is thrown.

```
case class ConflictException(conflicts: Set[Conflict])
  extends Exception("Syntax is not LL(1).")
```

The exception contains a set of `Conflict` objects, each of which can take one of three forms: `NullableConflict`, `FirstConflict`, and `FollowConflict`.

```
case class NullableConflict(
  source: Disjunction[_]) extends Conflict

case class FirstConflict(
  source: Disjunction[_],
  ambiguities: Set[Kind]) extends Conflict

case class FollowConflict(
  source: Disjunction[_],
  root: Sequence[_, _],
  ambiguities: Set[Kind]) extends Conflict
```

Each `Conflict` object contains information about the cause of the conflict. `NullableConflict` contains a reference to a `Disjunction` with two nullable branches within the `Syntax`. `FirstConflict` contains reference to a `Disjunction` within the `Syntax` and a non-empty set of `Kinds`. The non-empty set of kinds is a subset of the first set of both branches of the provided disjunction. In the case of a `FollowConflict`, references to both a `Disjunction` and a `Sequence` subexpressions are given, as well as a non-empty set of `Kinds`. The non-empty set of kinds is a subset of both the should-not-follow set of the left side of the `Sequence` as well as of the first set of right side of the `Sequence`. In that case, the `Disjunction` given is a subexpression of the left branch of the `Sequence` that caused the provided set of `Kinds` to be part of the should-not-follow-set of the left-side of the `Sequence`; one of the branch of the `Disjunction` is nullable, while the other has a first set which is a superset of the provided set of ambiguous `Kinds`.

SCALL1ON provides utilities to help users understand and debug LL(1) conflicts. While `Conflict` objects are in theory sufficient to locate the root causes of conflicts, they are unfortunately not user-friendly. The reason is that `Conflict` objects contains information about low-level details of the graph structure of `Syntaxes`, while users of the library operate at a higher abstraction level: that of combinators. Forcing users to reason about graphs of `Syntax`-constructor nodes in order to debug conflicts would go against the mental model of users, and would break the abstraction provided by combinators. Therefore, some efforts are made to communicate conflicts in terms that users can easily relate to.

By calling the `debug` function, users of the library are provided with a *LL(1) Conflicts Report* about their `Syntax`. In Listing 24, I present a conflict report obtained on an example JSON parser, which contains an error on line 205.

```
204  val arrayValue: Syntax[Value] =
205    ('[' ~ repsep(value, ',') ~ '[').map {
206      case _ ~ vs ~ _ => ArrayValue(vs)
207    }
```

In this example, the token indicating the end of the array has been erroneously replaced by an array *opening* token, leading to a LL(1) conflict. The report of Listing 24 correctly identifies this and presents to the user a stack trace which pinpoints the cause of the conflict.

In addition to the stack trace, a small number of sequences of tokens kinds are also shown as part of the report. Such sequences of kinds show ways to arrive at the conflict. In this particular instance, the conflict can be exhibited after a single `[` token. Indeed, at that point, an extra `[` token could either indicate a new array opening, or (erroneously) close the current array.

The two additional ways of presenting conflicts are more in line with the mental model of users. Stack traces refer to explicit locations in the parser's code, while enumerated sequences of kinds indicate where the conflict occurs semantically.

```
=== LL(1) Conflicts Report ===

The syntax is not LL(1).

A single conflict has been found:

--- Conflict 1/1 ---

First/Follow conflict.

The left branch of a sequence can stop or
continue on the same token than the right side can start with.

The ambiguous token kind is [.

The source of the conflict can be traced to:

  scallion.Syntaxes$Syntax.$init$(Syntaxes.scala:116)
  scallion.Syntaxes$Syntax$Sequence.<init>(Syntaxes.scala:432)
  scallion.Syntaxes$Syntax.$tilde(Syntaxes.scala:249)
  scallion.Syntaxes$Syntax.$tilde$(Syntaxes.scala:245)
  scallion.Syntaxes$Syntax$Sequence.$tilde(Syntaxes.scala:431)
  example.json.JSONParser$.<init>(JSON.scala:205)
  example.json.JSONParser$.<clinit>(JSON.scala)
  $line11.$read$$iw$$iw$$iw$$iw$$iw$$iw$.<init>(<console>:18)
  $line11.$read$$iw$$iw$$iw$$iw$$iw$$iw$.<clinit>(<console>)
  $line11.$eval$.$print$lzycompute(<console>:7)
  $line11.$eval$.$print(<console>:6)
  $line11.$eval.$print(<console>)

The following sequences lead to an ambiguity when followed by a token of kind [:

  (1) [
  (2) { <string> : [
  (3) { <string> : { <string> : [
  (4) { <string> : <boolean> , <string> : [
  (5) { <string> : <number> , <string> : [
```

Listing 24 – Example LL(1) Conflict Report

6.2.7 Parsing

Once a `Syntax[A]` has been checked to be LL(1) and has been converted into a proper `Parser[A]` object, that object can be used for actual parsing. Objects of type `Parser[A]` offer an `apply` method to process an iterator of tokens into a value of type `ParseResult[A]`. Parse results can take on of three forms:

```scala
case class Parsed[A](
  value: A,
  rest: Parser[A]) extends ParseResult[A]

case class UnexpectedToken[A](
  token: Token,
  rest: Parser[A]) extends ParseResult[A]

case class UnexpectedEnd[A](
  rest: Parser[A]) extends ParseResult[A]
```

A result of type `Parsed` indicates a successful parse. In that case, the produced parse `value` is returned. The two other constructors indicate parse errors. `UnexpectedToken`, as the name suggests, indicates that an unexpected token has been encountered during parsing. The incriminated `token` is returned. In that case, the input iterator of tokens is not consumed any further than that token. Finally, a result of type `UnexpectedEnd` indicates that the input ended at a point when the parser could not, or no longer, accept.

All three constructors contain a reference to a *residual parser* in their `rest` field. This residual parser encodes the state of the original parser after the successful parse, respectively at the point of error. This parser can be extremely useful to provide users of the parser with ways to understand and fix their input text. For instance, the `first` set of the residual parser can be queried to indicate valid ways to continue the program. Coupled with the enumeration capabilities of SCALL1ON, residual parsers can also be used to provide features such as code completion or error recovery. Residual parsers can also be used to process input tokens in a non-blocking fashion: Parsers can be stored and invoked at later points to process newly available tokens.

6.3 Enumeration

The deep embedding of context-free expressions used by SCALL1ON enables features that require inspection of expressions and that are thus typically not found in parsing combinators library that employ shallow embeddings. The availability of methods to compute properties of expressions, such as productivity, nullability, first sets, and LL(1), is one example. *Enumeration* is an other example.

In SCALL1ON, the sequences of tokens *kinds* that a `Syntax` describes can be iterated over. In its simplest from, the `enumerate` function accepts a single argument `Syntax` and returns a lazy iterator over sequences of kinds described by the syntax. The returned iterator produces sequences of kinds in increasing order. The signature of the `enumerate` function is shown in Listing 25. An example invocation of the function on the JSON syntax is given in Listing 26.

```
def enumerate(syntax: Syntax[_]): Iterator[Iterator[Kind]]
```

Listing 25 – Simple enumeration function from SCALL1ON.

```
scala> enumerate(jsonSyntax).take(20).foreach(it => println(it.mkString))
<boolean>
<number>
<string>
<null>
[]
{}
[<boolean>]
[<number>]
[<string>]
[<null>]
[[]]
[{}]
[[<boolean>]]
[[<number>]]
[[<string>]]
[[<null>]]
[<boolean>,<boolean>]
[<number>,<boolean>]
[<string>,<boolean>]
[<null>,<boolean>]
```

Listing 26 – Example execution of the `enumerate` function on the JSON syntax.

6.3.1 Markings

A second version of the function offers the possibility to specify a function `kindFunction` to apply on kinds, and a function to optionally apply on `Marked` nodes instead of visiting them. The signature of the function is shown in Listing 27.

```
def enumerate[A](
  syntax: Syntax[_],
  kindFunction: Kind => A)
  (markFunction: PartialFunction[Mark, A]): Iterator[Iterator[A]]
```

Listing 27 – Richer interface to the enumeration function.

This enumeration function is useful in practice to abstract over sequences of tokens. Instead of unconditionally visiting the inner syntax of `Marked` nodes, the `markFunction` is called on the associated `Mark`. In case the function is defined on the `Mark`, the returned value is emitted. Otherwise, the inner syntax is visited.

6.3.2 Implementation

In SCALL1ON, enumeration is implemented in a style that is reminiscent of the *propagation networks* used for computing properties. Using propagation networks unchanged would unfortunately potentially lead to non-termination, as the information propagation phase could run forever. Indeed, potentially infinitely many sequences of kinds or values can be enumerated. To handle this, the propagation phase is broken up and lazily executed. Only sequences up until a fixed size are enumerated. The size limit is local to each sub-syntax. When all sequences up until a certain size have been enumerated, a downwards phase updates the size limits in the sub-syntaxes, leading to the resumption of the propagation phase. In addition, a mechanism is put in place to detect when all sequences have been enumerated. The size limit is not global as smaller limits can be used for sides of sequences.

6.4 Pretty Printing

SCALL1ON, in addition to parsing, also support pretty printing. While parsing converts a sequence of tokens into a value, pretty printing is the inverse: Given a value, the goal of a pretty printer is to return a sequence of tokens that describes the value.

To support pretty printing, SCALL1ON's `Transform` `Syntax` constructor accepts a (partial) inverse of the function applied during parsing. In practice, users of the library that wish to obtain a pretty printer along with their parser must add an extra argument in some combinators of SCALL1ON, notably `map`. Most derived combinators, such as `many`, work out of the box with pretty printing.

Given appropriate partial inverses, pretty printing is guaranteed to return a sequence of tokens that can be parsed back to the same value. Under a set of reasonable assumptions, the pretty printing procedure is guaranteed to run in time linear in the number of output tokens.

The first assumption is that the syntax does not contain a `Recursive` node that contains a reference to itself without additional context. Without such nodes, between each visit of a `Recursive` node within a call to the pretty printer a strictly positive number of tokens is bound to be produced. Non-left-recursive syntaxes do not contain such problematic `Recursive` nodes. As LL(1) syntaxes can not be left-recursive, the LL(1) syntaxes of SCALL1ON are guaranteed not to contain any such nodes.

The second assumption is that `Disjunction` and `Transform` nodes do not create competing alternatives. Alternatives should either fail fast, or end up producing the resulting tokens. This ensures a single logical thread of execution within the pretty printing procedure. I will discuss some ways one can ensure this holds by construction.

6.4.1 Basic Algorithm

Listing 28 presents a simplified implementation of the pretty printing procedure found in SCALL1ON. The simple implementation presented here can be subject to stack overflows and is rather inefficient for reasons I will discuss. The complete optimised implementation found in the library will be discussed afterwards. In the interest of the presentation, I discuss the simplified version first.

The pretty printing function, `pretty`, is given three arguments: a syntax description, a value to be pretty printed, and an `entered` set, whose purpose is to avoid certain types of infinite loops. The function returns an optional `Tree` of tokens. A return value of `None` indicates that the value can not be produced by the given syntax. In the other case, the tokens returned are guaranteed to be of minimal length and to yield the pretty printed value when parsed. I discuss termination of the pretty printing function later in this section.

Note that those properties rely on the correctness of the local inverses. When the local inverses are incomplete, the minimality and completeness of the pretty printer are not guaranteed. Additionally, when the local inverses are unsound, the property that the returned sequence parses back to the original value is no longer guaranteed. By default, SCALL1ON provides an empty inverse function as the default argument of all operators that accept an optional inverse. This default is sound but potentially incomplete.

The `Tree` datatype returned by `pretty` represents a sequence of values implemented as an immutable binary tree. This representation allows for constant time concatenation. In addition, a field containing the size of the tree is maintained. This `Tree` datatype is internal to SCALL1ON and not visible to users of the library. In the actual library, the pretty printing function returns an iterator over the tokens of the tree instead of the actual `Tree`.

```scala
def pretty[A](
    syntax: Syntax[A],
    value: A,
    entered: Set[(RecId, Any)] = Set()): Option[Tree[Token]] =
  syntax match {
    case Success(other) => if (value == other) Some(Empty) else None
    case Failure() => None
    case Elem(kind) => if (getKind(value) == kind) Some(Node(value)) else None
    case Disjunction(left, right) =>
      (pretty(left, value, entered), pretty(right, value, entered)) match {
        case (leftRes, None) => leftRes
        case (None, rightRes) => rightRes
        case (Some(leftTree), Some(rightTree)) =>
          if (leftTree.size <= rightTree.size) Some(leftTree) else Some(rightTree)
      }
    case Sequence(left, right) =>
      val leftValue ~ rightValue = value
      pretty(left, leftValue, entered).flatMap { leftTree =>
        pretty(right, rightValue, entered).map { rightTree =>
          leftTree ++ rightTree
        }
      }
    case Transform(_, inverse, inner) =>
      inverse(value).flatMap(pretty(syntax, _, entered)).minByOption(_.size)
    case Marked(_, inner) => pretty(inner, value, entered)
    case Recursive(_, inner) =>
      if (entered.contains((id, value))) {
        None
      } else {
        pretty(inner, value, entered + ((id, value)))
      }
  }
```

Listing 28 – Simplified implementation of the pretty-printing algorithm.

The `pretty` function matches the argument `syntax` against the argument `value`:

- In case the `syntax` is `Success(other)`, the `value` is checked against `other` value found in the `Success` constructor. In case the two values are equal, the empty sequence of tokens is returned. In case they do not match, no sequence is returned.
- In case of `Failure()`, no possible representation of the `value` following the `syntax` exist, and so no sequence is returned.
- In case of `Elem(kind)`, the `value` to be pretty printed is guaranteed to be a `Token` by the type system. If that token is of the appropriate `kind`, the singleton sequence consisting of that token is returned. In case the token is not of the expected `kind`, no sequence is returned.
- In case of `Disjunction(left, right)`, both sides are queried and the sequence of minimal length, if any, is returned.
- In case of `Sequence(left, right)`, the `value` is bound to be a pair of two values, `leftValue` and `rightValue`. Each side is queried with their respective value and the two resulting sequences, if any, are concatenated. As an optimisation, the right side is only queried when the left call is successful.
- In case of a `Marked` syntax, the call is simply forwarded to the inner syntax.
- Last is the `Recursive` syntax case. The `entered` argument finally plays a role here. In case the combination of the recursive syntax's identifier and argument value has not been encountered in the call stack yet, the call is forwarded to the inner syntax and the combination is recorded. Otherwise, the call stops and no sequence is returned.

  The check performed in this case prevents a certain type of infinite loops that would otherwise frequently happen in practice. For instance, consider a syntax for expressions with support for parentheses. Such parenthesis are generally only materialised in the syntax, but are typically not reflected in the abstract syntax tree. In this syntax, when an expression is pretty printed, the disjunct that encodes the parenthesised syntax will be queried, which in turn will invoke the recursive expression syntax on the same exact value. Without this check, the recursion is bound to loop.

  Note that this check does not impact the completeness of the pretty printer. Indeed, if a syntax associates a certain sequence of tokens with a given value, it is guaranteed that there exists a smallest derivation of that fact. As it is smallest, that derivation may not contain a derivation with the given fact as a subtree. A similar argument is made to show that the check does not impact the minimality of the returned sequence.

6.4.2 Termination

Note that, even with the previously mentioned mitigation strategy put in place, the pretty printing function is not guaranteed to terminate, even in case of sound and complete inverses.

The check performed in the Recursive case is unfortunately not sufficient to prevent all infinite loops. Indeed, the check only prevents a Recursive syntax to be repeatedly queried with the *same* value.

Consider the example of a misbehaved syntax of Listing 29. When called on the misbehaved syntax and any BigInt n, the pretty printing function will end up in a non-terminating loop. Indeed, the misbehaved recursive syntax is first queried on n, which leads to a query on base and n, and to a recursive query on misbehaved and $n+1$. This recursive query leads to a query on base and $n+1$, and misbehaved and $n+2$, and so on. The looping behaviour is however logically justified in light of the minimality constraint. It is a priori impossible to know for which n the base syntax will return the smallest sequence of tokens.

```scala
val base: Syntax[BigInt] = ??? // Any Syntax

val misbehaved: Syntax[BigInt] = recursive {
  base | misbehaved.map({
    case n => n - 1
  }, {
    case n => Seq(n + 1)
  })
}
```

Listing 29 – Example Syntax on which pretty-printing does not terminate.

Note that the misbehaved syntax of Listing 29 exhibits left-recursion, and as such is not LL(1). As shown in Listing 30, the syntax can nevertheless be adapted to be LL(1) and still remain problematic for the pretty printer. Assuming the base syntax is LL(1) and does not contain SeparatorKind("-") in its first set, the syntax misbehaved is LL(1).

```scala
val base: Syntax[BigInt] = ??? // Any LL(1) Syntax

val misbehaved: Syntax[BigInt] = recursive {
  base | (elem(SeparatorKind("-")) ~ misbehaved).map({
    case _ ~ n => n - 1
  }, {
    case n => Seq(SeparatorToken("-") ~ n + 1)
  })
}
```

Listing 30 – Example LL(1) Syntax on which pretty-printing does not terminate.

Even when the syntax is LL(1), one does not a priori know for which index n the sequence will be of minimal length. In theory, the LL(1) constraint at least ensures that each iteration of the recursive loop adds a non-zero number of prefix tokens to the resulting sequence, assuming sound inverses. This fact could in theory be exploited to stop the loop when the number of prefix tokens exceeds the number of tokens of an already found valid sequence. This is currently not implemented in SCALL1ON, as other ways of circumventing the problem exist, but could be envisioned as future work.

In practice, this non-terminating behaviour is however rarely encountered. Users of SCALL1ON can prevent this problematic behaviour by slightly modifying their syntaxes, and so in a way that does not interfere with the parsing behaviour. I will discuss some patterns to do so in the later sections.

6.4.3 Linear-Time Pretty Printing

As shown in the previous few paragraphs, the pretty printing function is not always guaranteed to terminate. Using similar techniques, one could devise syntaxes and values on which the pretty printer performs an arbitrary number of execution steps and results in an arbitrarily small number of tokens. It is therefore pointless to discuss the worst-case complexity of the pretty printing function in the general case.

In practice however, the pretty printer often behaves linearly in the number of returned tokens, especially in case of LL(1) syntaxes. In this section, I examine conditions which guarantee linearity of the pretty printer. In the interest of simplicity, I will assume that applying user-defined inverses takes constant time. In this simple presentation, I will also ignore the cost of set operations on the `entered` set. Finally, I will make the simplifying assumption that the initial input `Syntax` to the pretty printer contains only a single `Recursive` expression. The argument can be adapted to multiple such nodes, but becomes more intricate.

A first observation to be made is that, in any invocation of the pretty printer in which no `Recursive` syntax is recursively reentered (regardless of the pretty printed value) each node is guaranteed is be visited only finitely many times. Furthermore, assuming that `Transform` inverses returns a bounded number of pre-images regardless of the input, the number of paths to each node in the `Syntax` is also bounded, and so regardless of the value to be pretty printed. In this case, one can thus bound the runtime of the pretty printing function by a number K that only depends on the argument `Syntax`. For our purposes, I will assume a fixed initial `Syntax`, and thus will consider this number K to be constant.

In practice, requiring `Recursive` syntaxes not to be reentered is too strong of a requirement. This stringent requirement can be relaxed thanks to a second observation: if a `Recursive` syntax is visited multiple times, and between each re-visit a non-zero number of tokens of the resulting pretty printed sequence can be credited to this path, then the running time of the pretty printer is guaranteed to be linear. Indeed, the work done in between two visits of a `Recursive` syntax is at most a constant K (from first observation). If the exact non-zero tokens produced in between two visits of the `Recursive` syntax end up in the resulting pretty printed sequence, then the time K taken in between the two visits can be credited for the output of at least one token. Since the time taken before the first and after the last visit of the `Recursive` syntax is constant, then the total running time of the pretty printing function is linear in the number of outputted tokens.

Interestingly, in the case of LL(1) syntaxes, at least one token is to be produced in between two visits. Indeed, the syntax would otherwise be left-recursive and exhibit a conflict. This partially fulfils the condition outlined by the second observation.

Crucially, the second observation requires that the tokens produced in between two visits of a `Recursive` syntax are included as part of the resulting pretty printed sequence. If, for some reason, the tokens produced were to be discarded, then the linearity argument would not hold. It is therefore essential that paths that end up visiting arbitrarily many times a `Recursive` syntax are not competing against other valid alternatives. In practice, this property seems to be often respected. For instance, Listing 31 shows a syntax describing sequences of numbers. Note that the `numbers` syntax could be written using the `many` combinator from Table 6.3. I argue that any list of `Int` can be pretty printed in time linear in the number of output tokens.

```
val number: Syntax[Int] = accept(NumberKind)({
  case NumberToken(n) => n
}, {
  case n => Seq(NumberToken(n))
})

// Equivalent to "val numbers = many(number)"
lazy val numbers: Syntax[List[Int]] = recursive {
  epsilon(Nil) |
  (number ~ numbers).map({
    case n ~ ns => n :: ns
  }, {
    case n :: ns => Seq(n ~ ns)
    case _ => Seq()
  })
}
```

Listing 31 – Example well-behaved LL(1) syntax for pretty printing.

While queried many times (once per element of the input list, and once for the empty list) the recursive `numbers` is not in competition with any other alternative: Looking at a list of numbers, it is immediately clear which of the two alternatives presented by `numbers` is to be taken. If the list is `Nil`, only the left alternative succeeds. If the list is a `Cons`, only the right alternative may succeed. In this case, the pretty printing function takes time linear in the number of output tokens.

In contrast, consider the ill-behaved example of Listing 32. Looking at a non-empty list of numbers, it is impossible to decide which of the last two alternatives produced the sequence. Both alternatives therefore have to be tried, leading to an exponential blow-up. Notice that, in this case, the exponential blow-up can be avoided by some sort of left-factoring: writing the `numbersOfBits` syntax as `many(number | bits)` completely avoids this issue.

```
val number: Syntax[Int] = accept(NumberKind)({
  case NumberToken(n) => n
}, {
  case n => Seq(NumberToken(n))
})

val bits: Syntax[Int] = accept(BitsKind)({
  case BitsToken(n) => n
}, {
  case n => Seq(BitsToken(n))
})

lazy val numbersOrBits: Syntax[List[Int]] = recursive {
  epsilon(Nil) |
  (number ~ numbersOrBits).map({
    case n ~ ns => n :: ns
  }, {
    case n :: ns => Seq(n ~ ns)
    case _ => Seq()
  }) |
  (bits ~ numbersOrBits).map({
    case b ~ ns => b :: ns
  }, {
    case b :: ns => Seq(b ~ ns)
    case _ => Seq()
  })
}
```

Listing 32 – Example ill-behaved LL(1) syntax for pretty printing.

6.4.4 Syntax Disambiguation

In the last few paragraphs, I examined a set of conditions under which pretty printing is guaranteed to run in time linear in the number of produced tokens. One condition has to do with pretty-printing ambiguities: In case of disjunctions, both branches are queried but only a single contributes to the pretty printed output. This behaviour can be problematic for linearity. The same problematic behaviour can arise with `Transform` nodes, when the inverse function returns more than one result. When all but at most one alternatives fail in constant time, the syntax does not display ambiguities. In this case, logically only at most one single branch is further explored by the pretty-printing procedure, paving the way for linear-time pretty printing.

Users of SCALL1ON have tools at their disposition to disambiguate syntaxes for pretty printing purposes. SCALL1ON users can use parsing-neutral combinators to influence the behaviour of the pretty printer without altering its parsing behaviour. Notably, users can use *tagged disjunctions*, noted s_1 `||` s_2 (see Table 6.3), to reduce non-determinism in the pretty printer.

When both s_1 and s_2 are of type `Syntax[A]`, values of the tagged union s_1 `||` s_2 are of type `Either[A, A]`, which indicates which side of the disjunction the values come from. One can recover a unique value of type A using the `map` combinator:

```
val example: Syntax[Int] = (number || bit).map({
  case Left(n) => n
  case Right(b) => b
}, {
  case n => Seq(Left(n))  // Left-biased.
})
```

Listing 33 – Disambiguation using tagged unions.

By only using the tagged disjunction e_1 `||` e_2 instead of normal disjunctions, one ensures that `Disjunction` nodes do not create multiple competing alternatives. Furthermore, by only allowing inverse functions of `Transform` nodes to return at most one pre-image for each input, one ensures that `Transform` nodes do not introduce multiple competing alternatives.

Interestingly, those restrictions do not influence parsing at all. Any `Syntax` can be rewritten to eliminate untagged disjunctions, using the pattern shown in Listing 34. The transformation does not introduce LL(1) conflicts.

```
def or[A](left: Syntax[A], right: Syntax[A])
    (direction: PartialFunction[A, Boolean]): Syntax[A] =
  (left || right).map({
    case Left(x) => x
    case Right(x) => x
  }, {
    case y => direction.lift(y) match {
      case None => Seq()
      case Some(true) => Seq(Left(y))
      case Some(false) => Seq(Right(y))
    }
  })
```

Listing 34 – Pretty-printing friendly disjunction function. The parsing behaviour of `or(`e_1`,` e_2`)` is the same as e_1 `|` e_2. For pretty printing, the `direction` function argument indicates which side of the disjunction is responsible for pretty printing the given value.

The use of the `or` function from Listing 34 asks of users to provide a `direction` function for each disjunction, which may be somewhat unpractical.

Using the `up` method of Table 6.3 in conjunction with untagged disjunctions can also provide a way to ensure that disjunctions do not introduce competing pretty printing alternatives. Thanks to reflection capabilities of Scala, `up` discards values during pretty printing that do not correspond to the particular original subtype. Disjunctions of `up`'d syntaxes with disjoint base types is guaranteed not to introduce competing alternatives during pretty printing. An example of the pattern is given in Listing 35.

```
val boolSyntax: Syntax[BoolValue] = ...
val numberSyntax: Syntax[NumberValue] = ...
val stringSyntax: Syntax[StringValue] = ...
val nullSyntax: Syntax[NullValue]
val arraySyntax: Syntax[ArrayValue] = ...
val objectSyntax: Syntax[ObjectValue] = ...
val jsonSyntax: Syntax[Value] =
  boolSyntax.up[Value] |
  numberSyntax.up[Value] |
  stringSyntax.up[Value] |
  nullSyntax.up[Value] |
  arraySyntax.up[Value] |
  objectSyntax.up[Value]
```

Listing 35 – Example showing the pattern of combining `up` and untagged disjunctions to guarantee non-competing alternatives during pretty printing.

While theoretically one could statically check those conditions in order to show a pretty printer linear, in the present implementation in SCALL1ON, it is unfortunately impossible to automatically and statically detect possible pretty printing ambiguities. In contrast, SCALL1ON's LL(1) checks ensure that parsing is well-behaved. Providing the same kind of guarantees for pretty printing is interesting future work, with many interesting practical challenges. The techniques mentioned in this section provide research direction worth exploring.

6.4.5 Optimised Implementation

The pretty printer implementation discussed so far in this section is unfortunately not practical. Due to the use of non-tail-recursive functions, the pretty printer can, and does, exhibit stack overflows on relatively large inputs. Furthermore, the use of a Scala `HashSet` to record already visited pairs of `Recursive` syntax and value is problematic, as it means that potentially large values end up being checked for equality. As a result of this, the pretty printer becomes sluggish in practice.

The implementation presented in Listing 36 addresses those issues. The code is refactored in a callback-driven style to avoid having unbounded use of the call stack. A `stack` data structure maintains a list of explorations of syntax and value pairs to perform, while a priority `queue` records productions of trees of tokens to execute, ordered by increasing size of token sequences. The priority policy of the queue ensures that parallel queries, for instanced triggered upon visit of a `Disjunction` node, are resolved in order of resulting sequence size. This in turn ensures that the only the smallest sequence is further propagated.

The local `entered` sets of the unoptimised implementation are replaced by a global `recs` map. The mutable recs map is indexed by Recursive identifiers. To each such identifier is assigned a unique mutable IdentityHashMap, indexed by values. IdentityHashMap is a reference-indexed hash map, which avoids the costly comparison on values encountered in the unoptimised version. While reference equality is often stricter than user-provided equality on values, such an equality is sufficient to handle the types of recursive loops that arise in practice. The IdentityHashMap associates to each encountered value a way to be notified of produced token sequences. Upon the first visit of a value in a Recursive object, an entry is added to the associated IdentityHashMap and the inner syntax is queried. Upon revisit, a callback is registered to be called if and when a sequence is produced.

Having a global map instead of the local recs is also useful to avoid re-traversing the same syntax with the same argument in alternatives, as would for instance occur in the ill-behaved example of Listing 32.

```
def pretty[A](syntax: Syntax[A], value: A): Option[Iterator[Token]] = {
  var stack: List[() => Unit] = Nil
  val queue: PriorityQueue[(Int, () => Unit)] =
    PriorityQueue.empty(Ordering.by(-_._1))
  val recs: LongMap[IdentityHashMap[Any, ((Tree[Token]) => Unit) => Unit]] =
    new LongMap()

  def thenGo[B](
      syntax: Syntax[B],
      value: B,
      subscriber: Tree[Token] => Unit): Unit = {
    stack +:= (() => go(syntax, value, subscriber))
  }

  def go[B](
      syntax: Syntax[B],
      value: B,
      subscriber: Tree[Token] => Unit): Unit = {

    def record(tree: Tree[Token]): Unit = {
      queue += ((tree.size, () => subscriber(tree)))
    }

    syntax match {
      case Success(other) => if (value == other) record(Empty)
      case Failure() => ()
      case Elem(kind) => if (getKind(value) == kind) record(Node(value))
      case Disjunction(left, right) => {
        var sent = false
        val update: Tree[Token] => Unit = (tree: Tree[Token]) => {
          if (!sent) {
            sent = true
            record(tree)
          }
        }
    thenGo(left, value, update)
    thenGo(right, value, update)
  }
  case Sequence(left, right) => {
    val leftValue ~ rightValue = value
    val leftUpdate: Tree[Token] => Unit = (leftTree: Tree[Token]) => {
      val rightUpdate: Tree[Token] => Unit = (rightTree: Tree[Token]) => {
        record(leftTree ++ rightTree)
      }
      thenGo(right, rightValue, rightUpdate)
    }
    thenGo(left, leftValue, leftUpdate)
  }
  case Transform(_, inv, inner) => {
    var sent = false
    val update: Tree[Token] => Unit = (tree: Tree[Token]) => {
      if (!sent) {
        sent = true
        record(tree)
      }
    }
    for (invValue <- inv(value)) {
      thenGo(inner, invValue, update)
    }
  }
  case Marked(_, inner) => {
    thenGo(inner, value, subscriber)
  }
  case Recursive(id, inner) => {
    val idMap = recs.getOrElseUpdate(id, new IdentityHashMap())
    if (!idMap.containsKey(value)) {
      val subscribers = new Queue[Tree[Token] => Unit]()
      subscribers += subscriber
      var cache: Option[Tree[Token]] = None
      def addSub(subscriber: Tree[Token] => Unit): Unit =
        cache match {
          case None => subscribers += subscriber
          case Some(tree) => queue += ((tree.size, () => subscriber(tree)))
        }
      def update(tree: Tree[Token]): Unit = {
        cache = Some(tree)
        subscribers.foreach { subscriber =>
          queue += ((tree.size, () => subscriber(tree)))
        }
      }
      idMap.put(value, addSub)
      thenGo(inner, value, update)
    }
    else {
      val addSub = idMap.get(value)
      addSub(subscriber)
    }
```

```
      }
    }
  }

  var res: Option[Tree[Token]] = None

  def updateRes(tree: Tree[Token]): Unit = {
    res = Some(tree)
  }

  go(syntax, value, updateRes)

  while (res.isEmpty && (queue.nonEmpty || stack.nonEmpty)) {
    while (stack.nonEmpty) {
      val next = stack.head
      stack = stack.tail
      next()
    }

    if (queue.nonEmpty) {
      val (_, next) = queue.dequeue()
      next()
    }
  }

  res.map(_.values)
}
```

Listing 36 – Optimised pretty printing procedure.

6.5 Performance Evaluation

In this section, I evaluate the performance of the SCALL1ON library against several competing approaches. The first two benchmarks measure the performance of various approaches on parsing JSON files. JSON is a relatively simple and widely adopted format.
Interestingly, the JSON language is *context-free* and *non-regular*, making it too complex to handle by simpler techniques such as deterministic finite-state automaton. Additionally, the JSON language is also LL(1). While simple, JSON is not trivial to parse, as shown by a recently discovered performance bug (t0st, 2021).
Measurements appearing in this section were done on a 2018 MacBook Pro with a 2.2 GHz Intel Core i7 processor. I used Java 1.8 and Scala 2.12.13 running on top of the Java HotSpot™ virtual machine. Each entry corresponds to the mean of 36 runs on a hot virtual machine. Measurements were done using ScalaMeter (Prokopec, 2019).

6.5.1 Parsing JSON

In a first set of benchmarks, I measured the performance of three different approaches on JSON Parsing on files ranging from 100KB to 10MB. Table 6.5 shows the results. The first approach, labeled *Simple*, implements the LL(1) *parsing with derivatives* approach *without the zipper-based representation of expressions* presented in Chapter 4. The second approach, labeled *Zipper*, is a SCALL1ON JSON parser. SCALL1ON uses the *parsing with derivatives and zippers* algorithm presented in Chapter 5. The last approach, labeled *SPC*, is a JSON parser implemented using the *Scala Parser Combinators* library (LAMP EPFL and Lightbend, Inc, 2019). In all cases, parsers are directly given tokens produced by the same lexer. Lexing time is not reported.

File size (KB)	Tokens	Parse time (ms)			Speed (token/ms)		
		Simple	Zipper	SPC	Simple	Zipper	SPC
100	9649	99.9	2.8	2.3	96.6	3446.0	4195.2
1000	97821	7069.2	14.3	19.0	13.8	6840.6	5159.3
10000	971501	†	150.2	166.0	†	6468.0	5852.4

Table 6.5 – Performance comparison between simple LL(1) *parsing with derivatives* (Simple), LL(1) *parsing with derivatives and zippers* as implemented in SCALL1ON (Zipper), 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.

The performance of the LL(1) parsing with derivatives and zippers implemented by SCALL1ON is comparable to the performance of the recursive descent algorithm implemented by the Scala Parser Combinators library. Worth noting, SCALL1ON does not 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 my tests). SCALL1ON also offers more comprehensive error reporting and recovery, in part thanks to encoding of parser states as explicit analysable expressions.
I also benchmarked the performance of Parseback (Spiewak, 2018), a recent Scala implementation of the *parsing with derivatives* algorithm (Might et al., 2011) by one of the original authors, with performance optimisations from later works (Adams et al., 2016). The results are not reported in Table 6.5 as the parser encounters a stack overflow in each of the benchmarks. The largest file I managed to parse with using Parseback was 1387 bytes long, and it took 1388ms.

6.5.2 Lexing and Parsing JSON

In the second set of benchmarks, I measured the performance of lexer and parser pair implemented using SILEX and SCALL1ON against an ANTLR-generated JSON lexer and parser (Parr, 2013; Parr and Fisher, 2011; Parr, 2019), and two lexerless JSON parsers respectively implemented using FastParse2 (Haoyi, 2021) and the Scala Parser Combinators library (LAMP EPFL and Lightbend, Inc, 2019). All implementations were run on the same JSON input files as the previous benchmarks.

Comparison Against ANTLR

In this subsection, I report the performance of the SILEX and SCALL1ON solution against that of ANTLR using both its code-generator mode and its interpreter mode. The results are reported in Table 6.6.

File size (KB)	Tokens	Lex & parse time (ms)		
		SILEX + SCALL1ON	ANTLR (Int.)	ANTLR (Gen.)
100	9649	7.3	9.6	1.9
1000	97821	33.4	33.1	15.8
10000	971501	344.7	239.9	145.9

Table 6.6 – Performance of a JSON lexer and parser implemented using the SILEX and SCALL1ON libraries (SILEX + SCALL1ON) compared to an ANTLR-generated lexer and parser (ANTLR (Int.)) and an interpreted ANTLR lexer and parser (ANTLR (Gen.)).

The implementation using SILEX and SCALL1ON, which does not use code generation, is only a small constant factor (~3) slower than the ANTLR-generated implementation and displays performance similar to that of ANTLR in interpreter mode.
Compared to ANTLR, both SILEX and SCALL1ON offer a highly flexible and extensible interface embedded in a rich programming language, while ANTLR grammars are described in a domain-specific language of limited expressivity. In addition, while SCALL1ON directly builds values of the appropriate type, extra work must be done to convert parse trees produced by the ANTLR parsers into proper user-defined JSON values. This extra work is not reported in the results.

Comparison Against Lexerless Parser-Combinators

In this subsection, I report the performance of the SILEX and SCALL1ON solution against that of two lexerless parsers implemented using FastParse2 and the Scala Parser Combinators (SPC) library. Being lexerless, the two implementations directly integrate lexical aspects in the definition of the parser, while the SILEX and SCALL1ON solution features a separate lexer. Note that the SPC-based implementation differs from that of the earlier benchmarks since it operates on character tokens instead of higher level tokens. The results are reported in Table 6.7.

File size (KB)	Tokens	Lex & parse time (ms)		
		SILEX + SCALL1ON	FastParse2	SPC (lexerless)
100	9649	7.3	2.6	21.0
1000	97821	33.4	21.6	189.9
10000	971501	344.7	196.5	1959.5

Table 6.7 – Performance of a JSON lexer and parser implemented using the SILEX and SCALL1ON libraries (SILEX + SCALL1ON) compared to a lexerless FastParse2 parser (FastParse2) and a lexerless Scala Parser Combinators parser (SPC (lexerless)).

The implementation using SILEX and SCALL1ON, which does not use code generation, is only a small constant factor (~3) slower than the FastParse2 parser but is about 3-to-5 times faster than the SPC parser. Note that the fastest implementation does, as was the case in the previous benchmark, make use of code generation: Indeed, FastParse2 relies on *macros* to generate a recursive-descent parser at compile-time.
Both the FastParse2 and the SPC parsers are susceptible to stack overflows, while the SILEX and SCALL1ON is not. In addition, FastParse2 and SPC do not guarantee linear-time parsing and do not provide residual parsers for resumption and inspection.

6.5.3 Pretty Printing JSON

As discussed earlier in the chapter, SCALL1ON also provides pretty printing capabilities. Using the slightly modified JSON parser shown in Appendix D, one can obtain a correct pretty printer almost for free from the description of the syntax. Table 6.8 shows the time taken by the SCALL1ON pretty printer to output the tokens corresponding to JSON values of various sizes.

File size (KB)	Tokens	Pretty printing time (ms)
100	9649	16.3
200	19297	29.9
300	28945	44.6
400	38593	56.0
500	48241	69.6
600	57889	82.5
700	67537	96.3
800	77185	110.1
900	86833	119.3
1000	97821	132.8
10000	971501	1561.6

Table 6.8 – Performance of the JSON pretty printer implemented using SCALL1ON.

6.5.4 Lexing and Parsing Python

As part of a Bachelor semester project under my supervision, Maillard (2020) built a complete lexer and parser for Python using SCALL1ON for the parser, and relying on his own library for the lexer. The student was able to build a fully functional Python parser, demonstrating the applicability of SCALL1ON to build large real-word parsers. The student checked that output produced by his parser indeed corresponds to that of the built-in CPython parser on a variety of large real-world projects, notably Django (Django Software Foundation, 2021), Zulip (Kandra Labs, Inc, 2021) and Flask (Ronacher, 2021). The performance of the lexer and parser, as benchmarked on those projects, is about a factor 15-to-20 behind that of the built-in CPython implementation, the SCALL1ON parser accounting for about a third of the running time, while two thirds of the time are spent in the custom lexer. Table 6.9 shows the performance of the student's lexer and parser for a selection of files of various sizes from the Django codebase. The performance is good enough for many practical uses.

File	Size (KB)	Tokens	Lex time (ms)	Parse time (ms)	CPython (ms)
`django/middleware/clickjacking.py`	2	125	3.0	9.4	0.2
`django/dispatch/dispatcher.py`	11	1122	17.0	22.2	1.4
`django/http/multipartparser.py`	26	3320	47.7	28.9	4.1
`django/forms/models.py`	58	8313	120.1	49.6	11.2
`django/test/testcases.py`	66	8911	129.4	77.8	9.2

Table 6.9 – Performance of the Python lexer and parser from Maillard (2020) compared to the builtin CPython parser on various Python source files from the Django codebase (Django Software Foundation, 2021). Note that the lexer used is based on a custom Scala library implemented by the student, not SILEX, which was not available at the time. The parser is based on SCALL1ON.

6.6 Applicability

SCALL1ON, as well as SILEX, have been used in practice in a variety of projects. Notably, SCALL1ON and SILEX are used as part of the *Compiler Construction* course offered to third year computer science bachelor students. As part of a semester long project, students build a compiler for a subset of Scala. Students user SILEX and SCALL1ON to build their lexer, respectively parser. Reception from students seems generally favourable.
SILEX and SCALL1ON have also been used as part of other EPFL courses. For instance, the two have been used to build a first-order-logic formula *quasiquoter* within Scala as part of the infrastructure for a project in the *Formal Verification* course.
A collection of examples built using SILEX and SCALL1ON are also provided in the code repository of SCALL1ON[1]. These examples include a JSON lexer and parser, a lambda calculus lexer, parser, and pretty printer, a lexer and parser for a small arithmetic operation language, and a parser for roman numerals.

[1]https://github.com/epfl-lara/scallion

7 Generalising Zippy Parsing with Derivatives

In Chapter 5, I have shown how using a zipper to represent LL(1) expression derivatives greatly speeds up the computation of successive derivations, leading to an efficient LL(1) parsing algorithm. The algorithm on which the zipper was applied was a variant of the *parsing with derivatives* algorithm specialised to LL(1) expressions. This begs the question if the zipper optimisation technique can be ported to the setting of the original *parsing with derivatives* algorithm and general value-aware context-free expressions. In this chapter, I present a generalisation of the LL(1) *parsing with derivatives and zippers* algorithm to arbitrary context-free expressions. A prototype implementation is available at https://github.com/redelmann/eclair.

7.1 Overview of Modifications

The algorithm presented in this chapter can be seen as a modified version of the parsing algorithm presented in Chapter 5 and implemented as part of SCALL1ON (see Chapter 6). The zipper datatype used in Chapter 5 for LL(1) expressions was relatively simple thanks to the LL(1) restriction. Without this restriction, the zipper data type used by the algorithm has to handle:

1. **Ambiguous Values.** In the general case, context-free expressions may associate an arbitrary, even infinite, number of values with any sequence of tokens. The representation of parse values used by the zipper must therefore accommodate this. In the LL(1) case, the syntax was bound to be unambiguous, and therefore the zipper could represent parse results as naked values. In the version of the zipper used in this chapter, collection of parse values are represented using a class called `Result`. Objects of type `Result` represent graphs of such parse values. Internal nodes of the graph may indicate cartesian products, unions, or function applications. Additionally, the graph may contain cycles and thus describe infinite collections of values.

2. **Concurrent Partial Interpretations.** In the case of LL(1) expressions, the derivation function followed a single logical thread of execution. In case of disjunction, the disjoint first sets of the two sides ensured that at most one side could be visited during derivation. The same applied for sequences, where the LL(1) property ensured that at most one of

the sides would be visited. This property transposed to the `Contexts` of the zipper: The upwards phase ended as soon as a subsequent syntax starting with the appropriate kind was found. In the general case, the upwards phase must continue exploring contexts as long as subsequent syntaxes are nullable.

In the general case, it is impossible to disambiguate looking only at the currently derived token. The zipper must be able to support multiple concurrent interpretations of the input. To do so, the `Context` data structure used by the generalised algorithm is no longer a simple stack but a *graph*. Each context node may have multiple parents, and cycles may be formed.

7.2 Data Structure Definition

As in Chapters 2 and 5, the data structure the parsing algorithm operates on is a *zipper*. The zipper combines aspects of both regular and LL(1) zippers. The zipper presented in this chapter is a collection of *values* in *contexts*. The zipper operates on an embedding of context-free expressions as `Syntax`es.

7.2.1 Syntaxes

The definition of the `Syntax` data type is unsurprising and resembles that used by SCALL1ON (see Chapter 6). One notable difference with the definition shown in Chapter 6 is that, in the interest of simplicity, the properties, such as nullability and first sets, are directly encoded as fields of the data type. Their computation however is still backed by a propagation network. The various case classes, as well as the interface of the `Syntax` trait, are presented in Listing 37. In the interest of simplicity, the body of the various case classes has been omitted. Such bodies mostly consist of the construction of the propagation networks used to compute properties.

7.2.2 Contexts

The definition of the `Context` class is given in Listing 38. The `Context` type is parameterised by two types, `A` and `Z`. The type parameter `A` represents the type of the *hole*, or *inner type*, while `Z` is the *outer type*. The various case classes of `Context` are as follows:

- The case class `Empty` represents an empty context. In this case, the type parameters `A` and `Z` of `Context` coincide.
- The case class `FollowBy` represents being in the left of a sequence. The `right` field contains a reference to the right `Syntax`.
- The case class `PrependBy` represents being in the right of a sequence. The `left` field contains a reference to a `Result` data type that represents a collection of parse values for the left syntax. Indeed, to visit the right of a syntax, the left must have already been processed down to a `Result`. I will discuss the `Result` type later in this chapter.

```
sealed trait Syntax[+A] {
  def isProductive: Boolean
  def isNullable: Boolean
  def hasFirst(kind: Kind): Boolean
}

case class Success[+A](value: A) extends Syntax[A]
case object Failure extends Syntax[Nothing]
case class Elem(kind: Kind) extends Syntax[Token]
case class Disjunction[+A](left: Syntax[A], right: Syntax[A]) extends Syntax[A]
case class Sequence[+A, +B](left: Syntax[A], right: Syntax[B]) extends Syntax[(A, B)]
case class Transform[A, +B](inner: Syntax[A], function: A => B) extends Syntax[B]

sealed abstract class Recursive[+A] extends Syntax[A] {
  def inner: Syntax[A]
}
object Recursive {
  def apply[A](deferred: => Syntax[A]): Recursive[A] =
    new Recursive[A] {
      override lazy val inner: Syntax[A] = deferred
    }
  def unapply[A](syntax: Syntax[A]): Option[Syntax[A]] = {
    if (syntax.isInstanceOf[Recursive[_]])
      Some(syntax.asInstanceOf[Recursive[A]].inner)
    else
      None
  }
}
```

Listing 37 – Definition of the generalised `Syntax` trait and associated case classes.

```
sealed trait Context[-A, +Z]
case class Empty[A]() extends Context[A, A]
case class FollowBy[A, B, Z](
  right: Syntax[B],
  parent: Context[(A, B), Z]) extends Context[A, Z]
case class PrependBy[A, B, Z](
  left: Result[A],
  parent: Context[(A, B), Z]) extends Context[B, Z]
case class Apply[A, B, Z](
  function: A => B,
  parent: Context[B, Z]) extends Context[A, Z]
case class Shared[A, Z](
  parents: Iterable[Context[A, Z]]) extends Context[A, Z]
```

Listing 38 – Definition of the generalised `Context` trait and associated case classes.

- The case class `Apply` represents being under the application of a host-language function. The field `function` contains a reference to the function to be applied on parse values.
- Finally, the `Shared` case class offers the possibility of multiple parent contexts. `Shared` nodes also allow cycles to be formed in the contexts.

Note that the definition of contexts is similar to that of LL(1) expressions seen in Chapter 5, with two notable differences:

1. The `left` field of `PrependBy` represents a collection of values instead of a single value,
2. The introduction of the `Shared` case class with the possibility of multiple parents. Indeed, the same syntax may be found in multiple contexts, either because the syntax appears initially multiple times as part of the top-level syntax, or because of duplication caused by derivations.

   The possibility of having multiple parents is essential to enable strategies that avoid duplication of work: During each derivation, each visited syntax node will be associated with a `Shared` context node. Upon revisit of the same syntax node during the same derivation call, the context can be added to the list of parents instead of revisiting the syntax.

7.2.3 Results

General context-free expressions can be ambiguous. The same expression can assign multiple values to the same sequence of tokens, potentially infinitely many. To represent collections of such values, I use a data type similar to `Syntax`. The class `Result` (shown in Listing 45), represent graphs of values. Cycles may be introduced, through mutation of the collection of values stored in the `results` field of `Disjunctions`.

```scala
sealed trait Result[+A]

object Result {
  case class Value[A](
    value: A) extends Result[A]
  case class Closed[A](
    syntax: Syntax[A]) extends Result[A]
  case class Sequence[A, B](
    left: Result[A],
    right: Result[B]) extends Result[(A, B)]
  case class Disjunction[A](
    results: Iterable[Result[A]]) extends Result[A]
  case class Transform[A, B](
    inner: Result[A],
    function: A => B) extends Result[B]
}
```

Listing 39 – Definition of the generalised `Result` trait and associated case classes.

The meaning of the various constructs of the `Result` class is as follows:

- The case class `Value` represents a single value.
- The case class `Closed` contains an unmodified *nullable* `Syntax`. The syntax is *closed*: Only the values it associates with the empty sequence of tokens are of interest.
- The case class `Sequence` represents cartesian production of results.
- The case class `Disjunction` represents union of results.
- Finally, the case class `Transform` represents the application of a function on top on the underlying results.

To avoid confusion with `Syntax` constructors, the various `Result` case classes are located within the `Result` companion object. I will refer to these constructors using their qualified names in the listings to follow.

7.2.4 The Zipper

The parsing algorithm showed in the next section makes use of the `Syntax`, `Context` and `Result` types I have just defined. In addition, the algorithm sometimes materialises values of some type paired with a matching context. To do so, the `Focused` trait is introduced. The parameter type constructor `F` is instantiated as either `Syntax` or `Result`, depending on whether a syntax or a result is to be paired with the context. Note that the type parameter of `Syntax` or `Result` must correspond to the inner type of the `Context`.

```
sealed trait Focused[F[_],+Z]
object Focused {
  case class Pair[F[_], A, Z](
    value: F[A],
    context: Context[A, Z]) extends Focused[F, Z]
}
```

Listing 40 – Definition of the `Focused` trait.

7.3 Derivation Algorithm

Now that I have introduced the various data structures used by the parsing algorithm, I can finally present the implementation of the derivation function. As was the case for the LL(1) *parsing with derivatives and zippers* algorithm, the derivation algorithm makes use of three main helper functions: `plug`, `locate`, and `pierce`.

7.3.1 Plug

The first helper function of interest is `plug`. The goal of `plug` is to propagate a result up a context. The definition of the function is more complicated as one could initially envision due to several factors:

1. Since contexts can end up being very large, the function can not be implemented recursively, as it would otherwise subject the derivation algorithm to stack overflows.
2. Additionally, `Shared` contexts may be visited multiple times during a single derivation. All invocations of `plug` done as part of a single derivation should avoid revisiting the same contexts multiple times.

To account for those constraints, the `plug` function is implemented via a helper `Plug` class (see Listing 41). In order to avoid using recursion, the function only peels off a single layer out of the argument context. The parameter functions `recordNext`, `recordStop`, and `recordResult` are called by the resulting `plug` function as a way of providing results to the caller. The function `recordNext` is called when more layers of the context must be peeled off before the next subsequent syntax, if any. The function `recordStop` is called when a subsequent syntax has been found. Finally, the function `recordResult` is called when the context is empty and the argument `Result` value is complete.
Each instance of the `Plug` class maintains internal state to avoid revisiting multiple times the same `Shared` contexts. In the case of `Shared` nodes, the helper function `updateOrDo` is used to query and update the cache. In case of a cache hit, instead of propagating the value up the context, the value is simply added to the mutable sequence associated with the node, thereby updating the associated `Disjunction` result. This crucially ensures that `Shared` contexts are only visited once, and that cycles in the context structures are gracefully handled.

7.3.2 Locate

The `locate` function (shown in Listing 42) is building on the `plug` function. The goal of `locate` is to return all subsequent syntaxes that start with a given kind and that are reachable within the context. Each such syntax is to be paired with the appropriate corresponding `Context`.
The `locate` function creates a new instance of the `plug` function, which it uses to unfold the context. Each time a subsequent syntax is found in the context, it is recorded, along with its context, in the result `points`, provided that it starts with the given kind. Additionally, if the syntax is nullable, the corresponding context is also further explored.

```
class Plug[Z](
    recordNext: Focused[Result, Z] => Unit,
    recordStop: Focused[Syntax, Z] => Unit,
    recordResult: Result[Z] => Unit) {

  private val entries: IdentityHashMap[Context[Nothing, Any],
                                       Result[Nothing] => Unit] =
    new IdentityHashMap()

  private def updateOrDo[A](result: Result[A], context: Context[A, Z])
      (action: Result[A] => Unit): Unit = {
    if (entries.containsKey(context)) {
      entries.get(context).asInstanceOf[Result[A] => Unit].apply(result)
    }
    else {
      val buffer = new ArrayBuffer[Result[A]]
      buffer += result
      entries.put(context, (extra: Result[A]) => buffer += extra)
      action(Result.Disjunction(buffer))
    }
  }

  def apply[A](result: Result[A], context: Context[A, Z]): Unit = {
    context match {
      case Empty() =>
        recordResult(result.asInstanceOf[Result[Z]])
      case Shared(parents) =>
        updateOrDo(result, context) { sharedResult =>
          for (parent <- parents) {
            recordNext(Focused.Pair(sharedResult, parent))
          }
        }
      case FollowBy(right, parent) =>
        recordStop(Focused.Pair(right, PrependBy(result, parent)))
      case PrependBy(left, parent) =>
        recordNext(Focused.Pair(Result.Sequence(left, result), parent))
      case Apply(function, parent) =>
        recordNext(Focused.Pair(Result.Transform(result, function), parent))
    }
  }
}
```

Listing 41 – Implementation of the `Plug` function factory.

```
def locate[A](
    focused: Focused[Result, A],
    kind: Kind): Iterable[Focused[Syntax, A]] = {

  val nextQueue: Queue[Focused[Result, A]] = new Queue
  val stopQueue: Queue[Focused[Syntax, A]] = new Queue

  val plug: Plug[A] = new Plug(nextQueue += _, stopQueue += _, _ => ())

  val points: ArrayBuffer[Focused[Syntax, A]] =
    new ArrayBuffer()

  nextQueue += focused

  while (nextQueue.nonEmpty || stopQueue.nonEmpty) {

    while (nextQueue.nonEmpty) {
      val Focused.Pair(result, context): Focused.Pair[Result, _, A] =
        nextQueue.dequeue()
      plug(result, context)
    }

    while (stopQueue.nonEmpty) {
      val Focused.Pair(syntax, context): Focused.Pair[Syntax, _, A] =
        stopQueue.dequeue()
      if (syntax.hasFirst(kind)) {
        points += Focused.Pair(syntax, context)
      }
      if (syntax.isNullable) {
        plug(Result.Closed(syntax), context)
      }
    }
  }

  points
}
```

Listing 42 – Implementation of the `locate` function.

Note that, contrarily to the LL(1) version of the function, the `locate` function may return multiple points.

7.3.3 Pierce

The final helper function used by derivation is `pierce` (see Figure 7.1). Like `plug`, `pierce` must maintain state across all calls performed as part of a single derivation. For this reason, the function is to be instantiated from the `Pierce` class.
The `pierce` function is called recursively over syntaxes and instantiates layers of context. A cache is maintained to avoid visiting the same syntax nodes multiple times. Upon revisit of a syntax, an extra parent context is added as a to the corresponding `Shared` context node instead of revisiting the syntax. When the revisit occurs as part of the visit to the syntax itself, a cycle in the resulting context is created. By construction, those cycles contain a `Shared` node, which ensures they are correctly handled by `plug` and other upwards functions.

7.3.4 Derive

The `derive` function is straightforward to implement given the `locate` and `pierce` helper functions I have previously shown. As an upwards phase, `locate` is called to gather all points which must be visited by `pierce` during the downwards phase. Intuitively, `locate` moves the focal points up the contexts during this upwards phase. During the downwards phase, `pierce` collects the contexts around matching `Elem` nodes found in left-most positions within the points found by `locate`.

```scala
def derive[A](focused: Focused[Result, A], token: Token): Focused[Result, A] = {
  val kind = getKind(token)
  val points = locate(focused, kind)
  val contexts = new ArrayBuffer[Context[Token, A]]()
  val pierce = new Pierce[A](contexts += _)
  for (point <- points) {
    val Focused.Pair(syntax, context): Focused.Pair[Syntax, _, A] = point
    pierce(syntax, kind, context)
  }
  Focused.Pair(Result.Value(token), Shared(contexts))
}
```

Listing 43 – Implementation of the `derive` function.

As a result of the derivation function, the derived token is put in the contexts that have been gathered by `pierce`.

```
class Pierce[Z](recordContext: Context[Token, Z] => Unit) {
  private val entries: IdentityHashMap[Syntax[Any], Context[Any, Z] => Unit] =
    new IdentityHashMap

  private def updateOrDo[A](expr: Syntax[A], context: Context[A, Z])
      (action: Context[A, Z] => Unit): Unit = {
    if (entries.containsKey(expr)) {
      entries.get(expr).asInstanceOf[Context[A, Z] => Unit].apply(context)
    }
    else {
      val buffer = new ArrayBuffer[Context[A, Z]]
      buffer += context
      entries.put(expr, (extra: Context[A, Z]) => buffer += extra)
      action(Shared(buffer))
    }
  }

  def apply[A](expr: Syntax[A], kind: Kind, context: Context[A, Z]): Unit = {
    updateOrDo(expr, context) { (sharedContext: Context[A, Z]) =>
      expr match {
        case Elem(_) => recordContext(sharedContext)
        case Disjunction(left, right) => {
          if (left.hasFirst(kind))
            apply(left, kind, sharedContext)
          if (right.hasFirst(kind))
            apply(right, kind, sharedContext)
        }
        case Sequence(left, right) => {
          if (left.hasFirst(kind))
            apply(left, kind, FollowBy(right, sharedContext))
          if (left.isNullable && right.hasFirst(kind))
            apply(right, kind, PrependBy(Result.Closed(left), sharedContext))
        }
        case Transform(inner, function) =>
          apply(inner, kind, Apply(function, sharedContext))
        case Recursive(inner) => {
          apply(inner, kind, sharedContext)
        }
        case _ => ()
      }
    }
  }
}
```

Figure 7.1 – Implementation of the `Pierce` function factory.

7.4 Parsing Algorithm

In this section, I present the generalised *parsing with derivatives and zippers* algorithm. The derive function I have just shown is the main component of the parsing algorithm. However, such a derivation function alone is not sufficient. The derivation function is used to evolve a zipper-based represent of a syntax as tokens of input are consumed. Before derivation can be first applied, the syntax must be converted into a zipper-based representation, which is done using the focus function. After all derivations are applied, the zipper-based syntax must be converted to a Result, using the result function.

7.4.1 Focus

The focus function is used to produce a Focused[Result, A] from an initial Syntax[A]. Since the zipper requires an explicit focal point, an artificial Result(Epsilon(())) is created and used as the focal point. The context consists of two layers:

1. The syntax itself, as part of a FollowBy.
2. A final Apply layer, whose goal is to discard the artificial value introduced by the initial focal point.

The function is only called once per invocation of the parsing algorithm, at the start.

```
def focus[A](syntax: Syntax[A]): Focused[Result, A] =
  Focused.Pair(
    Result(Epsilon(())),
    FollowBy(syntax, List(Apply(_._2, List(Empty())))))
```

Listing 44 – Implementation of the focus function.

7.4.2 Result

Once all tokens have been derived, a final upwards phase must take place before a Result can be obtained. The result function propagates upwards the context the collection of values in focus in the argument Focused[Result, A].
The structure of the function is reminiscent of the locate I have previously shown. Instead of recording syntaxes that start with a given kind on the way, the result function simply tries to reach the top of the context, skipping nullable subsequent syntaxes.

7.4.3 Parse

Given the presented helper functions, the definition of the parsing algorithm is extremely simple: The argument syntax is converted into a zipper-based representation using focus. Then, derivation is iteratively applied onto that zipper for each input token. Finally, result is called on the resulting zipper to compute the final collection of values.

```
def result[A](focused: Focused[Result, A]): Option[Result[A]] = {
  val nextQueue: Queue[Focused[Result, A]] = new Queue
  val stopQueue: Queue[Focused[Syntax, A]] = new Queue
  var result: Option[Result[A]] = None

  val plug: Plug[A] = new Plug(nextQueue += _, stopQueue += _, r => result = Some(r))

  nextQueue += focused

  while (nextQueue.nonEmpty || stopQueue.nonEmpty) {

    while (nextQueue.nonEmpty) {
      val Focused.Pair(result, context): Focused.Pair[Result, _, A] =
        nextQueue.dequeue()
      plug(result, context)
    }

    while (stopQueue.nonEmpty) {
      val Focused.Pair(syntax, context): Focused.Pair[Syntax, _, A] =
        stopQueue.dequeue()
      if (syntax.isNullable) {
        plug(Result(syntax), context)
      }
    }
  }

  result
}
```

Listing 45 – Implementation of the `result` function.

```
def parse[A](syntax: Syntax[A], tokens: Iterable[Token]): Option[Result[A]] = {
  var current = focus(syntax)
  for (token <- tokens) {
    current = derive(current, token)
  }
  result(current)
}
```

Listing 46 – Implementation of the `parse` function.

7.5 Producing Values

The `Result[A]` datatype represents collections of values of type `A`. The data structure underlying `Result` forms a graph which may contain cycles, making value enumeration non-trivial. As usual, a bottom-up algorithm may be used to lazily iterate over values. In case a single value is of interest to the user, a simpler method may be invoked on `Result`. As I will discuss later in this chapter, the `Result` datatype could also be substituted with naked values in this case to save space.

7.6 On Immutability

The parsing algorithm presented in this chapter makes heavy use of mutation. Yet, I argue that the algorithm is still functional. As I will show, the zipper data structure used to represent derivatives is still immutable and fully persistent. The reason is that mutations performed during derivations only occur on newly created structures, never on already existing ones.

The generalised algorithm makes use of mutation in several places. Notably, both `plug` and `pierce` maintain mutable state for memoisation purposes. This state is invisible to users; it is entirely discarded after each derivation.

Next, the contents of the `results` fields of `Disjunction` nodes of `Results` can be mutated by the derivation procedure. These mutations are also invisible to the user, as only fresh `Disjunction` nodes may have their `results` field altered.

Finally, the `parents` fields of contexts can be mutated during derivation. Yet, as was the case for `Disjunction` nodes of `Results`, only freshly created contexts may have their `parents` field altered.

In the shown implementation, this is further guaranteed by a typing discipline: Mutable collections are typed with the rather general and immutable interface `Iterable` within nodes. The derivation procedure, on the other hand, maintains collections it creates under more concrete types that offer a mutable interface, such as `ArrayBuffer`.

Thus, the derivation procedure does not mutate anything it receives as argument, making the zipper data structure immutable and persistent for all intent and purposes.

7.7 Correspondence with Other Parsing Techniques

The parsing algorithm presented in this section corresponds in many interesting aspects with other parsing algorithms. A discussion of the most notable links are discussed in this section.

7.7.1 Earley Parsing

Interestingly, the parsing algorithm shown in the chapter is reminiscent of Earley's parser (Earley, 1970). In this section, I give a brief overview of the Earley's recognition algorithm for context-free grammars. The main objects used by Earley's algorithm are Earley's items. Each item is of the form $(X \to \alpha \bullet \beta, i)$, where X is a non-terminal symbol of the grammar, α and β are sequences of symbols such that $X \to \alpha\beta$ is a rule of the grammar, and i is an integer

position. The symbol $\bullet$ indicates a point within rules, some sort of *focus*: To the left of the $\bullet$ are symbols that have already been recognised, while to the right are symbols that remain to be matched. The position i denotes the index in the input at which the item starts. Given n tokens of inputs, Earley's algorithm iteratively builds a collection of $n+1$ item sets denoted $S(0)$ to $S(n)$.
The algorithm maintains the invariant that a set $S(k)$ contains the item $(X \to \alpha \bullet \beta, i)$ only if α matches against the tokens $t_{i+1} \dots t_k$ and either their exists an item $(X' \to \alpha' \bullet X\beta', i')$ in $S(i)$ or $k = 0$ and X is the initial symbol of the grammar. Intuitively, the presence of an item $(X \to \alpha \bullet \beta, i)$ in a set $S(k)$ indicates a partial match of the symbol X starting at position i and currently positioned at k. α represents the already recognised symbols and β the remaining symbols to process.
Earley's algorithm proceeds position by position, saturating the sets $S(\cdot)$ iteratively as tokens of input are processed. In the end, the algorithm reports a successful parse if and only if an item $(S \to \alpha\bullet, 0)$, where S is the start symbol, exists in the set $S(n)$.
For each $k = 0 \dots n$, an iterative phase saturates the set $S(k)$. During this saturation phase, three types of operations are performed:

1. **Completions.** Items of the form $(X \to \gamma\bullet, i)$ in $S(k)$ can be *completed*. For each corresponding item $(Y \to \alpha \bullet X\beta, j)$ in $S(i)$, the item $(Y \to \alpha X \bullet \beta, j)$ is added to $S(k)$.

2. **Predictions.** For each item $(Y \to \alpha \bullet X\beta, i)$ in $S(k)$, items $(X \to \bullet\gamma, k)$ are added in $S(k)$ for every production rule $X \to \gamma$ of the grammar.

3. **Scannings.** For each item $(X \to \alpha \bullet t_{k+1}\beta, i)$, the item $(X \to \alpha t_{k+1} \bullet \beta, i)$ is added to the set $S(k+1)$.

Note that several actions may be performed on the same item: An item may offer both *completion* and *prediction* opportunities. Due to nullable symbols, completions opportunities can be discovered after predictions within a single saturation phase. Finally, remark that *scanning*, as it adds elements to the *subsequent* set $S(k+1)$, can not uncover more action opportunities in the saturation phase for $S(k)$.

Correspondence with Zipper-Based Algorithm

Each operation performed by the zipper-based algorithm shown in this chapter corresponds to one (or several) operations performed by Earley's algorithm:

1. The `locate` upwards phase corresponds to *completions*. Instead of looking up corresponding items in a different set, as is the case in Earley's algorithm, contexts contains a direct reference to all corresponding contexts: its *parents*.

   As a simple optimisation, the zipper-based algorithm of this chapter uses first sets of syntaxes to provide a lookahead of a single token. The way nullable expressions are handled is also slightly different: Whereas Earley's require a series of *predictions* and *completions* to move past nullable symbols, the zipper-based algorithm uses the nullable property of syntaxes to directly skip over them during the upwards phase.

2. Calls to `pierce` of the downwards phase correspond to *predictions*. As was the case for the upwards phase, first sets are used to provide lookahead. Nullable symbols are also handled slightly differently: Whereas Earley requires a series of *predictions* and *completions* to move past nullable symbols on the left and explore reachable symbols in the right of sequences, the zipper-based algorithm directly explores the right expression of a sequence if the left expression is nullable (and the right expressions starts with the appropriate kind).

3. Finally, the replacement of the focused `Elem(kind)` nodes by a `Result.Value(token)` node correspond to *scanning*.

One obvious difference between Earley's algorithm, as presented in this section, and the zipper-based parsing algorithm of this chapter is that value-elaboration aspect. As presented here, Earley's algorithm is a mere recogniser: To the left of • are symbols of the grammar, whereas in the case of the zipper-based algorithms, collections of values represented using the `Result` datatype are present in contexts.
The close correspondence of the zipper-based operations with Earley's operations strongly signals that the two algorithms share the same worst-case complexity. Each action performed by the zipper-based algorithm can be decomposed into a series of constant-time steps that correspond to actions performed by Earley's algorithm. This offers a simple argument that generalised *parsing with derivatives and zippers* is worst-case cubic in the general case, worst-case quadratic for *unambiguous grammars*, and worst-case linear for *deterministic grammars* Earley (1970).
A similar observation was made by Henriksen et al. (2019). These authors remark that their parsing algorithm based *derivative grammars*, a transposition of Brzozowski's derivatives to context-free grammars, also closely corresponds to Earley's algorithm.

7.7.2 GLR Parsing

The zipper-based algorithm presented in this chapter is also reminiscent of the *Generalised LR parsing* algorithm, also known as *GLR* or *Tomita's algorithm* (Tomita, 1987, 2012). The main difference being that GLR precomputes an LR parsing table to guide the algorithm, while the zipper-based algorithm presented in this chapter does not. Memoising calls to `pierce`, as done in the context of the LL(1) parsing algorithm presented in Chapter 5, could offer a way to lazily build a distributed version of the parse table. This is not done in the current version of the algorithm due to the global nature of contexts: contexts created by `pierce` may depend on contexts created by other calls to `pierce` performed in the same derivation invocation.
The *Graph-Structured Stack* (GSS) used by the GLR algorithm corresponds to the `contexts` of the zipper-based algorithm. As originally described, GSS are acyclic graphs, which prohibits certain types of recursion in grammars. The zipper-based algorithm of this chapter presents no such restrictions, and explicitly incorporate semantic actions.

7.7.3 GLL Parsing

Another parsing algorithm that uses graph structured stacks is the *Generalised LL* (GLL) parsing algorithm of Scott and Johnstone (2010). GLL is a *parser generation* algorithm, which generates parsers that operate on a modified GSS to support cycles. The GSS used by GLL parsers closely correspond to the contexts of the zipper-based algorithm of this chapter, which also support cycles. Being a parser generation technique, GLL is somewhat dissimilar to the dynamic parsing algorithms presented in this thesis.

7.8 Performance

In this section, I examine the performance of the parsing algorithm shown in this chapter in various settings.

7.8.1 On JSON Parsing

The performance of the algorithm in practice is, as expected, slower than that of the LL(1) version of the algorithm. On the JSON syntax benchmark, an implementation of the generalised algorithm processes tokens at a speed of around 325,000 tokens per second, which is about 20 times slower than the LL(1) parsing algorithm used by SCALL1ON. Interestingly, on an implementation of the JSON syntax written using left recursion, the performance of the algorithm slightly improves, reaching a speed of approximately 450,000 tokens per second. Figure 7.2 shows the time taken by the generalised version of the algorithm to parse JSON files of sizes ranging from 100KB to 1MB using either a right-recursive syntax or a left-recursive syntax. The performance of SCALL1ON is also reported for comparison.

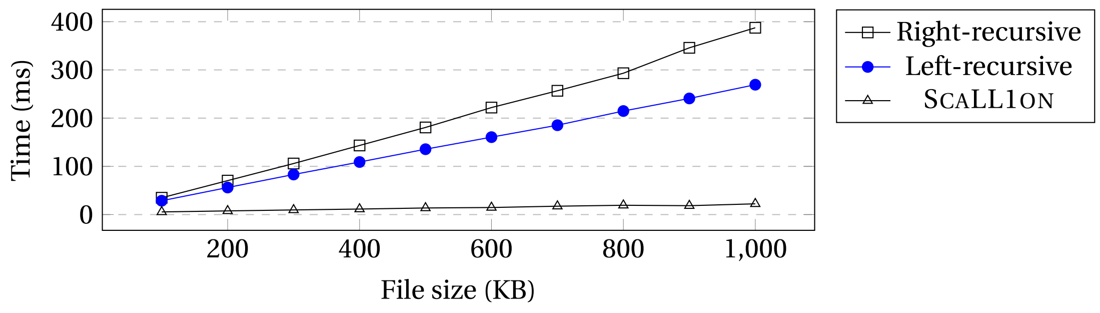


Figure 7.2 – Time taken to parse JSON files of various sizes.

This gap in performance compared to the LL(1) version is explained by the costly infrastructure in place to handle ambiguities. In the case of SCALL1ON, the LL(1) property allows for sweeping simplifications: The costly bookkeeping done by the `plug` and `pierce` functions of this chapter, which are necessary to avoid duplicating work, are entirely avoided in the LL(1) case. Furthermore, due to the simpler structure of contexts in the LL(1) case, the additional layers of context added by `pierce` can be memoised across derivations, which is not possible in the generalised case due to the non-local nature of the context graph.

7.8.2 On Highly Ambiguous Grammars

Contrarily to the LL(1) version of the algorithm, the generalised algorithm can be applied to any context-free expression, even ambiguous ones. This benchmark measures the performance of the algorithm on a simple but highly ambiguous example. The language of the example `atrees` syntax of Listing 47 ranges over all repetitions of the letter a. The values assigned to each sequence of as are *all* binary trees that cover the sequence of letters. Scala's tuples are used for branch nodes, while the `'a'` characters themselves form leaves.

```
lazy val atrees: Syntax[Any] = Recursive {
  Disjunction(Elem('a'), Sequence(atrees, atrees))
}
```

Listing 47 – Definition of the `atrees` syntax.

The number of such binary trees follows the sequence of *Catalan numbers*, which is asymptotically exponential. Yet, as demonstrated in Figure 7.3, the performance of the generalised parsing algorithm on increasingly longer sequences of as follows a cubic progression, not exponential. In the figure, the black squares indicate measurements. The blue line shows a fitted cubic curve ($y = 0.00004393835x^3$), and the red line shows a fitted exponential curve ($y = 1.020755^x$).

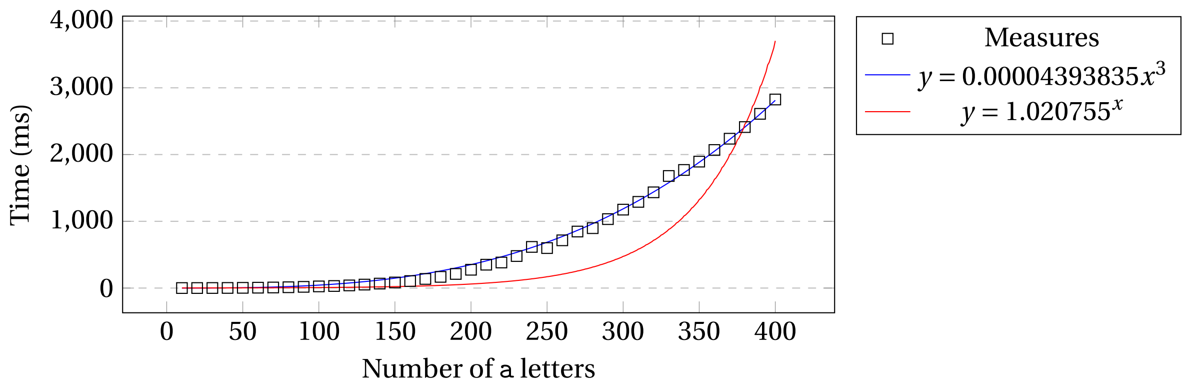


Figure 7.3 – Time required to parse sequences of a's of various sizes.

7.9 Variants

The parsing algorithm presented in this section can easily be adapted to be best suited for specific use cases. I present an overview of some of the major possible changes.

7.9.1 Results

For non-ambiguous expressions, or when only a single resulting value is needed, it makes sense to replace the `Result[A]` type by naked values of type `A`. Using naked values has the potential to save memory and may result in faster value computation. Upon revisit of a context

layer during the upwards phase, the additional values can simply be discarded instead of being recorded in as part of a `Disjunction`.
Similarly, the parsing algorithm can be turned into a recogniser by switching `Result[A]` with `Unit`. In this case, the value propagated during upwards phases is simply `()`.

7.9.2 Context Lookahead Caching

Since the same contexts can be visited by `locate` during multiple derivations, it may be worthwhile to record the kinds for which the visit led to successful location of subsequent syntaxes starting with the given kind, and for which kinds it did not. Upon visit of a context layer by `locate`, if the kind is known not to follow, then the visit of the context can be immediately halted.
Recording those kinds in contexts comes at a cost: During `locate`, those sets of kinds must be updated for layers traversed. This operation may still be worthwhile on certain classes of expressions.

7.9.3 Memoisation Policies

As presented in this chapter, the derivation algorithm makes heavy use of memoisation: At the level of contexts and at the level of syntaxes. In the LL(1) algorithm presented in Chapter 5, such memoisation efforts are wasted, as both contexts and syntaxes are bound not to be visited multiple times during derivation. The algorithm presented in the present chapter can be easily adapted to different memoisation strategies best suited for certain classes of expressions.

7.10 Discussion

The parsing algorithm presented in this chapter is an adaption of the *LL(1) parsing with derivatives and zippers* algorithm presented in Chapter 5. The algorithm is able to handle arbitrary context-free expressions, even those that feature left-recursion. This flexibility however comes at a cost in performance and predictability: The algorithm is several times slower than the LL(1) version and its worst-case complexity is cubic for the general class of context-free expressions.
I would argue against the use of general context-free parsing algorithms in settings such as programming languages, where files can grow relatively large and the expectation is worst-case linear complexity. Without help, designing a syntax that is worst-case linear is error prone. Instead, I would argue for solutions like SCALL1ON in which the syntax can be checked ahead of parsing to ensure linear-time behaviour. The algorithm presented in this section provides a solid foundation for designing linear-time parsing algorithms restricted to certain classes of expressions. The LL(1) parsing algorithm presented in Chapter 5 is one such example. One could envision using the generalised algorithm as the basis of an (LA)LR(1) parsing algorithm, for instance. A parser combinators interface, such as the one of SCALL1ON, substantially improves the expressiveness of parsing techniques in practice; *parsing with derivatives and zippers* algorithms very naturally support such interfaces.

8 Related Work

In this chapter, I give references to related works, broadly classified by topic. Even though some works span multiple categories, they are only discussed in the context of the category I consider the most relevant to them.

8.1 Parser Combinators

The parsing techniques presented in this thesis support a high-level parser combinators interface. The idea of using mutually recursive functions to describe parsers, which gave rise to parser combinators, dates back at least to Burge (1975). Parser combinators have then become the subject of many interesting works, notably (Hutton, 1992; Fokker, 1995; Hutton and Meijer, 1996). These authors present techniques for building recursive-descent parsers based on a collection of combinators, which generally includes a monadic *bind* operator. Parsing libraries based on this approach have been implemented in many functional programming languages, notably Haskell (Leijen and Meijer, 2001) and Scala (Moors et al., 2008).
The combinators of SCALL1ON, the parsing library presented in this thesis, offer a more restrictive interface based on *applicative functors* (McBride and Paterson, 2008) instead of *monads* (Wadler, 1995). Additionally, whereas typical parser combinator libraries use a shallow embedding of combinators as host-language functions, SCALL1ON makes use of a deep embedding of combinators. This deep embedding enables the parsing algorithm of SCALL1ON, as well as many of its additional features.

8.2 Parsing Algorithms and Techniques

Parsing algorithms and techniques are many and varied. In this section, I present several related works on parsing, roughly grouped together based on the classes of languages they handle.

8.2.1 Packrat Parsing

Ford (2002) 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. Such parsers however tend to be slower than many other linear-

time parsing techniques (Becket and Somogyi, 2008; Grimm, 2004).
Whereas PEGs disallow ambiguities through biased choices, LL(1) approaches such as the one presented in Chapters 4 to 6 support detecting ambiguities before parsing starts. The combinators used in this thesis also enjoy more natural algebraic properties than those of PEGs: Disjunctions I used are commutative and associative, which is not the case in PEGs, making the composition of PEGs trickier.

8.2.2 LL(1) Parsing

The parsing technique shown in Chapters 4 and 5 and part of the implementation of Chapter 6 is part of the family of LL(1) parsers. The traditional automaton-based LL(1) parsing algorithms dates back at least to (Lewis and Stearns, 1968).
Swierstra and Duponcheel (1996) 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 derivatives.
Krishnaswami and Yallop (2019) 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 fix-point computations, as opposed to propagation networks. They use a weaker definition of *should-not-follow* set (which they call *follow-last* set, abbreviated as FLAST). As a result, their type system is more restrictive than the one presented in this thesis: it does not allow nullable expressions to appear on the left of sequences.

8.2.3 LL(*) Parsing

Parr and Fisher (2011) presents a parsing algorithm called *LL(*)*, which operates in top-down fashion and uses a mechanism based on regular expressions to perform arbitrary lookahead in certain cases, falling back to backtracking when impossible. The approach is backed by a static analysis technique that tentatively builds deterministic finite-state automata to disambiguate rules of non-terminals. The LL(*) algorithm is used by the ANTLR parser generator Parr (2013).
As future work, techniques employed by the LL(*) algorithm could be adapted to the context of the generalised *parsing with derivatives and zippers* presented in Chapter 7. The automata building technique shown in Chapter 2 could be used in this context.

8.2.4 LR Parsing

Knuth (1965) presents a parsing algorithm for processing certain classes of context-free grammars, called LR(k) grammars, in linear-time. Many variants of the algorithm have been proposed since, notably (DeRemer, 1969, 1971; Čulik II and Cohen, 1973; Pager, 1977).
LR parsing algorithms are used in the context of the parsers generators *Yacc* (Johnson et al., 1975) and *Bison* (Donnely and Stallman, 2015).

8.2.5 Generalised Parsing Algorithms

The parsing algorithm presented in Chapter 7 is an instance of a generalised context-free parsing algorithm. Such algorithms can recognise arbitrary context-free languages, and generally run in worst-case cubic time in the number of input tokens. I discuss several connections between such algorithms and the zipper-based technique of this thesis in Section 7.7.

The Cocke–Younger–Kasami algorithm (CYK) (Cocke, 1969; Younger, 1967; Kasami, 1966; Sakai, 1962) is a chart-based, bottom-up parsing algorithm operating on context-free grammars in *Chomsky normal form* (Chomsky, 1959). Contrarily to many generalised parsing algorithms, CYK is purely bottom-up, which leads to more tentative interpretations of segments of the input compared to approaches that incorporate top-down aspects.

Earley (1970) proposes a generalised parsing algorithm based on the concept of *items*. The algorithm incorporates both top-down and bottom-up rules. As discussed in Chapter 7, Earley's item are similar to the context layers of the generalised zipper-based parsing algorithm presented in this thesis.

Tomita (1987, 2012) present a *Generalised LR parsing* algorithm (GLR). The GLR algorithm operates on an LR parsing table and explicitly represents concurrent tentative interpretations of the input. GLR uses a *Graph-Structured Stack* (GSS) (Tomita, 1988) to represent its stack, which is reminiscent of the *contexts* used by the algorithm shown in Chapter 7. However, as originally described, GSS are acyclic, which prohibits certain types of recursion in grammars. GLR has been implemented as part of *Elkhound* (McPeak and Necula, 2004). *Bison* (Donnely and Stallman, 2015) also supports GLR as an alternative parsing algorithm.

Scott and Johnstone (2010) present a parsing technique called *Generalised LL* parsing (GLL). GLL is framed as a *parser-generation* technique: The technique emits code to produce generalised parsers from a grammar description of the language. GLL parsers make use of a modified version of GSS that allows cycles to represent their stacks.

In a different vein, Herman (2020) presents a general parsing algorithm which simulates runs of non-deterministic pushdown automata using operations of regular languages, notably Brzozowski's derivation. Thanks to the immutable nature of the data-structures used by the algorithm, the technique can employ a memoisation technique that the author calls *context-free memoisation*.

8.2.6 Parsing with Derivatives

Might et al. (2011) present a parsing algorithm for general context-free expressions based on Brzozowski's derivatives. The algorithm is purely top-down. The worst-case complexity of their approach is cubic in general (Adams et al., 2016), and, as shown in Chapter 4, is quadratic for LL(1) expressions.

Brachthäuser et al. (2016) 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 the setting of SCALL1ON. However, most of the examples and patterns demonstrating the power of their approach require a monadic `flatMap` combinator, which is tricky to implement

within the setting of SCALL1ON without compromising LL(1) checks.
Henriksen et al. (2019) show a parsing technique based on derivatives for context-free grammars. They show that their approach is equivalent to Earley's algorithm (Earley, 1970) and argue that *parsing with derivatives* has deep connections with traditional parsing techniques.
In a work concurrent with the developments shown in this thesis, Darragh and Adams (2020) present a general parsing algorithm for context-free expressions using derivatives and zippers. Their approach is most similar to the one presented in Chapter 7. The algorithm presented in Chapter 7 of this thesis however differs in several significant aspects:

1. The technique presented by Darragh and Adams (2020) does not incorporate combinators for manipulating parse values. As such, their algorithm returns a graph representation of the parsed input, whereas the technique shown in this thesis returns a graph that explicitly contains function-application nodes, enabling the elaboration of user-defined values.

2. The algorithm presented in this thesis makes use of *first set* and *nullability checks* to guide derivation during downwards phase; the algorithm from Darragh and Adams (2020) does not.

3. The datatypes used by Darragh and Adams (2020) are mutated by the derivation function in a way that is visible to users of the algorithm, thereby breaking immutability and opportunities for persistence. In contrast, the work presented in this thesis uses mutation more cautiously, in a manner that preserves the immutable interface.

8.2.7 Memoisation-based Parsing Techniques

Norvig (1991) presents a technique for general context-free parsing based on a combination of top-down parsing, backtracking, and memoisation. The author shows that technique behaves similarly to Earley's parsing algorithm. However, the technique does not support left-recursion.
Johnson (1995) shows how to adapt the technique presented by Norvig (1991) to support left-recursion, and so by adopting a *continuation-passing style* representation.
In the same line of work, Izmaylova et al. (2016) present a parser combinators interface for memoised continuation-passing-style parsing. The implementation guarantees worst-case cubic time complexity and employs a packed representation to represent collections of parse values. The algorithm presented in Chapter 7 presents the same features but employs a deeper embedding of parsers and continuations (contexts).

8.2.8 Support for Conjunction and Negation

Some works have looked a supporting conjunction and negation as additional combinators. In the context-free setting, Okhotin (2001, 2013) presents adapted parsing techniques that support such combinators.
In the regular setting, Brzozowski's derivation Brzozowski (1964) is also defined on such extended expressions. Interestingly, Antimirov's technique (Antimirov, 1996), which is rem-

iniscent of the zipper-based technique of Chapter 2, can also be easily adapted to support extended expressions, as shown by Caron et al. (2011). This strongly signals that the zipper-based techniques presented in this thesis can also be adapted to support extended expressions. This is left as future work.

8.3 Enumeration

Koukoutos (2019) presents an enumeration algorithm over programs described by context-free grammars. This algorithm is reminiscent of the enumeration algorithm used as part of SCALL1ON and briefly discussed in Chapter 6.
Kuraj et al. (2015) present a combinator-based framework for enumeration of structures. The combinators used in this work differ however significantly from those used in this thesis. The work by Kuraj et al. (2015) could provide insights on implementing additional combinators for the purpose of enumeration in the context of SCALL1ON (see Chapter 6).
Madhavan et al. (2015) use enumeration as a way to find sequences that are accepted by one context-free language but rejected by another, thus providing a semi-procedure to check for context-free language non-equivalence.
Godefroid et al. (2008) present a whitebox fuzzing technique based on generating input from a grammar description. As future work, such a technique could be adapted to the setting of this thesis and provide a way of testing parsers, as well as potentially later phases, of formal language tools.

8.4 Correct-by-Construction Pretty Printing

Boulton (1996) presents a technique for building a lexer, parser, pretty printer, as well as abstract syntax trees, from a single language description based on EBNF notation Backus et al. (1960). Later, Ranta (2004, 2011) present *Grammatical Framework*, a domain-specific language for defining languages. Abstract syntax trees, parsers and pretty printers are obtained from the same language description. In contrast, the interface of the parsing and pretty-printing approach presented in this thesis is based on (embedded) combinators, and is not concerned with lexical aspects nor definition of abstract syntax trees.
Rendel and Ostermann (2010) present a technique for mutually inverse parsing and pretty printing based on *syntactic descriptions* and isomorphisms. Their combinator-based interface is similar to that of this thesis.
Matsuda and Wang (2013) present a technique for building parsers from pretty printer descriptions. Whereas the technique presented in this paper is biased towards the description of parsers, the technique they present is biased towards description of pretty printers. The pretty printers described by Matsuda and Wang (2013) are concerned with layouts and other lexical aspects, whereas the technique presented in this thesis operates strictly at the syntactic level; the technique produces sequences of tokens that requires further processing to turn into actual text. Combinators and techniques from Matsuda and Wang (2013) could be investigated to provide more control over lexical aspects.

Delaware et al. (2019) show a context-sensitive combinator library for building parsers and corresponding pretty printers for binary formats. The approach is implemented and verified in Coq. In contrast, the work of this thesis focuses on context-free text-based formats. The approach by Delaware et al. (2019) might provide insights on how to verify the correctness of the pretty-printing procedure were such efforts to be undertaken in the future.

8.5 Formally Verified Parsing

Recently, verification of parsers has been the subject of many interesting works. Formally verified parsers are of special interest to verified compilers such as CompCert (Leroy, 2009) and CakeML (Kumar et al., 2014).
Ramananandro et al. (2019) demonstrate the importance of parsers in security and present combinators for building verified high-performance parser for *lower-level* encodings of data formats. Koprowski and Binsztok (2010) present a formally verified Coq parser interpreter for Parsing Expression Grammars (PEGs). In a recent work, Lasser et al. (2019) 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.
As an alternative approach, Jourdan et al. (2012) 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.

8.6 Derivatives and Formal Reasoning

Brzozowski's derivatives (Brzozowski, 1964) have been used with great success in formal proofs about regular languages and some of their extensions. Pierce et al. (2018) propose proving the correctness of a regular expression matching procedure using Brzozowski's derivatives in a series of exercises in the context of a tutorial on Coq. Also, Ausaf et al. (2016) present a correctness proof of a POSIX regular expression matching algorithm in Isabelle based on derivatives. Finally, Traytel and Nipkow (2013) present a verified decision procedure for *monadic second-order logic of finite words* (MSO) based on an adaptation of Brzozowski's derivation operation to MSO formulas.
Antimirov (1996) introduces the concept of *partial derivatives*, which are sets of expressions that represent Brzozowski's derivatives. Several interesting properties of partial derivatives are shown, including a finiteness property. The zipper-based representation of derivatives I employ in Chapter 2 is reminiscent of such a concept. Antimirov's *partial derivatives* have also been successfully used in verification efforts. Notably, Moreira et al. (2012) use partial derivatives as part of a verified decision procedure for regular language equivalence.
The proof of finiteness of zipper-based derivatives shown in Chapter 2 could be built upon to provide an elegant and concise proof of the Myhill-Nerode theorem (Hopcroft et al., 2001b). An existing formal proof of the theorem in Isabelle/HOL by Wu et al. (2011) operates at the level of regular expressions but explicitly doesn't make use of derivatives. These authors

mention not using derivatives due to the difficulty in practice of deciding their equivalence. The zipper-based technique shown in Chapter 2 of this thesis alleviates this issue.

8.7 Fix-Point Computations

In this thesis, I suggest using *propagation networks* (Radul, 2009) to compute properties of context-free expressions. Interesting alternatives exist: notably, Pottier (2009) present a technique for computing least fix-points based on information dependency graphs, and propose an implementation as a library in OCaml called *fix* (Pottier, 2013). The dependency graph of (Pottier, 2009) is similar to the network of cells of (Radul, 2009), but offers an elegant high-level programming interface. Interestingly, (Pottier, 2009) showcases how to compute nullability of grammar symbols as an example application of the technique. Unfortunately, I am unaware of any implementation of the technique in Scala, and so had to resort to an *ad hoc* implementation of propagation networks in the various projects discussed in this thesis. Porting the *fix* library to Scala would be interesting and useful future work.

8.8 Datatype Derivatives

Brzozowski (1964) shows a technique for building deterministic finite-state automata based on regular expression *derivatives*. The derivation operation proposed by Brzozowski has a structure reminiscent of that of the derivation operation of calculus.
McBride (2001) highlights a correspondence between Huet's *zipper* (Huet, 1997) and *derivatives* of regular datatypes. McBride (2001) shows how to compute one-hole contexts of a datatype using an operation called *derivation*. Like Brzozowski's derivation, the operation has a structure reminiscent of derivation as in the context of calculus, hence its name. In a later work, McBride (2008) adapted the technique to support different types on the left and right sides of contexts. The zipper-based parsing techniques of this thesis also use different types on the left and right of focal points.
In this thesis, I demonstrate that marrying Brzozowski's derivatives and McBride's derivatives leads to efficient lexing and parsing techniques. In the light of their similar nature, their fruitful union is in retrospect unsurprising. Investigating this link in a deeper manner may be interesting future work.

9 Conclusion

In this thesis, I demonstrate that *parsing with derivatives and zippers* is a technique of choice for building fast and flexible lexical and syntactic analysis tools that can be trusted. Derivative-based approaches promise to combine the declarative interface found in parser generators with the lightweight, flexible, and value-aware nature of dynamic parsers. Unfortunately, the performance of derivative-based parsing approaches was, until this work, not practical. Through the use of zipper-inspired data structures, I showed how one can optimise derivative-based algorithms to the point of competitiveness with state-of-the-art techniques in terms of performance. This thesis touched upon many aspects of parsing, solving some of the challenges and identifying new ones.

Amenability to Formal Reasoning

This thesis demonstrates that context-free expressions are a valid alternative to context-free grammars. Context-free expressions are value-aware, easy to embed in functional programming languages and proof assistants alike, and are very natural to transition into from regular expressions. In Chapter 3, I have shown how to effectively define and compute properties of such expressions. In Chapter 4, I also offered a characterisation of the LL(1) class of expressions using the concept of *should-not-follow* sets, an alternative to the `FOLLOW` sets of traditional grammars.
This thesis demonstrates that derivatives and zippers-based approaches are amenable to formal reasoning. The derivatives and zippers-based technique showed in Chapter 2 for lazily building deterministic finite-state automata, as well as the LL(1) *parsing with derivatives and zippers* from Chapter 5, have both been successfully formalised in Coq.

Ease of Integration

This thesis demonstrates that derivatives and zippers-based lexical and syntactic analysis techniques are easy to integrate into software projects. In Chapter 2, I have shown how to implement an efficient lexical analysis library in a very elegant and concise way, and have demonstrated the applicability of the technique by providing both a Scala and a minimal Haskell implementation. In Chapter 6, I have shown an implementation of the LL(1) *parsing with derivatives and zippers* algorithm as part of the SCALL1ON library. I have shown that context-free expressions are natural to embed in functional programming environments. Thanks to the value-awareness of context-free expressions, value elaboration does not require a separate and burdensome conversion phase.

Support for Expressive Interfaces

In this thesis, I have shown that derivatives and zippers-based approaches can offer an expressive interface. In Chapter 2, I showed how to build an expressive rule-based interface for defining lexers. In Chapter 6, I have presented SCALL1ON, a parsing combinator library for LL(1) languages. SCALL1ON offers a collection of expressive combinators, and has features that are usually not available in traditional parsing combinators library, like property checks, enumeration, and pretty printing. I have demonstrated that derivative-based algorithms also support features that are harder to implement otherwise, such as precise error reporting and code completion. The addition of richer combinators, such as, for instance, negation, conjunction or the monadic *bind* operator, could be investigated in future work. Also, much work is yet to be done also in the area of correct-by-construction pretty printing.

Performance

In this thesis, I demonstrate that *parsing with derivatives and zippers* can be performant. In Chapter 2, I have shown that the performance of SILEX, a simple lexical analysis library built based on the approach, is no more than three times slower than state-of-the-art lexer generators, while being much more flexible and easier to integrate. In Chapter 6, I have shown that the SCALL1ON displays similar performance to that of the standard Scala parsing combinators library, while guaranteeing linear-time parsing, avoiding stack overflows, and providing extra features. The combination of SILEX and SCALL1ON displays performance no more than three times slower than that of ANTLR, an industrial-strength parser generator. The use of meta-programming techniques could be investigated to generate code and clear the performance gap of the discussed techniques with traditional parser generators, while hopefully not sacrificing too much on the flexibility and ease of integration aspects.

Unified Presentation of Lexical and Syntactic Analysis

This thesis shows that the same derivatives and zippers technique can be used on regular, LL(1) and general context-free expressions, paving the way towards a unified presentation of lexical and syntactic analysis. In Chapters 2, 5 and 7, have shown how the zipper-based approaches relate to well-established algorithms. In the case of regular expressions, I have shown that the zipper-based technique is reminiscent of Antimirov's partial derivatives and, thanks to memoisation, provides a lazy way to construct deterministic finite-state machines of near minimal sizes in practice. For LL(1) expressions, I have shown a correspondence with the traditional *stack-and-table* LL(1) parsing algorithm. Finally, I have shown how the zipper-based algorithm on general context-free expressions resembles more traditional algorithms such as Earley's or GLR. This thesis suggests that derivatives and zippers can be used exclusively to present both lexing and parsing in the context of, for instance, a compiler construction class. On the more theoretical side, the correspondence between datatype derivatives (zippers) and expression derivatives could be explored as future work.

Closing Message

Combining derivatives and zippers leads to lexical and syntactic analysis algorithms that are amenable to formal reasoning, are easy to integrate, support expressive interfaces, and are performant in practice.

A Regular Expression Derivation with Compaction

The code below shows how one could implement derivation with a *compaction* optimisation. The code is shown strictly for illustration.

```
sealed trait RegExpr {
  val isNullable: Boolean = this match {
    case Epsilon => true
    case Failure => false
    case Character(_) => false
    case Disjunction(left, right) =>
      left.isNullable || right.isNullable
    case Sequence(left, right) =>
      left.isNullable && right.isNullable
    case Repetition(_) => true
  }

  def deriveCompact(char: Char): RegExpr = this match {
    case Character(pred) if pred(char) =>
      Epsilon
    case Disjunction(left, right) =>
      left.deriveCompact(char) | right.deriveCompact(char)
    case Sequence(left, right) if left.isNullable =>
      (left.deriveCompact(char) ~ right) | right.deriveCompact(char)
    case Sequence(left, right) =>
      left.deriveCompact(char) ~ right
    case Repetition(inner) =>
      inner.deriveCompact(char) ~ this
    case _ => Failure
  }

  def ~(that: RegExpr): RegExpr = (this, that) match {
    case (Failure, _) => Failure
    case (_, Failure) => Failure
    case (Epsilon, _) => that
    case (_, Epsilon) => this
    case _ => Sequence(this, that)
  }

  def |(that: RegExpr): RegExpr = (this, that) match {
    case (Failure, _) => that
    case (_, Failure) => this
    case _ => Disjunction(this, that)
  }
}
case object Epsilon extends RegExpr
case object Failure extends RegExpr
case class Character(pred: Char => Boolean) extends RegExpr
case class Disjunction(left: RegExpr, right: RegExpr) extends RegExpr
case class Sequence(left: RegExpr, right: RegExpr) extends RegExpr
case class Repetition(inner: RegExpr) extends RegExpr
```

B Haskell Implementation of a Membership Checking Procedure using Derivatives and Zippers

The code below shows how one could can implement the technique regular language membership technique presented in Chapter 2. The code here to illustrate the applicability and conciseness of the approach.

```haskell
import Data.Function.Memoize
import Data.List (foldl')
import qualified Data.Set as Set

data RegExpr =
    Epsilon
  | Failure
  | Character Char
  | Disjunction RegExpr RegExpr
  | Sequence RegExpr RegExpr
  | Repetition RegExpr
 deriving (Eq, Ord, Show)
deriveMemoizable ''RegExpr

isNullable :: RegExpr -> Bool
isNullable Epsilon = True
isNullable (Disjunction l r) = isNullable l || isNullable r
isNullable (Sequence l r) = isNullable l && isNullable r
isNullable (Repetition _) = True
isNullable _ = False

type Context = [RegExpr]
type Zipper = [Context]

deriveZ :: Zipper -> Char -> Zipper
deriveZ z c = Set.elems $ Set.unions $ map up z
```

```haskell
  where
    up [] = Set.empty
    up (e : ctx)
      | isNullable e = Set.union (down e ctx) (up ctx)
      | otherwise = down e ctx

    down (Character c') ctx
      | c == c' = Set.singleton ctx
      | otherwise = Set.empty
    down (Disjunction l r) ctx = Set.union (down l ctx) (down r ctx)
    down (Sequence l r) ctx
      | isNullable l = Set.union (down l (r : ctx)) (down r ctx)
      | otherwise = down l (r : ctx)
    down r@(Repetition i) ctx = down i (r : ctx)
    down _ _ = Set.empty

isNullableZ :: Zipper -> Bool
isNullableZ z = any (all isNullable) z

data State = State { transitions :: Char -> State, isAccepting :: Bool }

build :: RegExpr -> State
build e = getState [[e]]
  where
    getState = memoize $ \ z ->
      let getNext = memoize (\ c -> getState (deriveZ z c))
      in State getNext (isNullableZ z)

run :: State -> [Char] -> Bool
run s cs = isAccepting (foldl' transitions s cs)
```

C JSON Values, Tokens and Kinds

The code below shows the definition of JSON tokens, kinds and values used in examples of Chapter 6.

```scala
object JSON {
  sealed trait Value
  case class BooleanValue(value: Boolean) extends Value
  case class NumberValue(value: Double) extends Value
  case class StringValue(value: String) extends Value
  case class NullValue() extends Value
  case class ArrayValue(values: Seq[Value]) extends Value
  case class ObjectValue(bindings: Seq[(StringValue, Value)]) extends Value

  sealed trait Token
  case class BooleanToken(value: Boolean) extends Token
  case class NumberToken(value: Double) extends Token
  case class StringToken(value: String) extends Token
  case class NullToken() extends Token
  case class SeparatorToken(sep: Char) extends Token
  case class SpaceToken() extends Token

  sealed trait Kind
  case object BooleanKind extends Kind
  case object NumberKind extends Kind
  case object StringKind extends Kind
  case object NullKind extends Kind
  case class SeparatorKind(char: Char) extends Kind
  case object IgnoreKind extends Kind

  object Kind {
    def of(token: Token): Kind = token match {
      case BooleanToken(_) => BooleanKind
      case NumberToken(_) => NumberKind
      case StringToken(_) => StringKind
      case NullToken() => NullKind
      case SeparatorToken(sep) => SeparatorKind(sep)
      case _ => IgnoreKind
    }
  }
}
```

D JSON Parser and Pretty Printer in SCALLION

The code below shows the definition of a JSON parser and pretty printer, as discussed in Chapter 6.

```
import scallion._
import JSON._

object JSONParser extends Parsers {
  override type Token = JSON.Token
  override type Kind = JSON.Kind
  override def getKind(token: Token): Kind = Kind.of(token)
  import SafeImplicits._

  val booleanSyntax: Syntax[Value] =
    accept(BooleanKind)({
      case BooleanToken(value) => BooleanValue(value)
    }, {
      case BooleanValue(value) => Seq(BooleanToken(value))
      case _ => Seq()
    })

  val numberSyntax: Syntax[Value] =
    accept(NumberKind)({
      case NumberToken(value) => NumberValue(value)
    }, {
      case NumberValue(value) => Seq(NumberToken(value))
      case _ => Seq()
    })

  val stringSyntax: Syntax[StringValue] =
    accept(StringKind)({
      case StringToken(value) => StringValue(value)
    }, {
```

```scala
      case StringValue(value) => Seq(StringToken(value))
      case _ => Seq()
    })

  val nullSyntax: Syntax[Value] =
    accept(NullKind)({
      case NullToken() => NullValue()
    }, {
      case NullValue() => Seq(NullToken())
      case _ => Seq()
    })

  implicit def separatorSyntax(char: Char): Syntax[Unit] =
    accept(SeparatorKind(char))({
      case SeparatorToken(_) => ()
    }, {
      case () => Seq(SeparatorToken(char))
    })

  lazy val arraySyntax: Syntax[Value] =
    ('['.skip ~ repsep(jsonSyntax, ',') ~ ']'.skip).map({
      case vs => ArrayValue(vs)
    }, {
      case ArrayValue(vs) => Seq(vs)
      case _ => Seq()
    })

  lazy val bindingSyntax: Syntax[(StringValue, Value)] =
    (stringSyntax ~ ':'.skip ~ jsonSyntax).map({
      case key ~ value => (key, value)
    }, {
      case (key, value) => key ~ value
    })

  lazy val objectSyntax: Syntax[Value] =
    ('{'.skip ~ repsep(bindingSyntax, ',') ~ '}'.skip).map({
      case bs => ObjectValue(bs)
    }, {
      case ObjectValue(bs) => Seq(bs)
      case _ => Seq()
    }

  lazy val jsonSyntax: Syntax[Value] = recursive {
    arraySyntax | objectSyntax | booleanSyntax |
      numberSyntax | stringSyntax.up[Value] | nullSyntax
  }

  val jsonParser: Parser[Value] = Parser(jsonSyntax)

  def apply(it: Iterator[Token]): ParseResult[Value] = jsonParser(it)
}
```

Bibliography

Adams, M. D., Hollenbeck, C., and Might, M. (2016). On the complexity and performance of parsing with derivatives. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 224–236, New York, NY, USA. ACM.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Aho, A. V. and Ullman, J. D. (1972). *The theory of parsing, translation, and compiling. 1: Parsing*. Prentice-Hall.

Ammann, S. S. (2021). Formal proofs about formal languages. https://github.com/soph2018/toc_on_coq.

Antimirov, V. (1996). Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319.

Ausaf, F., Dyckhoff, R., and Urban, C. (2016). Posix lexing with derivatives of regular expressions. *Archive of Formal Proofs*. http://isa-afp.org/entries/Posix-Lexing.html, Formal proof development.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., et al. (1960). Report on the algorithmic language algol 60. *Numerische Mathematik*, 2(1):106–136.

Barenghi, A., Mainardi, N., and Pelosi, G. (2018). Systematic parsing of x. 509: eradicating security issues with a parse tree. *Journal of Computer Security*, 26(6):817–849.

Baudelaire, C. (1855). Les Fleurs du mal. *Revue des Deux Mondes (1829-1971)*, 10(5):1079–1093.

Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., and Strollo, O. (2012). When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113.

Becket, R. and Somogyi, Z. (2008). Dcgs+ memoing= packrat parsing but is it worth it? In *International Symposium on Practical Aspects of Declarative Languages*, pages 182–196. Springer.

Bekić, H. (1984). Definable operations in general algebras, and the theory of automata and flowcharts. In *Programming Languages and Their Definition*, pages 30–55. Springer.

Boulton, R. J. (1996). Syn: a single language for specifiying abstract syntax tress, lexical analysis, parsing and pretty-printing. Technical report, University of Cambridge, Computer Laboratory.

Brachthäuser, J. I., Rendel, T., and Ostermann, K. (2016). Parsing with first-class derivatives. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 588–606, New York, NY, USA. ACM.

Brüggemann-Klein, A. (1993). Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213.

Brüggemann-Klein, A. and Wood, D. (1992). Deterministic regular languages. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 173–184. Springer.

Brzozowski, J. A. (1964). Derivatives of regular expressions. In *Journal of the ACM*. Citeseer.

Burge, W. H. (1975). Recursive programming techniques.

Caron, P., Champarnaud, J.-M., and Mignot, L. (2011). Partial derivatives of an extended regular expression. In *International Conference on Language and Automata Theory and Applications*, pages 179–191. Springer.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and control*, 2(2):137–167.

Cocke, J. (1969). Programming languages and their compilers: Preliminary notes.

Čulik II, K. and Cohen, R. (1973). Lr-regular grammars—an extension of lr (k) grammars. *Journal of Computer and System Sciences*, 7(1):66–96.

Danielsson, N. A. (2010). Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 285–296, New York, NY, USA. ACM.

Darragh, P. and Adams, M. D. (2020). Parsing with zippers (functional pearl). *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–28.

Degener, J. (1995). Ansi c grammar, lex specification. https://www.lysator.liu.se/c/ANSI-C-grammar-l.html.

Delaware, B., Suriyakarn, S., Pit-Claudel, C., Ye, Q., and Chlipala, A. (2019). Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29.

Deransart, P., Jourdan, M., and Lorho, B. (1988). *Attribute grammars: definitions, systems and bibliography*, volume 323. Springer Science & Business Media.

DeRemer, F. L. (1969). *Practical translators for LR (k) languages*. PhD thesis, Massachusetts Institute of Technology.

DeRemer, F. L. (1971). Simple lr (k) grammars. *Communications of the ACM*, 14(7):453–460.

Django Software Foundation (2021). Django: The web framework for perfectionists with deadlines. https://github.com/django/django.

Donnely, C. and Stallman, R. (2015). Gnu bison–the yacc-compatible parser generator. *Free Software Foundation, Cambridge*.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Edelmann, R. (2019). Scallion. https://github.com/epfl-lara/scallion.

Edelmann, R., Hamza, J., and Kunčak, V. (2020). Zippy LL(1) parsing with derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1036–1051.

Fokker, J. (1995). Functional parsers. In *International School on Advanced Functional Programming*, pages 1–23. Springer.

Ford, B. (2002). Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA. ACM.

Ford, B. (2004). Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA. ACM.

Frost, R. and Launchbury, J. (1989). Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121.

Gibbons, J. (2003). *Origami Programming*.

Gibbons, J. and Wu, N. (2014). Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347.

Godefroid, P., Kiezun, A., and Levin, M. Y. (2008). Grammar-based whitebox fuzzing. In Gupta, R. and Amarasinghe, S. P., editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 206–215. ACM.

Graham-Cumming, J. (2017). Incident report on memory leak caused by Cloudflare parser bug. https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/.

Grimm, R. (2004). Practical packrat parsing. Technical report, New York University.

Hamza, J. and Edelmann, R. (2019). Scallion proofs. https://github.com/epfl-lara/scallion-proofs.

Haoyi, L. (2021). Fastparse 2.2.2. http://www.lihaoyi.com/fastparse/.

Henriksen, I., Bilardi, G., and Pingali, K. (2019). Derivative grammars: A symbolic approach to parsing with derivatives. *Proc. ACM Program. Lang.*, 3(OOPSLA):127:1–127:28.

Herman, G. (2020). Faster general parsing through context-free memoization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1022–1035.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001a). Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001b). Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65.

Huet, G. (1997). The zipper. *Journal of functional programming*, 7(5):549–554.

Hutton, G. (1992). Higher-order functions for parsing. *Journal of functional programming*, 2(3):323–343.

Hutton, G. and Meijer, E. (1996). Monadic parser combinators.

Izmaylova, A., Afroozeh, A., and Storm, T. v. d. (2016). Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program manipulation*, pages 1–12.

Jin, W., Sun, Y., Wang, N., and Zhang, X. (2017). Why users purchase virtual products in mmorpg? an integrative perspective of social presence and user engagement. *Internet Research*.

Johnson, M. (1995). Memoization of top down parsing. *arXiv preprint cmp-lg/9504016*.

Johnson, S. C. et al. (1975). *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ.

Johnstone, A. and Scott, E. (1998). Generalised recursive descent parsing and follow-determinism. In *International Conference on Compiler Construction*, pages 16–30. Springer.

Jourdan, J.-H., Pottier, F., and Leroy, X. (2012). Validating lr (1) parsers. In *European Symposium on Programming*, pages 397–416. Springer.

Kandra Labs, Inc (2021). Zulip server and webapp - powerful open source team chat. https://github.com/zulip/zulip.

Kasami, T. (1966). An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*.

Klein, G. (2010). Jflex user's manual. http://www.jflex.de.

Knuth, D. E. (1965). On the translation of languages from left to right. *Information and control*, 8(6):607–639.

Koprowski, A. and Binsztok, H. (2010). Trx: A formally verified parser interpreter. In *European Symposium on Programming*, pages 345–365. Springer.

Koukoutos, E. (2019). *Scaling Functional Synthesis and Repair*. PhD thesis, EPFL.

Krishnaswami, N. R. and Yallop, J. (2019). A typed, algebraic approach to parsing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 379–393, New York, NY, USA. ACM.

Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA. ACM.

Kuraj, I., Kuncak, V., and Jackson, D. (2015). Programming with enumerable sets of structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 37–56, New York, NY, USA. Association for Computing Machinery.

LAMP EPFL and Lightbend, Inc (2019). Scala parser combinators. https://github.com/scala/scala-parser-combinators.

Lasser, S., Casinghino, C., Fisher, K., and Roux, C. (2019). A verified ll (1) parser generator. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Leijen, D. and Meijer, E. (2001). Parsec: Direct style monadic parser combinators for the real world.

Leiß, H. (1991). Towards kleene algebra with recursion. In *International Workshop on Computer Science Logic*, pages 242–256. Springer.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115.

Lesk, M. E. and Schmidt, E. (1975). Lex: A lexical analyzer generator.

Levine, J. (2009). *Flex & Bison: Text Processing Tools*. O'Reilly Media, Inc.

Lewis, II, P. M. and Stearns, R. E. (1968). Syntax-directed transduction. *J. ACM*, 15(3):465–488.

Madhavan, R., Mayer, M., Gulwani, S., and Kuncak, V. (2015). Automating grammar comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 183–200, New York, NY, USA. Association for Computing Machinery.

Maillard, B. L. (2020). Scallion python parser. https://github.com/benoitmaillard/scallion-python-parser.

Marlow, S. (2010). Haskell 2010 language report.

Matsuda, K. and Wang, M. (2013). Flippr: A prettier invertible printing system. In *European Symposium on Programming*, pages 101–120. Springer.

McBride, C. (2001). The derivative of a regular type is its type of one-hole contexts (extended abstract).

McBride, C. (2008). Clowns to the left of me, jokers to the right (pearl) dissecting data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–295.

McBride, C. and Paterson, R. (2008). Applicative programming with effects. *Journal of functional programming*, 18(1):1–13.

McPeak, S. and Necula, G. C. (2004). Elkhound: A fast, practical glr parser generator. In *International Conference on Compiler Construction*, pages 73–88. Springer.

Might, M. (2010). A non-blocking lexing toolkit for Scala in less than 800 lines of code, from regex derivatives. http://matt.might.net/articles/nonblocking-lexing-toolkit-based-on-regex-derivatives/.

Might, M., Darais, D., and Spiewak, D. (2011). Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 189–195, New York, NY, USA. ACM.

Moors, A., Piessens, F., and Odersky, M. (2008). Parser combinators in scala. *CW Reports*, 54.

Moreira, N., Pereira, D., and de Sousa, S. M. (2012). Deciding regular expressions (in-) equivalence in coq. In *International Conference on Relational and Algebraic Methods in Computer Science*, pages 98–113. Springer.

Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98.

Odersky, M., Blanvillain, O., Liu, F., Biboudis, A., Miller, H., and Stucki, S. (2017). Simplicitly: Foundations and applications of implicit function types. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29.

Okhotin, A. (2001). Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535.

Okhotin, A. (2013). Conjunctive and boolean grammars: the true general case of the context-free grammars. *Computer Science Review*, 9:27–59.

Omanashvili, V. (2019). Json generator. https://www.json-generator.com. Accessed 2019-11-20.

Owens, S., Reppy, J., and Turon, A. (2009). Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190.

Pager, D. (1977). A practical general method for constructing lr (k) parsers. *Acta Informatica*, 7(3):249–268.

Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

Parr, T. (2019). Grammars written for antlr v4; expectation that the grammars are free of actions. https://github.com/antlr/grammars-v4/tree/master/json. Accessed 2019-11-22.

Parr, T. and Fisher, K. (2011). Ll(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 425–436.

Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., and Yorgey, B. (2018). *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

Pottier, F. (2009). Lazy least fixed points in ml.

Pottier, F. (2013). fix: An ocaml library that provides facilities for memoization and fixed points. https://gitlab.inria.fr/fpottier/fix.

Pratt, V. R. (1973). Top down operator precedence. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 41–51.

Prokopec, A. (2019). Scalameter: Automate your performance testing today. https://scalameter.github.io/. Accessed 2019-11-20.

Radul, A. (2009). *Propagation networks: A flexible and expressive substrate for computation*. PhD thesis, Massachusetts Institute of Technology.

Ramananandro, T., Delignat-Lavaud, A., Fournet, C., Swamy, N., Chajed, T., Kobeissi, N., and Protzenko, J. (2019). Everparse: Verified secure zero-copy parsers for authenticated message formats. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1465–1482.

Ranta, A. (2004). Grammatical framework. *Journal of Functional Programming*, 14(2):145.

Ranta, A. (2011). *Grammatical framework: Programming with multilingual grammars*, volume 173. CSLI Publications, Center for the Study of Language and Information Stanford.

Rendel, T. and Ostermann, K. (2010). Invertible syntax descriptions: Unifying parsing and pretty printing. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 1–12, New York, NY, USA. ACM.

Reps, T. (1998). "maximal-munch" tokenization in linear time. *ACM Trans. Program. Lang. Syst.*, 20(2):259–273.

Ronacher, A. (2021). Flask: Web development, one drop at a time. https://flask.palletsprojects.com/en/2.0.x/.

Sakai, I. (1962). *Syntax in universal translation.* Her Magesty's Stationary Office.

Scalaz (2020). Scalaz: a scala library for functional programming. https://scalaz.github.io/.

Scott, E. and Johnstone, A. (2010). Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189.

Sipser, M. (2012). *Introduction to the Theory of Computation.* Cengage learning.

Spiewak, D. (2018). Parseback. https://github.com/djspiewak/parseback.

Stearns, R. E. and Lewis, P. (1969). Property grammars and table machines. *Information and Control*, 14(6):524–549.

Strickland, D. (2020). Gta 5 had record microtransaction earnings in april 2020. https://www.tweaktown.com/news/72714/gta-had-record-microtransaction-earnings-in-april-2020/index.html.

Swierstra, S. D. and Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. In *International School on Advanced Functional Programming*, pages 184–207. Springer.

t0st (2021). How I cut GTA Online loading times by 70%. https://nee.lv/2021/02/28/How-I-cut-GTA-Online-loading-times-by-70/.

Tomita, M. (1987). An efficient augmented-context-free parsing algorithm. *Computational linguistics*, 13:31–46.

Tomita, M. (1988). Graph-structured stack and natural language parsing. In *26th Annual Meeting of the Association for Computational Linguistics*, pages 249–257.

Tomita, M. (2012). *Generalized LR parsing.* Springer Science & Business Media.

Traytel, D. and Nipkow, T. (2013). Verified decision procedures for mso on words based on derivatives of regular expressions. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 3–12, New York, NY, USA. Association for Computing Machinery.

Typelevel (2020). Cats: Lightweight, modular, and extensible library for functional programming. https://typelevel.org/cats/.

Van Rossum, G. and Drake, F. L. (2009). *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA.

Wadler, P. (1995). Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76.

Wiebe, E. N., Lamb, A., Hardy, M., and Sharek, D. (2014). Measuring engagement in video game-based environments: Investigation of the user engagement scale. *Computers in Human Behavior*, 32:123–132.

Wiedijk, F. (2012). Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285:85 – 100. Proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10).

Wu, C., Zhang, X., and Urban, C. (2011). A formalisation of the myhill-nerode theorem based on regular expressions (proof pearl). In *International Conference on Interactive Theorem Proving*, pages 341–356. Springer.

Yorgey, B. (2009). The typeclassopedia. *The Monad. Reader Issue 13*, page 17.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208.

Romain Edelmann

Software Engineer
(ing. info. dipl. EPF)

Nationality:	Swiss
Birthdate:	November 12, 1988
Phone:	+41 79 699 29 65
Email:	romain.edelmann@gmail.com
Adress:	Romain Edelmann Route de Cheseaux 5 1054 Morrens, VD Switzerland
LinkedIn:	https://www.linkedin.com/in/romain-edelmann/
GitHub:	https://github.com/redelmann/

Education

École Polytechnique Fédérale de Lausanne **Computer Science PhD** Thesis: *Efficient Parsing with Derivatives and Zippers* Teaching: Introduction to Programming, Parallel and Concurrent Programming, Functional Programming, Formal Verification. Awards: Award for Teaching Excellence (2019)	09.2015 - 05.2021
École Polytechnique Fédérale de Lausanne **Computer Science Master** Specialisation: *Foundation of Software* Thesis: *BIP in Functional Programming Languages* GPA: 5.68 / 6	09.2012 - 02.2015
National University of Singapore **Computer Science Bachelor** Year abroad as part of an exchange program.	08.2011 - 06.2012
École Polytechnique Fédérale de Lausanne **Computer Science Bachelor** GPA: 5.38 / 6	09.2009 - 06.2012
Gymnase de Burier **Maturité Gymnasiale** Thesis: *Game Theory and Artificial Intelligence* Options: Physics and Maths, Advanced Maths, Chemistry (complementary)	09.2004 - 06.2008

Professional Experience

École Polytechnique Fédérale de Lausanne **Scientific Collaborator** Elaboration of technical solutions and drafting of teaching aids within the EduNum project for the teaching of computer science as a compulsory subject in high schools.	Since 02.2021
Fondation Suisse pour les Téléthèses, Neuchâtel **Software Engineer** Development of IT solutions for people with disabilities in the context of civil service.	02.2015 - 08.2015
Google, Zürich **Software Engineer Intern** Development of monitoring solutions for the data processing systems of the YouTube platform.	08.2013 - 02.2014
École Polytechnique Fédérale de Lausanne **Student Assistant** Assisted students during practice sessions, graded exams and projects, and proctored exams for several introductory programming courses as well as for an information science course.	09.2010 - 02.2015
EDSI-Tech, Lausanne **Associate, Developer** Development of websites and iOS and Android applications.	09.2009 - 08.2012

Languages

French:	Mother tongue
English:	C2

Publications

Romain Edelmann, Jad Hamza, and Viktor Kunčak. "Zippy LL (1) Parsing with Derivatives." *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020).

Romain Edelmann, and Viktor Kunčak. "Neural-Network Guided Expression Transformation." *arXiv preprint arXiv:1902.02194* (2019).

Romain Edelmann, Simon Bliudze, and Joseph Sifakis. "Functional BIP: Embedding Connectors in Functional Programming Languages." *Journal of Logical and Algebraic Methods in Programming* 92 (2017): 19-44.

Romain Edelmann. "Behaviour-Interaction-Priority in Functional Programming Languages: Formalisation and Implementation of Concurrency Frameworks in Haskell and Scala." *Master Thesis* (2015).